Lab 4: ROM-based Game Access Control on FPGA

Rogelio Lopez

1812758

ECE 5440

Introduction

The mental binary math game is a game that requires 1 player. A player will use a button to generate a random number. The player must enter a value using the player switches so that both numbers add up to 15, or hexadecimal value F. The 4 switches represent a binary number. Each switch represents a binary digit. A binary digit (bit) can take on the value of either a 1 or a 0. When a switch is down, the bit is set to 0; when the switch is up, the bit is set to 1. When a bit is set to one, it represents a number that is equal to a power of 2. The right most switch represents the least significant bit, or bit 0. This means that the number represented by this bit when set to 1, will be $\mathbf{2^0}$. The next switch will represent $\mathbf{2^1}$, the next will be $\mathbf{2^2}$, and the last will be $\mathbf{2^3}$. When the player successfully sums both values to a 15, the 2nd LED on the left-hand side with light up in red, indicating that the player has successfully entered the correct input. When the sum is not equal to 15, the left most LED will light up in red. The objective of the game is to get as many successful inputs as possible in 99 seconds. Additionally, the game has now gained an authentication feature to allow players to have a go at the game. Previously, players were able to just turn on the board and were allowed to play. Now, players must enter a password using the third button and the 4 left-most switches to gain access to the game. Once authenticated, the 7 segment displays will display the timer of 99 seconds, a random number, the sum, and the player's value which should be 0 as no input has been entered with the load button.

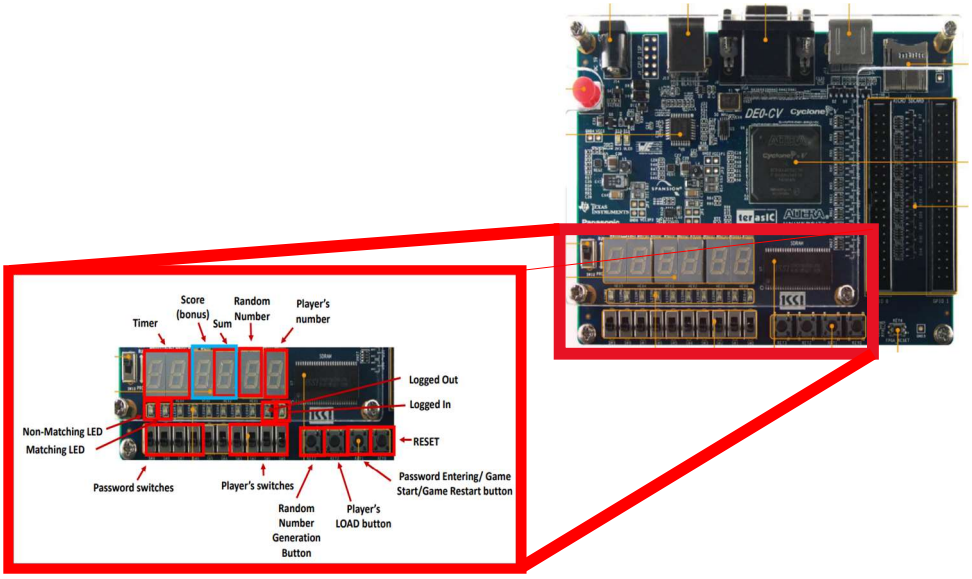

Figure 1. *FPGA Board DE0-CV*

System Design Architecture

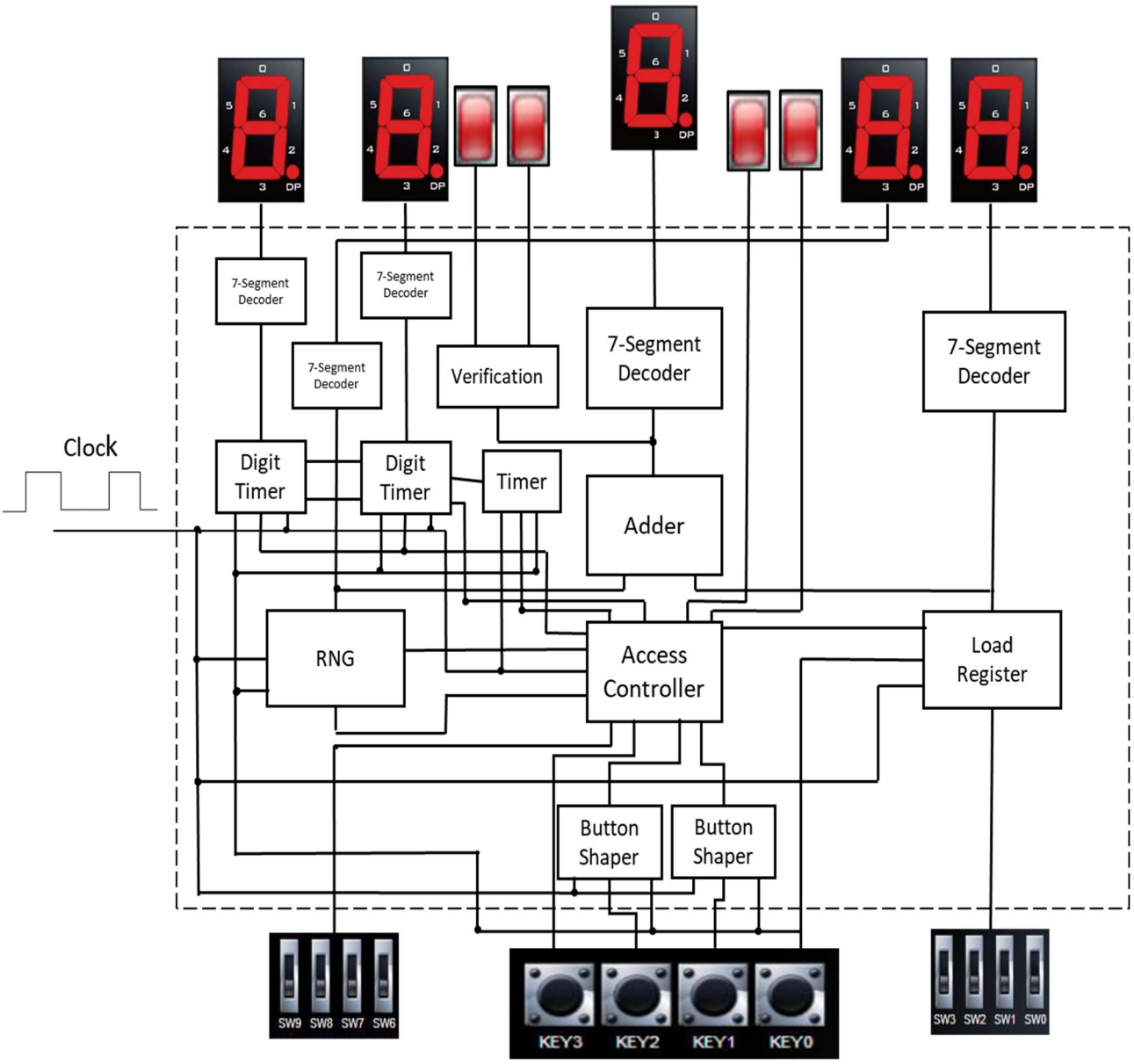

Figure 2. *Top-level module architecture*

Above is a simplification of the top-level architecture used to design the binary mental game with an authentication feature and timer. It takes 13 input signals – 4 switches for the participating player, 4 switches to enter the password, 4 buttons, and 1 clock that runs at 50 megahertz. The player inputs are entered through the flipping of switches. The buttons each have different functions such as generating a random number, sending a load signal, and resetting the game. One button has multiple functions depending on which state the finite state machine is in (More on this in the section below). Then, there are 39 outputs; 7 output signals are sent from

each decoder to the 7-segment displays and 4 output signals are sent to LEDs. Two of the 7-segment displays will display the timer for the game. One will display the random number generated when a button is pressed, and another will display the value the player has loaded.

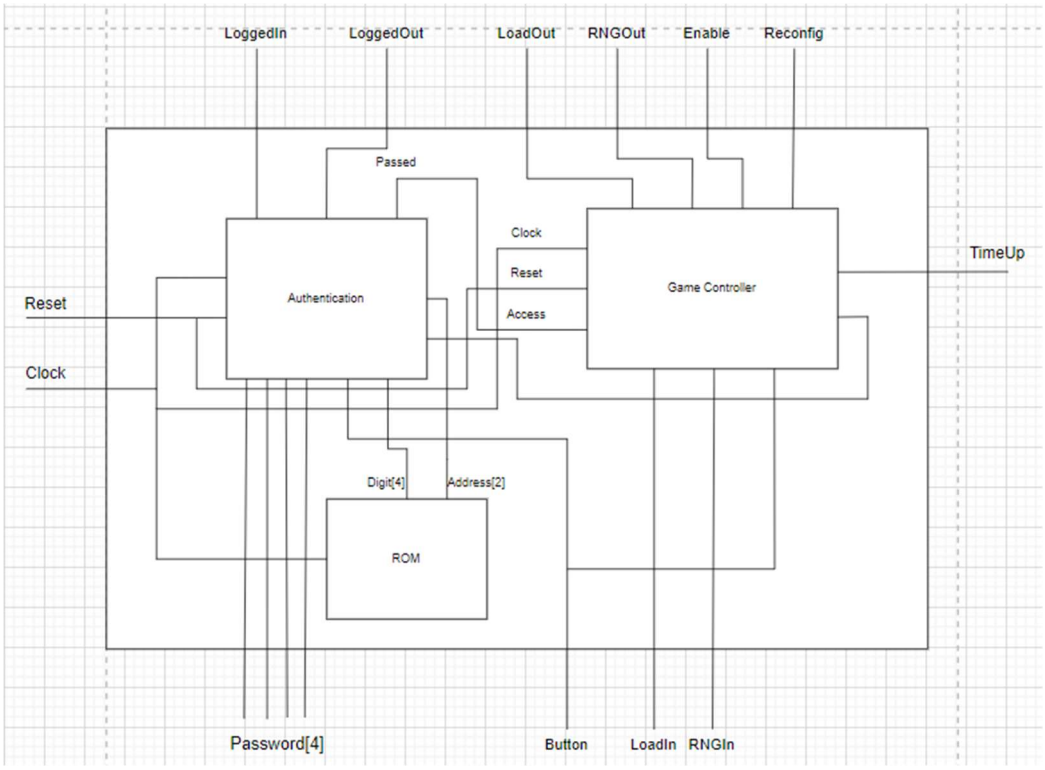

Figure 3. *Access Controller Diagram*

The access controller is made up of 3 modules: The Game Controller, the Authentication, and the ROM. Two of the three modules use high-level finite state machines to keeps strangers away from our machine and does not allow users to play unless unlocked by entering a password in the form of 4 consecutive digits. The access controller takes in 10 inputs – 4 inputs are the bits that make up a digit the user enters, 1 is for the button with multiple functions, 1 is for when the time is up in the game, 1 is to generate a random number, 1 is for a player to load a value to the display, 1 is the Reset, and 1 is the clock. It is capable of authentication through the high-level state machine programmed in the authentication module.

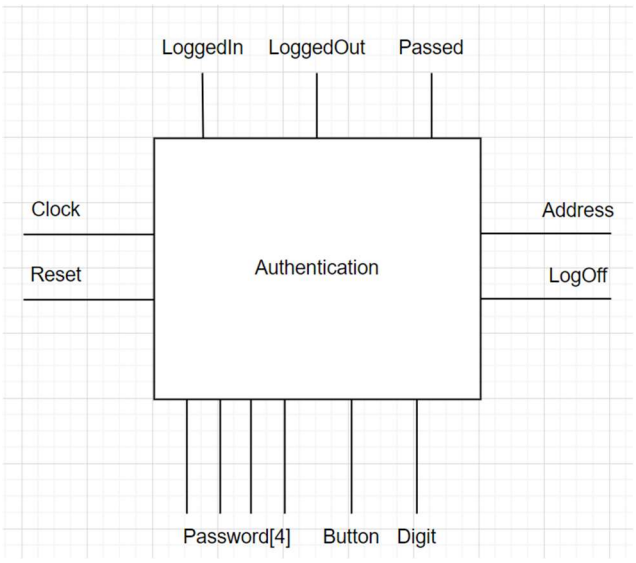

Figure 4. *Authentication State Machine Diagram*

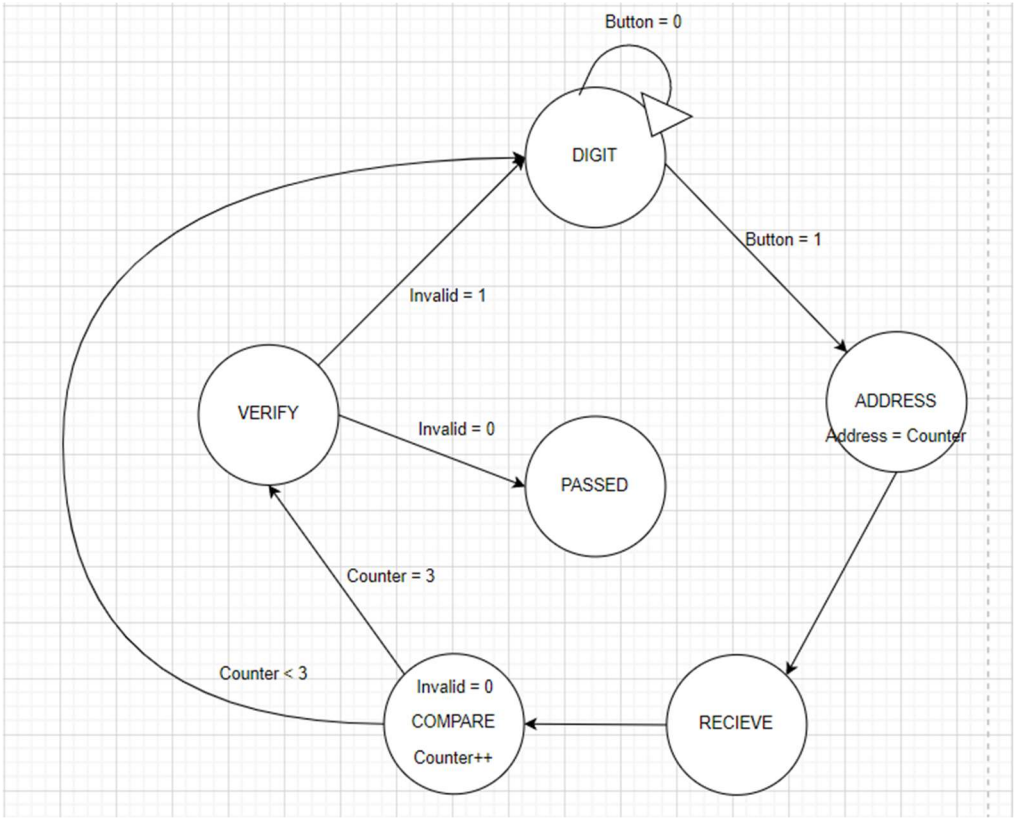

Figure 5. *Authentication State Machine Diagram*

There are 4 digits to check. The diagram above illustrates the recycling of states using a counter. Rather than having 4 states to check each digit, we have 4 states that check all 4 digits as the machine cycles through these states 4 times to check 4 digits. The first state waits until the

button is pressed. Then the machine moves onto the ADDRESS state where it sends out a 2 bit signal to a Read-Only Memory (ROM) module.

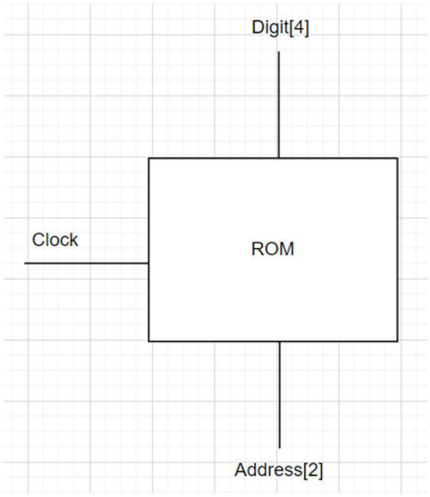

Figure 6. *Read Only Memory module Diagram*

The ROM receives a 2-bit address in the RECEIVE state and sends back the 4-bit digit that the user is supposed to enter. The next state compares the value received from the user and from the ROM. When the user's entered digit is incorrect, a flag is set to let the machine know that the user has entered the incorrect password. Now, if we are not checking the 4th digit the counter is increased and the machine goes back to the DIGIT state to receive a button signal. After checking 4 digits, and the flag is raised, the machine will not let the user play or load any value onto the display and expect the first digit to be entered once again. When a user enters the correct password, the machine will send a signal to the game controller to allow the player's load signal, the generation of a random number, and set the game timer to 99 seconds.

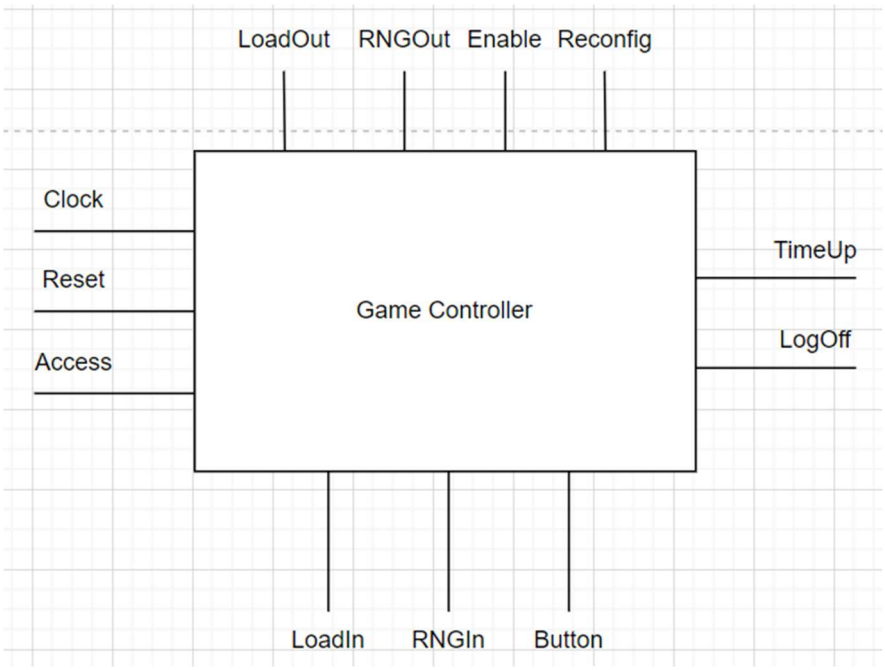

Figure 7. *Game Controller module Diagram*

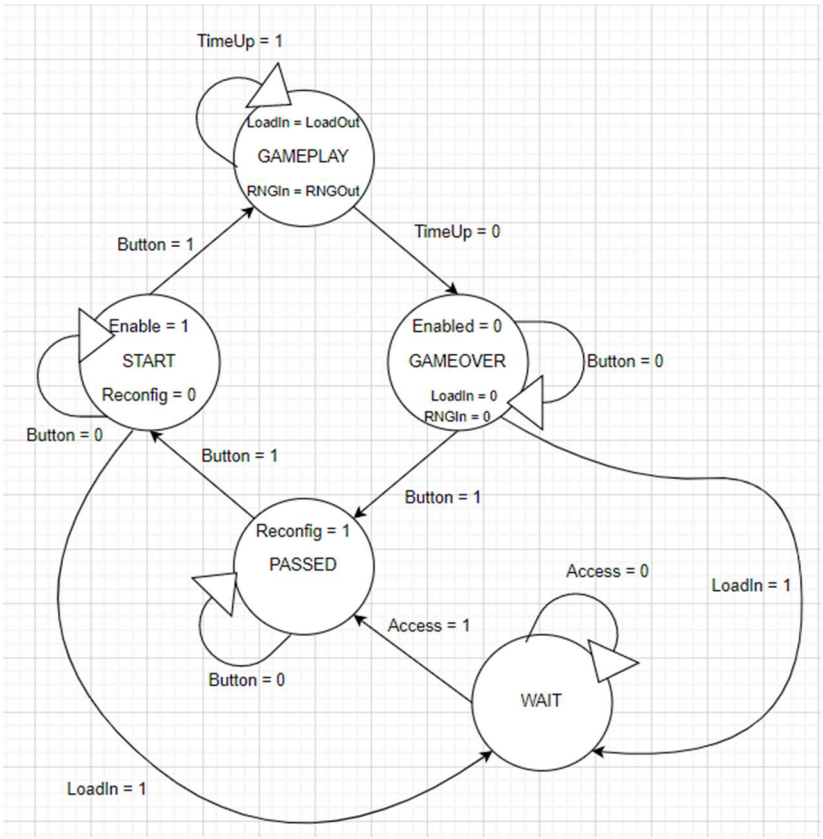

Figure 8. *Game Controller State Machine Diagram*

The game controller begins in the WAIT state until it gets the access granted signal. The machine waits for the player to start the game by pressing the button. The enable signal will then be sent out to start the timer and all functionality for the switches and buttons are enabled. Once the timer reaches an end, the timer will send a signal to the game controller, and the machine will block any functionality that was allowed for gameplay. The machine then waits in the GAMEOVER state for button press to reset the game. There is also a log out feature included. To log out, the user must simply press the Load button in either the START or GAMEOVER state and the user will then need to enter the password to be able to play the game again. The access controller has 6 output signals – 2 signals sent to the LEDs to indicate whether a user has logged in or not, 1 player load signal, 1 random number generator signal, 1 enable signal to start the game, and 1 signal to reset the game timer to 99 seconds.

The timer is comprised of multiple components: The digit timer and the counter. For the timer to work, 2 digit timers and 3 counters were used. Below we go over how 3 counters were used to create a one second timer.

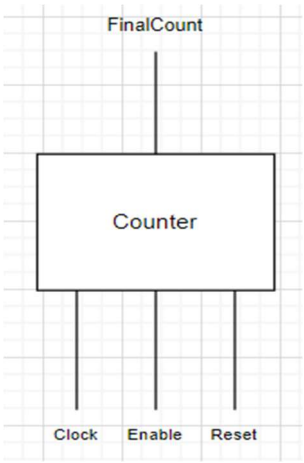

Figure 9. *Counter Diagram*

The counter is simple. It consists of 3 inputs – a clock, a reset, and a signal to enable the clock. When the enable signal is high, the counter will count and send a high output signal each time the counter reaches that pre-determined value. For example, as mentioned before, 3 counter modules were used to create a one second timer. The clock is a 50-megahertz clock so one second will go by every 50 million clock cycles. The use of multiple counters was due to the expensive nature of modules with the capacity of 26 bits to hold a value of 50 million clock cycles, therefore, we split up the count into 3 different counters.

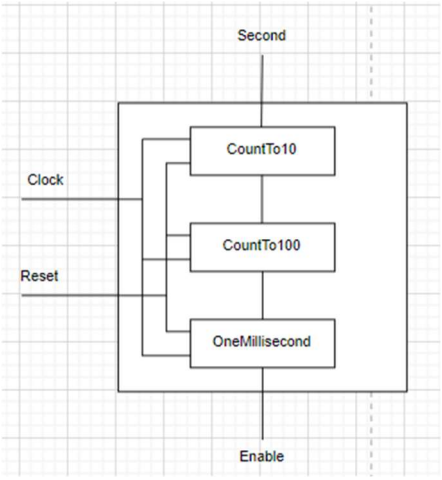

Figure 10. *One Second Counter Diagram*

First, we have a counter that counts 50 thousand clock cycles, then we have another that counts to 100. This counter counts the number of times the first counter counts 50 thousand clock cycles. Then we have another counter that counts to 10. The third counter counts the number of times the second counter counts to 100, or 5 million clock cycles. When the third counter reaches 10, 50 million clock cycles have gone by, or one second. To make this more cost-effective, we implemented linear-feedback shift registers inside the counters. The linear feedback shift register works by shifting the bits one position to the left and replaces the vacated bit by the exclusive or(xor) of the bit shifted off and the bit previously at a given tap position in the register. The code for the these counters will be shown in the Appendix.

The output signal of the one second counter is sent to a digit timer so that it subtracts 1 from the current value it holds. The digit timer is explained below.

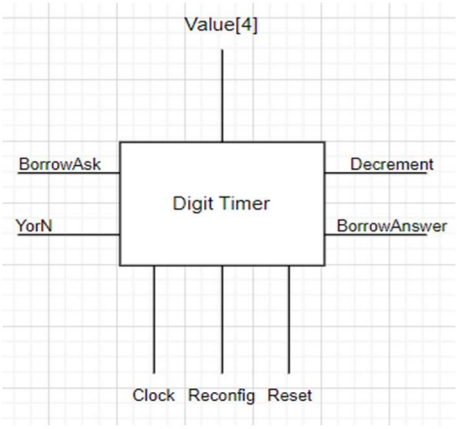

Figure 11. *Digit Timer Diagram*

The digit timer completes the game timer by controlling what is displayed on the board in terms of seconds. Two digit timers are used - 1 digit timer for the ones digit and another for the tenths digit. The digit timer is composed of 5 input signals – a clock, a reset, a reconfig signal to set the digit to 9, a decrement signal to know when to subtract one from the current value, and a signal that tells the digit whether it can borrow from the next digit. The digit timer also has 6 output signals – 4 for the value that should be displayed, 1 signal that asks the next digit to borrow, and another signal that answers the next digit if it can borrow. Below is the full module of the 99 second timer.

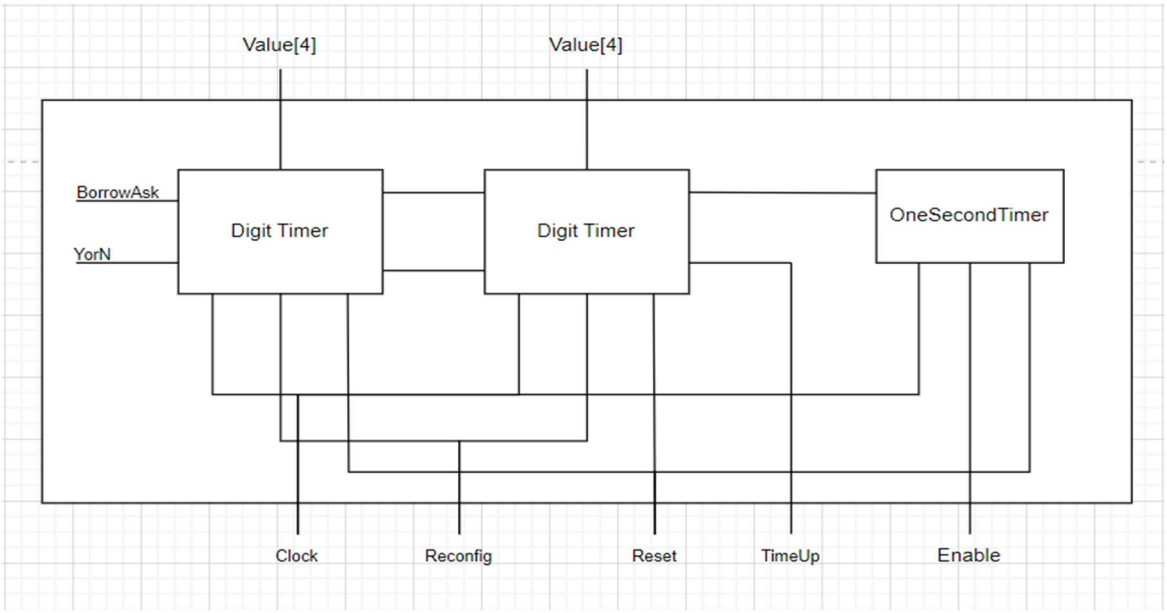

Figure 12. *Digit Timer Diagram*

When the game is started by the player, the player is given access to the random number generator. The random number generator is comprised of a counter. Below is the diagram.

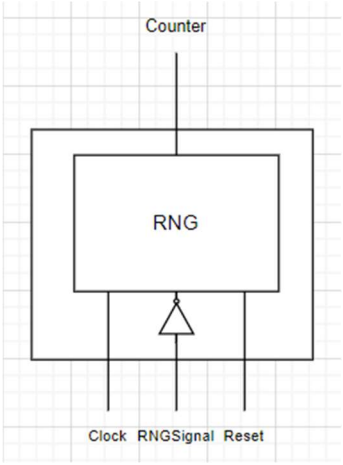

Figure 13. *Random Number Generator Module Diagram*

The random number generator is a counter that takes an enable signal but it is inverted to begin the counting process. The random number generator counter is based on linear-feedback shift registers using the XNOR gate feedback implementation, so our counter starts off with 0. The button for the random number generator is not configured by a button shaper so when pressed, the signal is a low input. The counter counts the number of clock cycles the button is pressed, and only counts up to 14. When the counter reaches 14, the counter resets to 0. One thing to note is that the counter will continue to count from where it stopped counting the last time it was pressed.

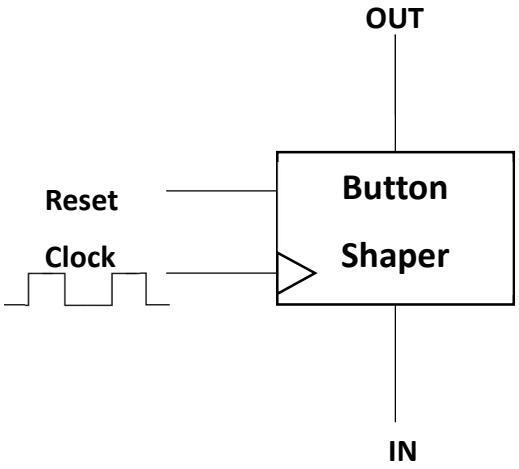

Figure 14. *Button Shaper Diagram*

The button shaper is another finite state machine takes in 3 inputs and has 1 output. The clock goes through many cycles when a button is pressed which would give a reading of a button signal many times rather than once. The purpose of the button shaper is to fix this issue and achieve a single reading of the press of a button. Below we illustrate the state machine diagram.

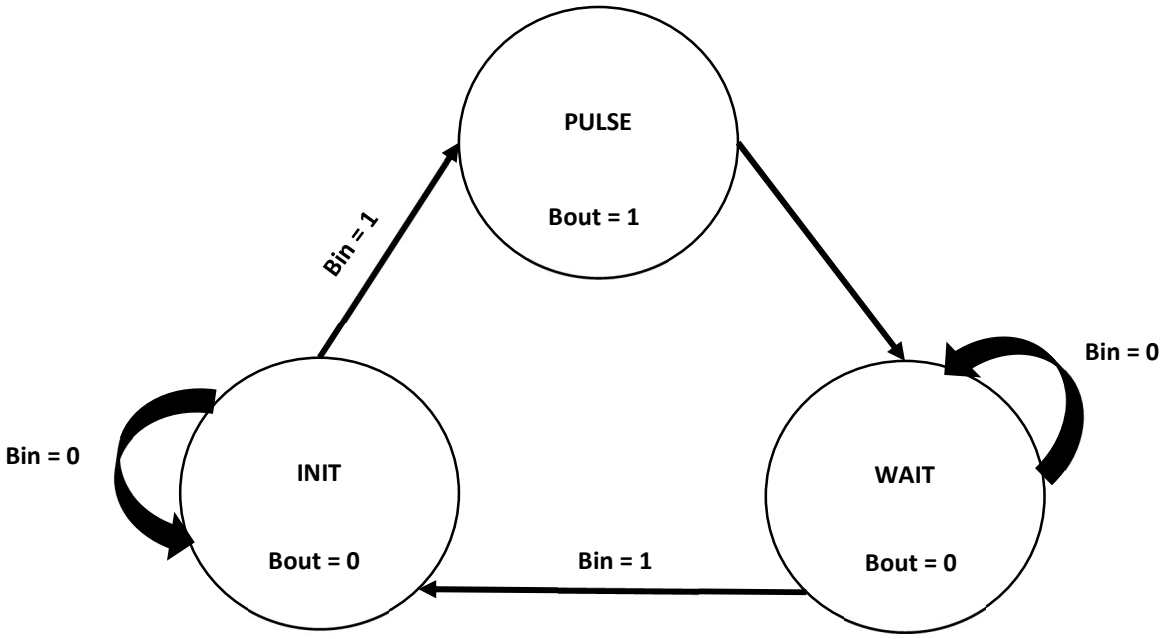

Figure 15. *Button Shaper State Machine Diagram*

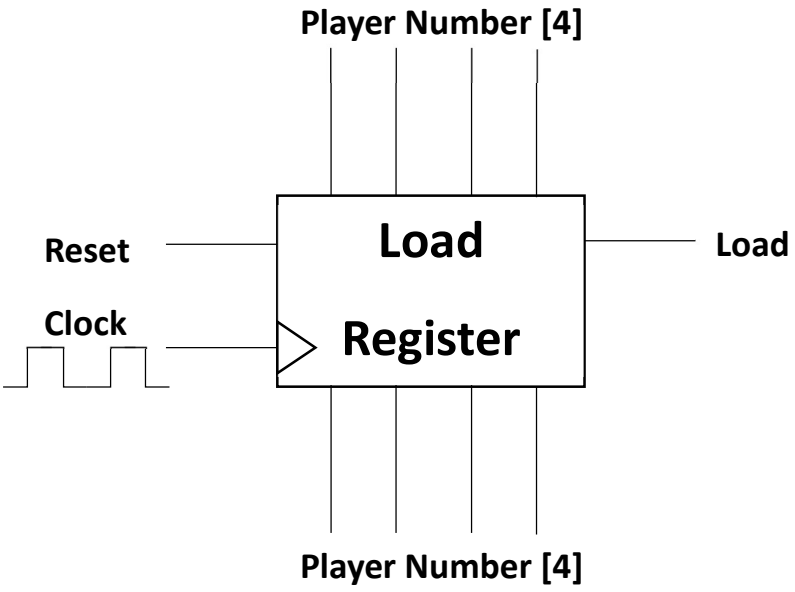

Figure 16. *Load Register Diagram*

The load register takes in a Reset signal, a clock signal, a load signal, and a 4 bit input that represent a number less than or equal to 15. The purpose of this load register is to prevent every single flip of a switch to display on the board. Rather than having every flip of a switch display on the board, the 7-segment display will only illuminate the value sent by this load register when a player sends a load signal to it.

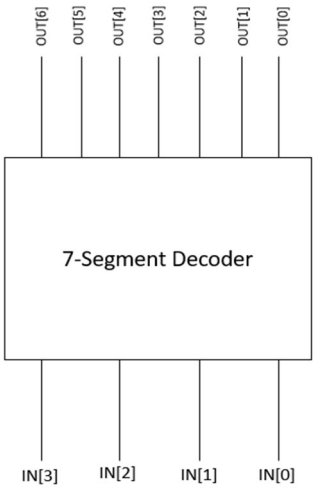

Figure 17. *7-Segment Decoder Diagram*

The 7-Segment Decoder will take in 4 inputs, either a 1 or a 0. The four inputs represent a decimal between 0 and 15. Based on those numbers, the decoder will set specific outputs to 0 to power on specific segments on a 7-segment display. Below is the Verilog code for the module.

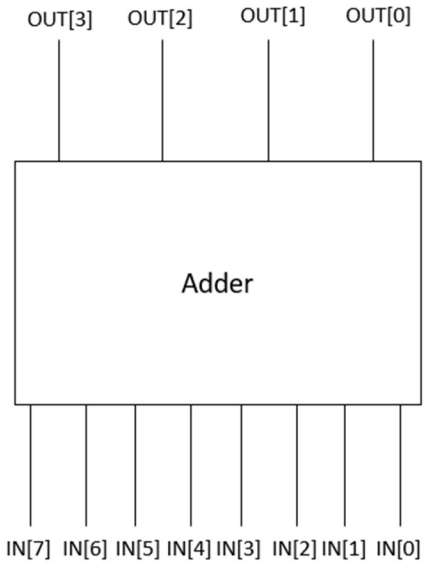

Figure 18. *Adder Diagram*

The Adder will take in 8 inputs, either a 1 or a 0, the 8 inputs represent 2 decimals between 0 and 15. The first 4 inputs on the right-hand side, will be Player One's input, the next 4 on the left-hand side will be Player Two's input. The module will add both numbers and send 4 output signals that represent the sum of the two decimal inputs. The Verilog module code is below.

Simulation

When testing the 7-segment decoder, I tested every possible input there is for a 4-bit binary number: 16 possible inputs. Each test case had its own output, so each output would be different. The outputs were determined using the FPGA Board's User Manual by going through and setting the corresponding signals low to light segments on the display.

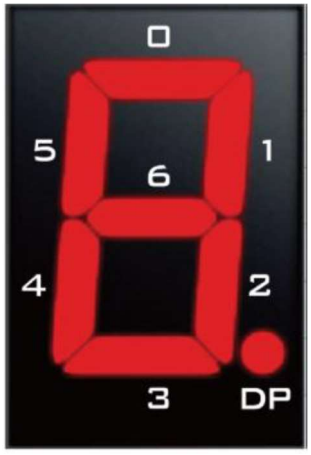

Figure 19. *7-Segment Display Diagram*

As an example, if a player decided to enter a number 4 as their input, their 7-segment display must also light up a 4. To accomplish this, the decoder sends out a low signal to segments 1, 2, 5, and 6. The rest of the segments are sent a high signal, so they won't turn on. Below are the waveforms for each case simulated in ModelSim.

Figure 20. *7-Segment display simulation waves*

The adder was tested with 3 different cases: the case where the two inputs added up to 15, the case where the two inputs did not add up to 15, and the case where the input sum exceeded

15. The simulation demonstrated that the module could perform basic addition in binary properly. The first 4 tests were examined for a sum of 15, the second 4 were tested for those that fell below a sum of 15, and the last 3 illustrated what the sum would be when the two inputs exceeded 15. Below are the wave results of the test cases in ModelSim.

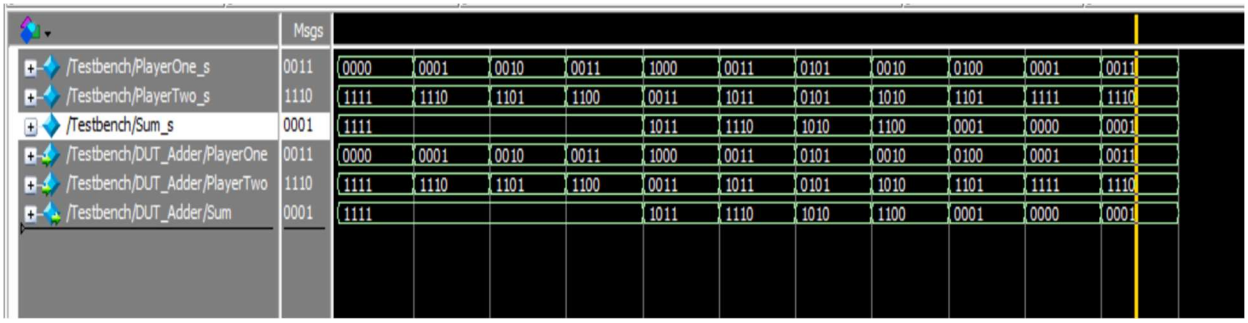

Figure 21. *Adder simulation waves*

For the access controller, 2 cases were considered: The case in which a user enters an incorrect digit in the password, and the case in which a user enters the correct password. Below we demonstrate the waveforms of each case described.

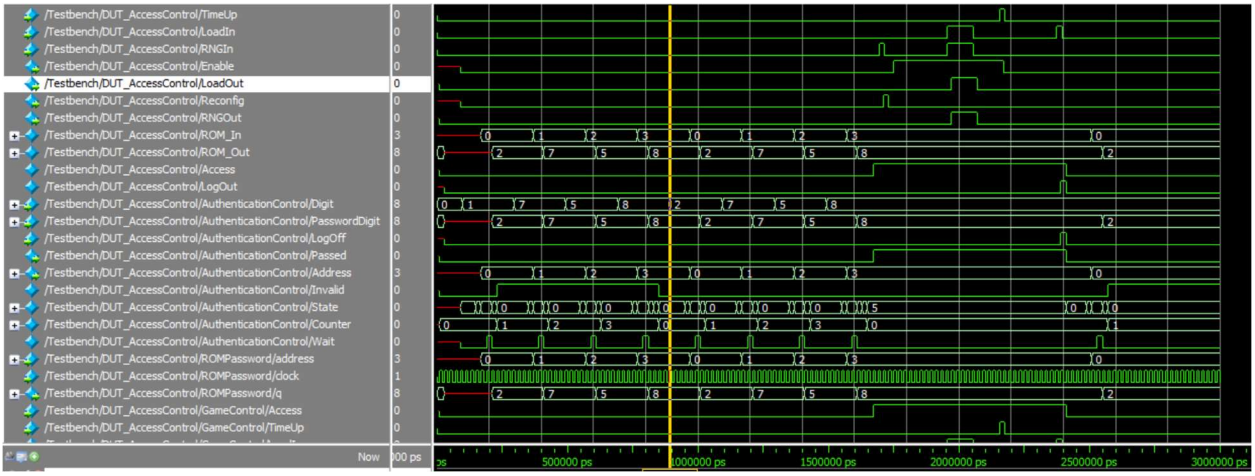

Figure 22. *Updated waveforms for Access Controller module in ModelSim*

As illustrated above, when a user enters the incorrect password, the access controller's LoadOut value remains turned off and set to 0. When a user enters a few digits and realizes they entered the wrong digit, the user can press the reset button and the machine will automatically be switched to the first state in which it checks for the first digit. When a user enters the password correctly, the access controller will output a high reconfiguration signal setting the timer to 99 seconds. If the user wishes to be logged off the game, they can press the load button before starting the game or after the game has finished but not during gameplay. When the button is

pressed one more time, the game should start by setting a high enable signal and setting LoadOut and RNGOut to LoadIn and RNGIn respectively. The machine will stay in state 7, or the gameplay phase, until the access controller receives a low TimeUp signal sent in by the timer. Since the access controller is made up of 3 different module, here are the 3 waveforms of each of the modules:

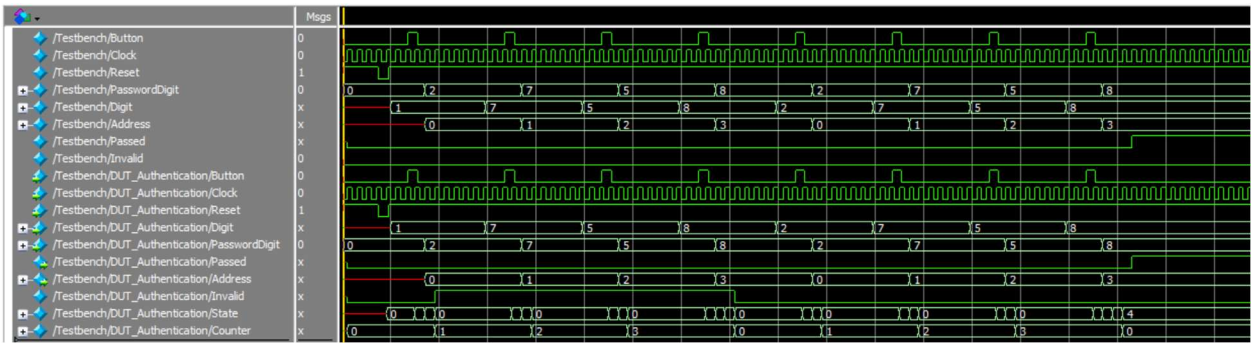

Figure 23. *Waveforms for Authentication module in ModelSim*

For the authentication module one incorrect password was tested and then a correct password to give the user access.

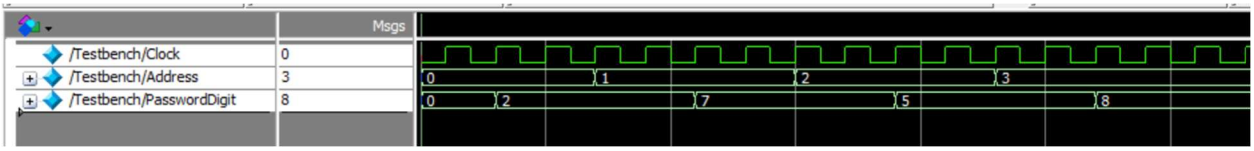

Figure 24. *Waveforms for ROM module in ModelSim*

For the ROM module, the address of each digit was tested. Each digit of the password was expected as output.

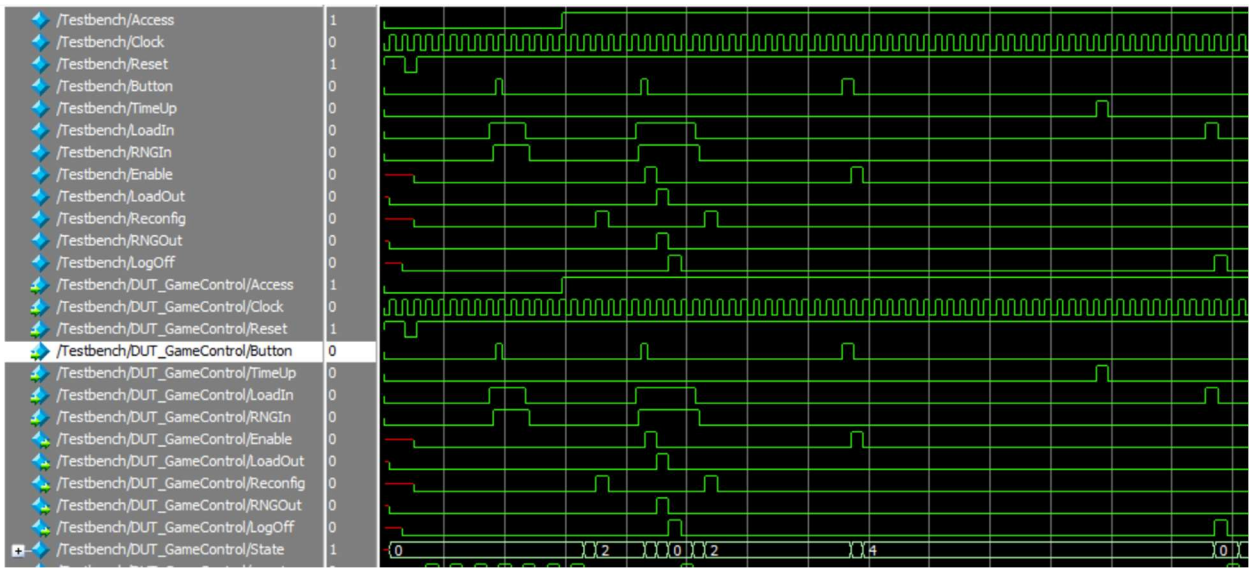

Figure 25. *Waveforms for Game Controller module in ModelSim*

For the Game Controller we check to see if the load and random number generator signal go through before being logged in. Then those signals are checked once logged in. Pressing the load button when the game is over should send a high log off signal, so that was tested at the end.

For the timer, we tested for the counter to count when an enable signal was high and for the counter to continue counting from where it last left off.

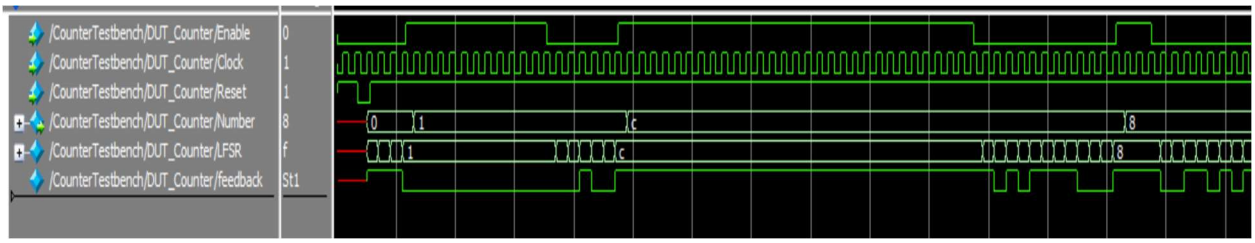

Figure 26. *Simulation Waveforms for LFSR-based counter conditioned to a maximum of 15*

Above is illustrated when the enable signal is high, the counter increases at each rising edge of the clock. When there is a low enable signal, the counter stays the same and continues to count when the enable signal is high again. This counter serves as the base for all other counters used in the timer. All other counters used are adjusted to fit the conditions needed to operate the system they are in.

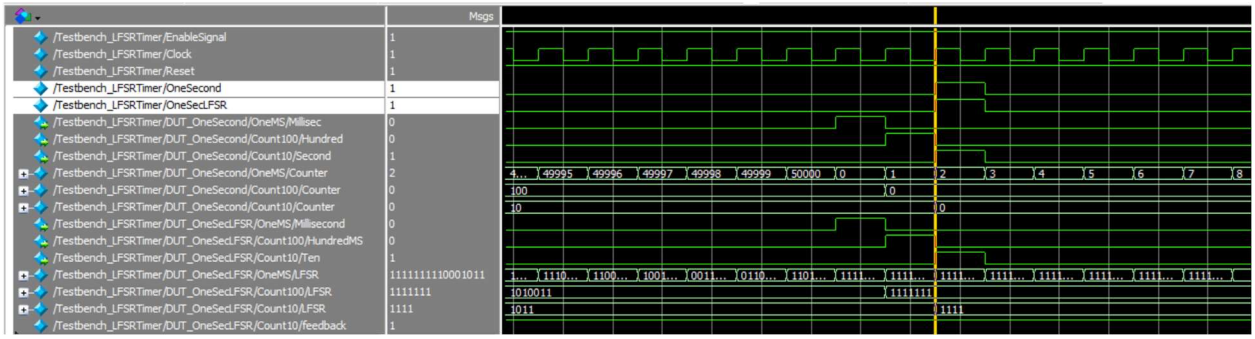

Figure 27. *Waveforms for LFSR-based one second timer zoomed in*

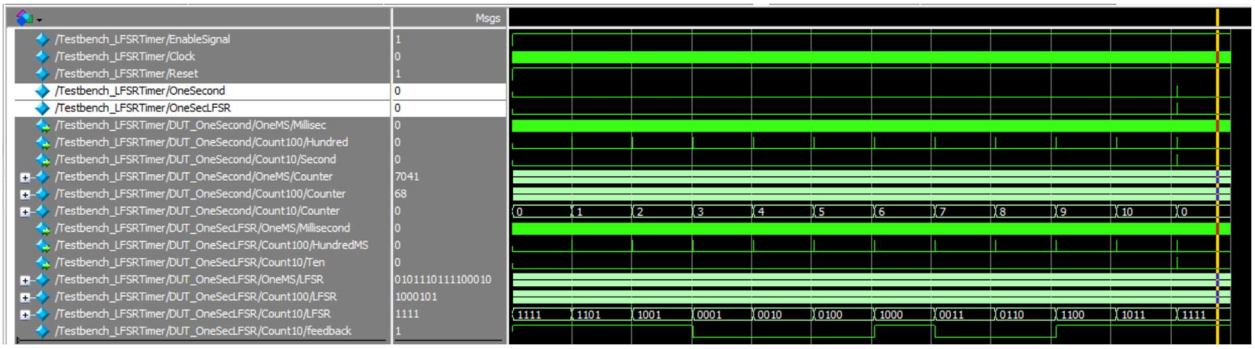

Figure 28. *Waveforms for LFSR-based one second timer*

For the one second timer, the counter-based one second timer was put in a simulation alongside the LFSR-based one second timer to see if both output a high signal after one second has gone by.

For the digit timer, we tested if our digit will be decremented at each decrement signal sent to the digit timer. We also tested for when the digit is 0.

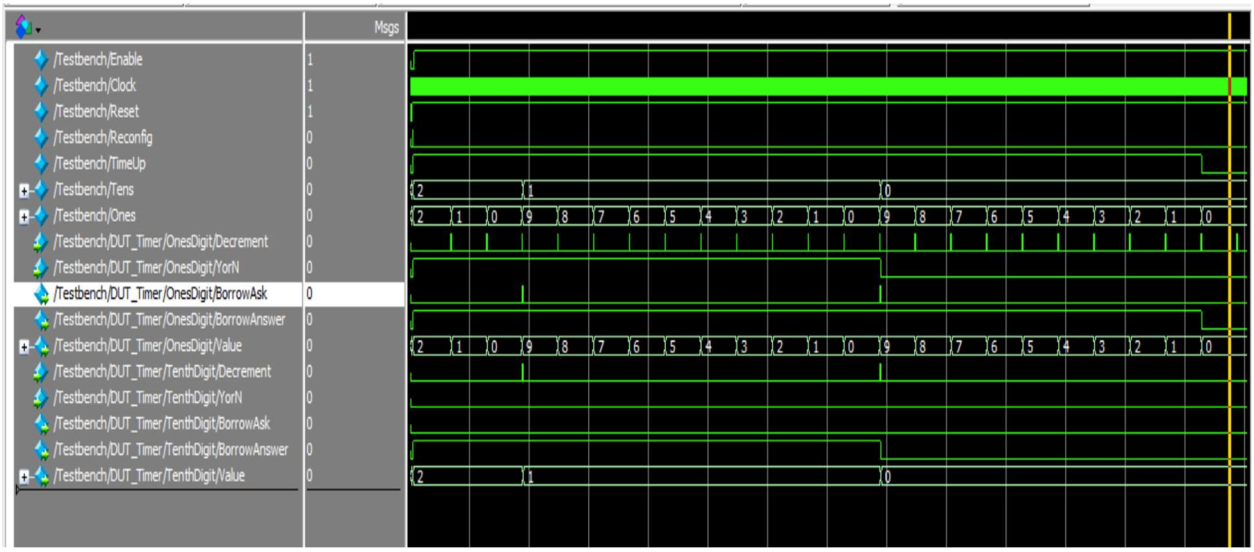

Figure 29. *Simulation Waveforms for Digit Timer in ModelSim*

When the digit is 0, there is a signal that asks another digit if it can borrow, and the digit timer module will receive an answer to that question in the form of a high or low signal. This signal is the YorN. When this value is high, it means that it can borrow from the next digit and the value of this digit will be set to 0. The signal sent to the next digit asking to borrow acts as the decrement signal to that digit and decrements the digit by 1. When all digits are 0, the digit timer modules have an output signal that lets another digit know they cannot borrow. This is the BorrowAnswer. When all digits are 0, the BorrowAnswer on the ones digit will act as the TimeUp signal that lets the Access Controller know the game is over.

The load register is what sends the player value to the 7-segment decoder to display on the board. The load register will only be able to send a value through when the access controller gives access to the player. The load register is tested to yield a value when sent a load signal and when there is no load signal. Below are the results.

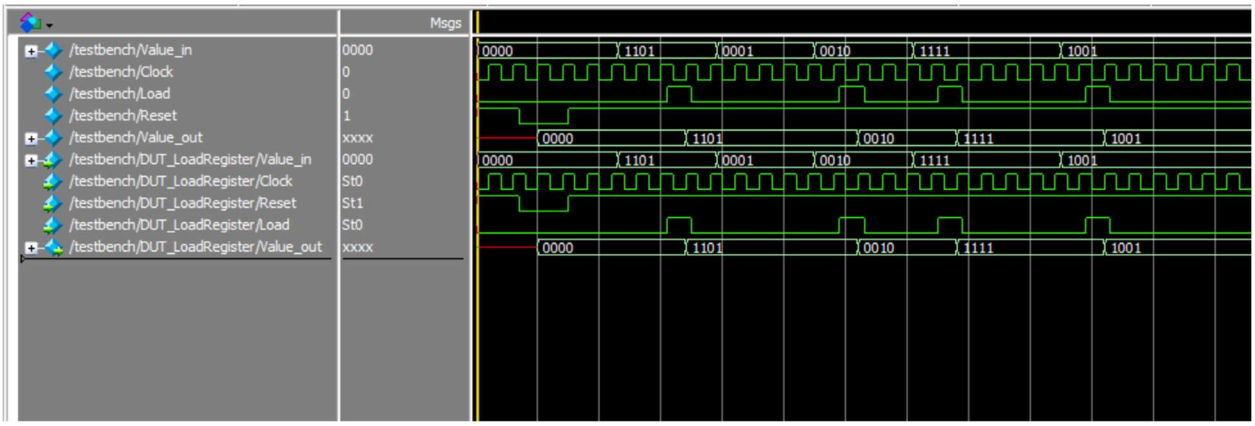

Figure 30. *Load register simulation waves*

As shown, value_out will be set to the value sent in as value_in when a load signal is detected by the rising edge of the clock.

The button shaper is meant to send a single cycle pulse regardless of how long a press of a button is. Below are the simulation results for the button shaper.

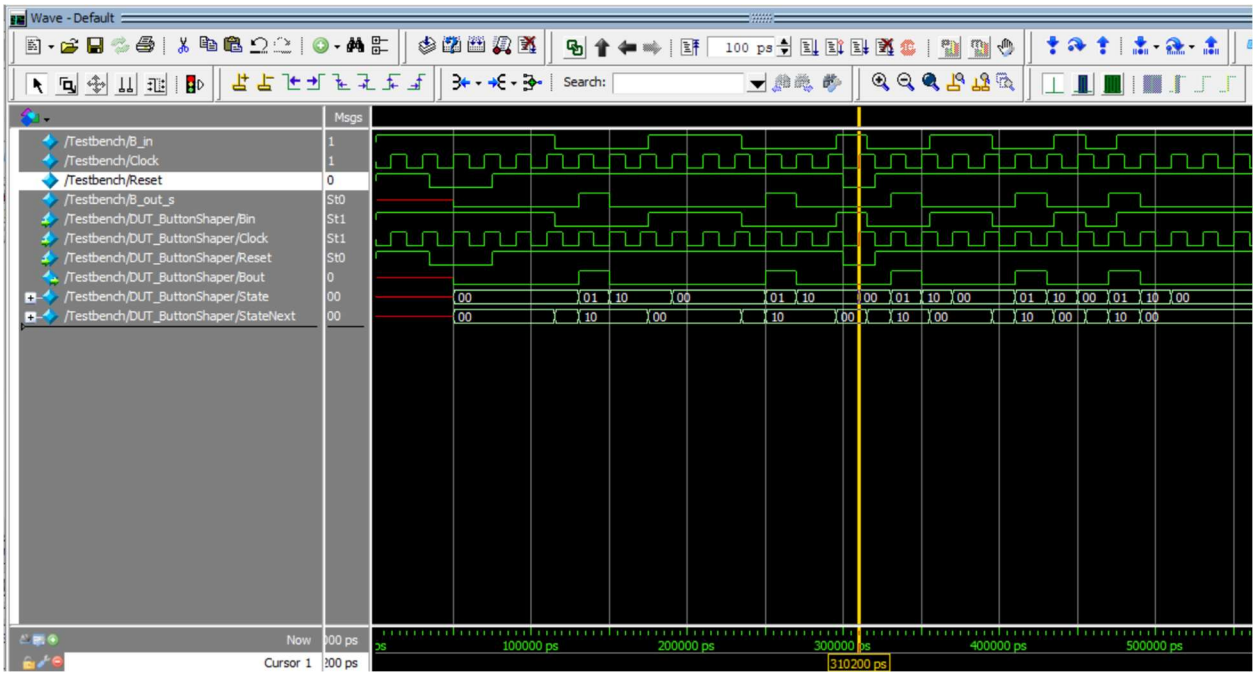

Figure 31. *Simulation waveforms for Button Shaper*

The simulation begins by having reset high and button input high. Each time we have a button press, we should see a button output of high for one clock cycle. This was tested various times and we got what we expected, but we also attempted to have some button inputs one after another to see if the time the button is pressed has a factor in determining the output.

FPGA Board Testing Results

When simulation results were as expected, we uploaded the Verilog code onto the FPGA Board and tested it with a few inputs. Below demonstrates how players can get access to play the game. It includes steps to begin gameplay in which two inputs add up to 15.

It is important to notice that the right-most LEDs indicate when a player has access to play the game. When you first turn on the board, you will notice the 2nd LED from the right turned on. This indicates that the players do not have access to the game.

Figure 32. *2nd LED turned on indicating that users are logged out*

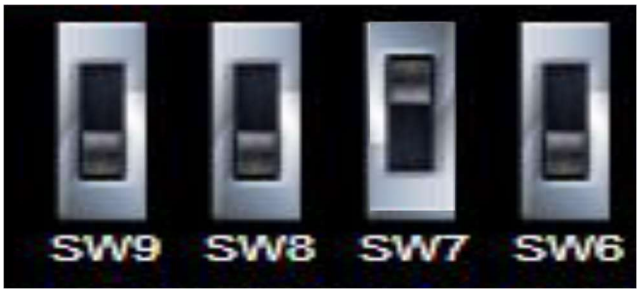

Figure 33. *User flips on bit 1 to represent the hexadecimal value 2*

Step 1: User must enter the password to gain access to the game. The user must start by entering one digit at a time. The user does this by flipping the switches representing the number they need to enter

Figure 34. *User presses the enter button*

Step 2: To enter the digit, the user must press the enter button. The first digit is now read.

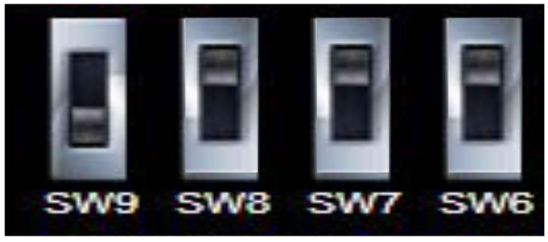

Figure 35. *User flips on bits 0,1, and 2 to represent hexadecimal value 7*

Step 3: User flips switches again for the 2nd digit.

Figure 36. *User presses the enter button*

Step 4: The user presses enter button. The second digit is now read.

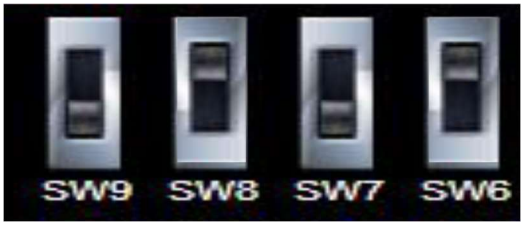

Figure 37. *User flips on bits 0 and 2 to represent hexadecimal value 5*

Step 5: User flips switches again for the 3rd digit.

Figure 38. *User presses the enter button*

Step 6: The user presses enter button. The third digit is now read.

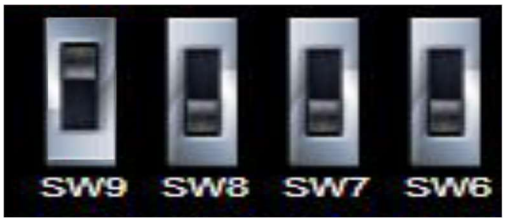

Figure 39. *User flips on bits 3 to represent hexadecimal value 8*

Step 7: User flips switches again for the 4th digit.

Figure 40. *User presses the enter button*

Step 8: The user presses enter button. The fourth digit is now read.

Players will now have access to the game and can input their desired numbers. The right-most LED will now turn on indicating that the players are logged in and have access to play the game as show below.

Figure 41. *LED on the right turns on to indicate users are logged in*

However, to initiate the game, the player must press the button they used for the password 2 more times. One to set the clock and another to begin the game. Below is an example of a player setting the timer and playing the game.

Step 9: The user presses enter button again to set the timer.

Figure 42. *User presses the Game Start button*

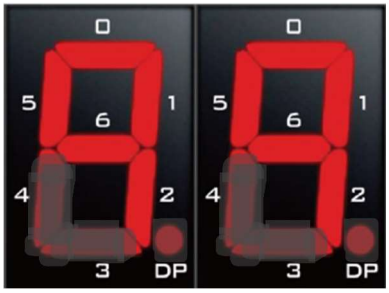

Figure 43. *Timer is set to 99 seconds*

Step 10: The user presses the button once again to begin the game.

Figure 44. *User presses the Game Start button*

The timer begins to count down and the user can now use the 1st button to generate a random number.

Step 11: The user presses the 1st button to generate a random number

Figure 45. *User presses the Random Number Generator button*

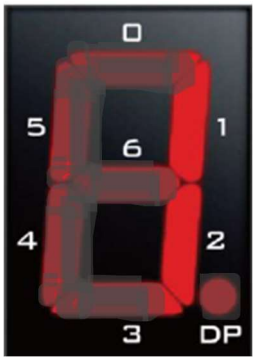

Figure 46. *The random number generated is a value of 1*

Using random number generated, the player must enter the value that will add both numbers to a hexadecimal value F, or 15.

Step 12: The player flips the corresponding to represent the value they want to input

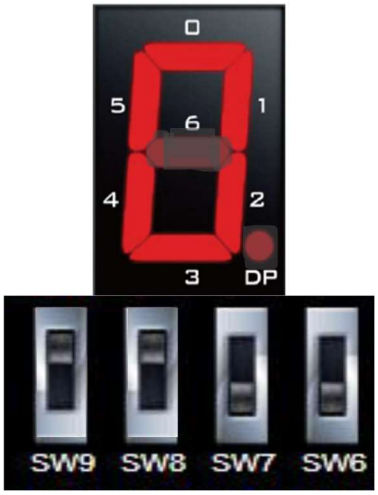

Figure 47. *User flips switches 2 and 3 which represent a hexadecimal value E, or 14*

Step 13: The player loads the value into 7-segment display when pressing their load button.

Figure 48. *User presses the Player Load Button*

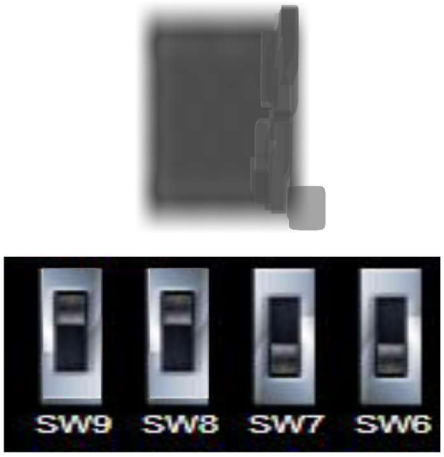

Figure 49. *7-segment display illustrates the loaded value*

Step 14: The display now illuminates the value loaded onto the board

If the player successfully enters a number that sums up both values to 15, the 2nd LED from the left hand side lights up. If a player fails to enter a number that sums up both values to 15, the 1st LED from the left hand side will light up as shown below.

Figure 50. *Player successfully enters value that adds to 15*

Figure 51. *Player fails to enter value that adds to 15*

When the game time ends, the user may log off the device using the load button, the second button from the left.

Figure 52. *User presses the Player Load Button to Log off*

Figure 53. *LED indicated the user is logged out*

Video Demo

This is the video demo explaining the game and going over the new features of the game on the board:

https://drive.google.com/file/d/1sqoWPq0O45FiAcDWb33BtSQaJQu99pvA/view?usp=sharing

Conclusion

In this lab we update the access controller by making separate modules for the authentication and the game control. We also learned how to be more efficient with our finite state machines by implementing high level finite state machines. Without these, separating these module would have been so much more tedious and probably very messy as could've had as much as 18 different states in the authentication module. In updating this module, we developed our versatility and adaptation to change in requirements. We also updated our counters and timers with a more cost-efficient method using what is called linear-feedback shift registers. A way to make this game better is to have the access controller generate a new number each time the player inputs a value whether it is right or wrong. A player will know if they input the correct value by having a score displayed and increase at the press of the load button for the user. The game requires some knowledge on binary digits and how they work, but once acquired, the game becomes a breeze.

Appendix

Appendix A: Top-Level Module Code

```
11  module BinaryMathGame(Player, Password, Reset, Clock, PlayerLoad, RNGButton, GameButton, PlayerDisplay,
12                  RNGDisplay, Result, Match, SignedIn, SignedOut, OnesDisplay, TensDisplay);
13
14      ////////////Switches////////////
15      input[3:0] Player;
16      input[3:0] Password;
17
18      ////////////Buttons////////////
19      input Reset, PlayerLoad, RNGButton, GameButton;
20
21      ////////////Clock////////////
22      input Clock;
23
24      ////////////7-Segment Displays////////////
25      output[6:0] PlayerDisplay;
26      output[6:0] RNGDisplay;
27      output[6:0] Result;
28
29      output[6:0] OnesDisplay;
30      output[6:0] TensDisplay;
31
32      ////////////LEDs////////////
33      output[1:0] Match;
34      output SignedIn, SignedOut;
35
36      ////////////Wires////////////
37      //Adder to Verification and Deocder
38      wire[3:0] AdderSum;
39      //Access Controller to Load Register
40      wire LoadAccess;
```

```
41      //Access Controller to RNG
42      wire RNGAccess;
43
44      //Load Register to Adder and Decoder
45      wire [3:0] PlayerVal;
46      //RNG to Adder and Decoder
47      wire [3:0] RNGVal;
48
49      //Player 1 Load to Access Controller
50      wire PLoadIn;
51      //GameButton to Access Controller
52      wire GameWire;
53
54      //Timer to Access Controller
55      wire TimeUp;
56      //Access Controller to Timer
57      wire EnableSignal;
58      //AccessController to Timer
59      wire Reconfig;
60
61      //Timer Digits
62      wire [3:0] Ones;
63      wire [3:0] Tens;
64
65
66      ////////////////////Buttons////////////////////
67      //GameButton Button
68      ButtonShaper EnterButton(GameButton, Clock, Reset, GameWire);
69      //Player Load Button
```

```
70      ButtonShaper LoadButton1(PlayerLoad, Clock, Reset, PLoadIn);
71
72      //Access Controller
73      AccessControl AccessController(GameWire, Password, Clock, Reset, TimeUp, EnableSignal, Reconfig, PLoadIn,
74          LoadAccess, RNGButton, RNGAccess, SignedIn, SignedOut);
75
76      //Random Number Generator Decoder
77      RNG_LFSR RandomNumGenerator(RNGAccess, Clock, Reset, RNGVal);
78      SevenSegDecoder RNGDecoder(RNGVal, RNGDisplay);
79
80      //Adder
81      Adder AdderModule(PlayerVal, RNGVal, AdderSum);
82      Verification Verification1(AdderSum, Match);
83
84      //Player Decoder
85      LoadRegister PlayerOneLoad(Player, PlayerVal, Clock, Reset, LoadAccess);
86      SevenSegDecoder PlayerDecoder(PlayerVal, PlayerDisplay);
87
88      //Adder Decoder
89      SevenSegDecoder AdderResult(AdderSum, Result);
90
91      ////////////////////////////////TIMER/////////////////////////////////////////
92      Timer CountdownTimer(EnableSignal, Reconfig, Clock, Reset, Tens, Ones, TimeUp);
93
94      //Seven Segment Decoder for Timer
95      SevenSegDecoder OnesDigitDisplay(Ones, OnesDisplay);
96      SevenSegDecoder TensDigitDisplay(Tens, TensDisplay);
97
98  endmodule
```

Figure 54. *Verilog Code for BinaryMathGame Module*

Appendix B: Seven Segment Decoder Module Code

```
Ln#
 1  module SevenSegDecoder (Decoder_in, Decoder_out);
 2
 3      input [3:0]Decoder_in;
 4      output [6:0]Decoder_out;
 5      reg [6:0]Decoder_out;
 6
 7      always @ (Decoder_in)
 8          begin
 9              case(Decoder_in)
10                  4'b0000: begin Decoder_out = 7'b1000000; end
11                  4'b0001: begin Decoder_out = 7'b1111001; end
12                  4'b0010: begin Decoder_out = 7'b0100100; end
13                  4'b0011: begin Decoder_out = 7'b0110000; end
14
15                  4'b0100: begin Decoder_out = 7'b1011001; end
16                  4'b0101: begin Decoder_out = 7'b0010010; end
17                  4'b0110: begin Decoder_out = 7'b0000010; end
18                  4'b0111: begin Decoder_out = 7'b1111000; end
19
20                  4'b1000: begin Decoder_out = 7'b000_0000; end
21                  4'b1001: begin Decoder_out = 7'b001_1000; end
22                  4'b1010: begin Decoder_out = 7'b000_1000; end
23                  4'b1011: begin Decoder_out = 7'b000_0011; end
24
25                  4'b1100: begin Decoder_out = 7'b100_0011; end
26                  4'b1101: begin Decoder_out = 7'b010_0001; end
27                  4'b1110: begin Decoder_out = 7'b000_0110; end
28                  4'b1111: begin Decoder_out = 7'b000_1110; end
29
30                  default: begin Decoder_out = 7'b100_0000; end
31              endcase
32          end
33  endmodule
34
```

Figure 55. *Verilog Code for 7-Segment Decoder Module*

Appendix C: Adder Module Code

C:\Users\rogel\Documents\Spring 2023\ECE 5440\Lab1_Lopez_Rogelio\sim_adder\Adder.v - Default

```
Ln#
 1  module Adder(PlayerOne, PlayerTwo, Sum);
 2      input[3:0] PlayerOne;
 3      input[3:0] PlayerTwo;
 4      output[3:0] Sum;
 5      reg[3:0] Sum;
 6
 7      always @ (PlayerOne, PlayerTwo)
 8          begin
 9              Sum = PlayerOne + PlayerTwo;
10          end
11
12  endmodule
```

Figure 56. *Verilog Code for Adder Module*

Appendix D: Verification Module Code

C:/Users/rogel/Documents/Spring 2023/ECE 5440/Lab1_Lopez_Rogelio/sim_verif/Verification.v - Default

```
Ln#
 1  module Verification(Sum, LEDs);
 2          input[3:0] Sum;
 3          output[1:0] LEDs;
 4          reg[1:0] LEDs;
 5
 6          always @(Sum)
 7          begin
 8                  if(Sum < 15)
 9                          LEDs = 2'b01;
10                  else
11                          LEDs = 2'b10;
12          end
13  endmodule
```

Figure 57. *Verilog Code for Verification Module*

Appendix E: Load Register Module Code

```
1   module LoadRegister(Value_in, Value_out, Clock, Reset, Load);
2      input [3:0] Value_in;
3      input Clock, Reset;
4      input Load;
5      output [3:0] Value_out;
6
7
8      reg [3:0] Value_out;
9
10     always@(posedge Clock)
11        begin
12           if(Reset == 1'b0)
13              begin
14                    Value_out <= 4'b0000;
15              end
16           else
17              begin
18                    if(Load == 1'b1)
19                       begin
20                             Value_out <= Value_in;
21                       end
22              end
23        end
24  endmodule
25
```

Figure 58. *Verilog Code for Load Register Module*

Appendix F: Button Shaper Module Code

```
1  module ButtonShaper(Bin, Clock, Reset, Bout);
2          input Bin;
3          input Clock, Reset;
4          output Bout;
5          reg Bout;
6
7          parameter INIT = 0, PULSE = 1, WAIT = 2;
8          reg [1:0] State, StateNext;
9
10         always @(State, Bin)
11            begin
12                 case(State)
13                 INIT:
14                 begin
15                         Bout = 1'b0;
16                         if(Bin == 1'b0)
17                                 StateNext = PULSE;
18                         else
19                                 StateNext = INIT;
20                 end
21                 PULSE:
22                 begin
23                         Bout = 1'b1;
24                         StateNext = WAIT;
25                 end
26                 WAIT:
27                 begin
28                         Bout = 1'b0;
29                         if(Bin == 1'b1)
30                                 StateNext = INIT;
31                         else
32                                 StateNext = WAIT;
33                 end
34                 endcase
35            end
36
37         always @(posedge Clock)
38            begin
39                 if(Reset == 1'b0)
40                         State <= INIT;
41                 else
42                         State <= StateNext;
43            end
44
45 endmodule
46
47
```

Figure 59. *Verilog Code for Button Shaper Module*

Appendix G: Access Controller Module Code

```
module AccessController(Button, Value, Clock, Reset, TimeUp, Enable, Reconfig, LoadIn, LoadOut, RNGIn, RNGOut, LoggedIn, LoggedOut);
    input Button;
    input Clock, Reset;
    input [4:0] Value;
    input TimeUp;
    input LoadIn;
    input RNGIn;

    output Enable;
    output LoadOut;
    output Reconfig;
    output RNGOut;
    output LoggedIn, LoggedOut;


    //Wire for Authentication to ROM
    wire [4:0] ROM_In;
    //Wire for ROM to Authentication
    wire [3:0] ROM_Out;

    //Wire for Authentication to GamerControl
    wire Access;

    //Wire for GamerControl to Authentication
    wire LogOut;

    Authentication AuthenticationControl(Clock, Reset, LogOut, ROM_Out, Button, Value, ROM_In, Access, LoggedIn, LoggedOut);
    ROM ROMPassword(ROM_In, Clock, ROM_Out);
    GameController GameControl(Clock, Reset, Access, Button, TimeUp, Enable, Reconfig, LoadIn, LoadOut, RNGIn, RNGOut, LogOut);

endmodule
```

Figure 60. *Verilog Code for Access Controller Module*

Appendix H: Counter Module Code

```
module CounterLFSR(Enable, Clock, Reset, Number);
    input Enable;
    input Clock, Reset;

    output [3:0] Number;

    reg [3:0] Number;
    reg [3:0] LFSR;
    wire feedback = LFSR[3];

    always@(posedge Clock)
        begin
            if(Reset == 1'b0)
                begin
                    Number <= 4'b0000;
                    LFSR <= 4'b1111;
                end
            else if(Enable == 1'b1)
                begin
                    Number <= LFSR;
                end
            else
                begin
                    if(LFSR == 4'b1110)
                        begin
                            LFSR <= 4'b1111;
                        end
                    else
                        begin
                            LFSR[0] <= feedback;
                            LFSR[1] <= LFSR[0] ^ feedback;
                            LFSR[2] <= LFSR[1];
                            LFSR[3] <= LFSR[2];
                        end
                end
        end

endmodule
```

Figure 61. *General code for a LFSR-based counter module*

Appendix I: One Second Timer Top Module

```
module OneSecondLFSR(EnableSignal, Clock, Reset, OneSecond);

    input EnableSignal;
    input Clock, Reset;
    output OneSecond;

    //Wire for OneMillisecond to CountTo100
    wire OneMSWire;
    //Wire for CountTo10
    wire OneHundredMSWire;

    OneMillisecondLFSR OneMS(EnableSignal, Clock, Reset, OneMSWire);

    HundredMSLFSR Count100(OneMSWire, Clock, Reset, OneHundredMSWire);

    CountTo10LFSR Count10(OneHundredMSWire, Clock, Reset, OneSecond);

endmodule

```

Figure 62. *Code for a once second timer with multiple counters*

Appendix J: One Millisecond Counter Module

```
module OneMillisecondLFSR(EnableSignal, Clock, Reset, Millisecond);

    input EnableSignal;
    input Clock, Reset;
    output Millisecond;

    reg Millisecond;
    reg [15:0] LFSR;
    wire feedback = LFSR[15];

    always @ (posedge Clock)
        begin
            if(Reset == 1'b0)
                begin
                    LFSR <= 16'b1111_1111_1111_1111;
                end
            else if(EnableSignal == 1'b1)
                begin
                    if(LFSR == 16'b1101_1011_0110_1100)
                        begin
                            LFSR <= 16'b1111_1111_1111_1111;
                             <= 1'b1;
                        end
                    else
                        begin
                            LFSR[0] <= feedback;
                            LFSR[1] <= LFSR[0];
                            LFSR[2] <= LFSR[1] ^ feedback;
                            LFSR[3] <= LFSR[2] ^ feedback;
                            LFSR[4] <= LFSR[3];
                            LFSR[5] <= LFSR[4] ^ feedback;
```

```
32                              LFSR[6] <= LFSR[5];
33                              LFSR[7] <= LFSR[6];
34                              LFSR[8] <= LFSR[7];
35                              LFSR[9] <= LFSR[8];
36                              LFSR[10] <= LFSR[9];
37                              LFSR[11] <= LFSR[10];
38                              LFSR[12] <= LFSR[11];
39                              LFSR[13] <= LFSR[12];
40                              LFSR[14] <= LFSR[13];
41                              LFSR[15] <= LFSR[14];
42
43                              Millisecond <= 1'b0;
44                          end
45                  end
46          end
47
48  endmodule
```

Figure 63. *Code for LFSR-based one millisecond timer*

Appendix K: CountTo100 Module

```
1   module HundredMSLFSR(EnableSignal, Clock, Reset, HundredMS);
2
3       input EnableSignal;
4       input Clock, Reset;
5       output HundredMS;
6
7       reg HundredMS;
8       reg [6:0] LFSR;
9       wire feedback = LFSR[6];
10
11      always @ (posedge Clock)
12          begin
13              if(Reset == 1'b0)
14                  begin
15                      LFSR <= 7'b111_1111;
16                      HundredMS <= 1'b0;
17                  end
18              else if(EnableSignal == 1'b1)
19                  begin
20                      if(LFSR == 7'b110_1000)
21                          begin
22                              LFSR <= 7'b111_1111;
23                              HundredMS <= 1'b1;
24                          end
25                      else
26                          begin
27                              LFSR[0] <= feedback;
28                              LFSR[1] <= LFSR[0] ^ feedback;
29                              LFSR[2] <= LFSR[1];
30                              LFSR[3] <= LFSR[2];
31                              LFSR[4] <= LFSR[3];
32                              LFSR[5] <= LFSR[4];
33                              LFSR[6] <= LFSR[5];
34
35                              HundredMS <= 1'b0;
36                          end
37                  end
38              else
39                  begin
40                      HundredMS <= 1'b0;
41                  end
42          end
43
44  endmodule
```

Figure 64. *Code for an LFSR-based counter with modified conditions counting to 100*

Appendix L: CountTo10 Module

```
module CountTo10LFSR(EnableSignal, Clock, Reset, Ten);
    input EnableSignal;
    input Clock, Reset;
    output Ten;

    reg Ten;
    reg [3:0] LFSR;
    wire feedback = LFSR[3];

    always @ (posedge Clock)
        begin
            if(Reset == 1'b0)
                begin
                    LFSR <= 4'b1111;
                    Ten <= 1'b0;
                end
            else if(EnableSignal == 1'b1)
                begin
                    if(LFSR == 4'b1100)
                        begin
                            LFSR <= 4'b1111;
                            Ten <= 1'b1;
                        end
                    else
                        begin
                            LFSR[0] <= feedback;
                            LFSR[1] <= LFSR[0] ^ feedback;
                            LFSR[2] <= LFSR[1];
                            LFSR[3] <= LFSR[2];

                            Ten <= 1'b0;
                        end
                end
            else
                begin
                    Ten <= 1'b0;
                end
        end

endmodule
```

Figure 65. *Code for a LFSR-based counter with modified conditions counting to 10*

Appendix M: Digit Timer Module

```
module DigitTimer(Decrement, YorN, Reconfig, Clock, Reset, BorrowAsk, BorrowAnswer, Value);
    input Decrement;
    input YorN;
    input Clock, Reset;
    input Reconfig;
    output BorrowAsk;
    output BorrowAnswer;
    output [3:0] Value;

    reg [3:0] Value;
    reg BorrowAsk;
    reg BorrowAnswer;

    always@(posedge Clock)
        begin
            if(Reset == 1'b0)
                begin
                    Value = 4'b0000;
                    BorrowAsk = 0;
                    BorrowAnswer = 0;
                end
            else if(Reconfig == 1'b1)
                begin
                    Value <= 4'b1001;
                    BorrowAnswer <= 1'b1;
                    BorrowAsk <= 1'b0;
                end
            else
                begin
                    //If we get a signal to decrement the value
                    if(Decrement == 1'b1)
                        begin
                            //If our value is 0, check to see if we can borrow from next digit
                            if(Value == 4'b0000)
                                begin
                                    //If we can't borrow
                                    if(YorN == 1'b0)
                                        begin
                                            BorrowAsk <= 1'b0;
                                            BorrowAnswer <= 1'b0;
                                        end
                                    //If we can borrow
                                    else
                                        begin
                                            //Ask to Borrow
                                            BorrowAsk <= 1'b1;
                                            Value <= 4'b1001;
                                            BorrowAnswer <= 1'b1;
                                        end
                                end
                            //If our value is greater than 0, subtract 1
                            else
                                begin
                                    Value <= Value - 1;
                                    //If our value is 1 before subtraction and we cannot
                                    //borrow from the next digit, this digit cannot let
                                    //another digit borrow
                                    if(Value == 4'b0001 & YorN == 1'b0)
                                        begin
                                            BorrowAnswer <= 1'b0;
                                        end
                                    else
```

```
                                        begin
                                            BorrowAnswer <= 1'b1;
                                        end
                                    end
                                end
                            else
                                begin
                                    BorrowAsk <= 0;
                                end
                        end
                end

endmodule
```

Figure 66. *Code for the digit timers*

Appendix N: Timer Module

```
module Timer(Enable, Reconfig, Clock, Reset, Tens, Ones, TimeUp);
    input Enable;
    input Reconfig;
    input Clock, Reset;
    output[3:0] Tens, Ones;
    output TimeUp;

    //Wire for OneSecondTimer to DigitTimer (OnesDigit)
    wire OneSecond;
    //Wire for Ones Digit to ask to borrow
    wire BorrowFromTens;
    //Wire for Tens to answer Ones for borrow
    wire YesOrNo;
    //Wire for PlaceHolder
    wire PlaceHolder;

    OneSecondLFSR OneSecondSignal(Enable, Clock, Reset, OneSecond);

    //Digit Timer for the Ones
    DigitTimer OnesDigit(OneSecond, YesOrNo, Reconfig, Clock, Reset, BorrowFromTens, TimeUp, Ones);
    //Digit Timer for the Tens
    DigitTimer TenthDigit(BorrowFromTens, 1'b0, Reconfig, Clock, Reset, PlaceHolder, YesOrNo, Tens);

endmodule
```

Figure 67. *Top Level Code for the 99 second timer module*

Appendix O: Random Number Generator Module

```
module RNG(RNG_signal, Clock, Reset, RandomNumber);
    input RNG_signal;
    input Clock, Reset;

    output [3:0] RandomNumber;

    Counter CounterRNG(~RNG_signal, Clock, Reset, RandomNumber);

endmodule
```

Figure 68. *Top Level Code for Random Number Generator*

Appendix P: Random Number Generator Counter Module

```
1   module CounterLFSR(Enable, Clock, Reset, Number);
2       input Enable;
3       input Clock, Reset;
4
5       output [3:0] Number;
6
7       reg [3:0] Number;
8       reg [3:0] LFSR;
9       wire feedback = LFSR[3];
10
11      always@(posedge Clock)
12          begin
13              if(Reset == 1'b0)
14                  begin
15                      Number <= 4'b0000;
16                      LFSR <= 4'b0000;
17                  end
18              else if(Enable == 1'b1)
19                  begin
20                      Number <= LFSR;
21                  end
22              else
23                  begin
24                      if(LFSR == 4'b0001)
25                          begin
26                              LFSR <= 4'b0000;
27                          end
28                      else
29                          begin
30                              LFSR[0] <= feedback;
31                              LFSR[1] <= LFSR[0] ~^ feedback;
32                              LFSR[2] <= LFSR[1];
33                              LFSR[3] <= LFSR[2];
34                          end
35                  end
36          end
37
38  endmodule
```

Figure 69. *Code for Random Number Generator Counter Module*

Appendix P: Authentication Module

```
7  module Authentication(Clock, Reset, LogOff, PasswordDigit, Button, Digit, Address, Passed, LoggedIn, LoggedOut);
8      input Button;
9      input Clock, Reset;
10     input [3:0] Digit;
11     input [3:0] PasswordDigit;
12     input LogOff;
13
14     output Passed;
15     output [4:0] Address;
16     output LoggedIn;
17     output LoggedOut;
18
19     reg Passed;
20     reg [4:0] Address;
21     reg LoggedIn;
22     reg LoggedOut;
23     reg Invalid;
24
25     parameter DIGIT = 0, ADDRESS = 1, RECIEVE = 2, COMPARE = 3, VERIFY = 4, PASSED = 5;
26     reg [2:0] State;
27     reg [4:0] Counter;
28     reg Wait;
29
30     always @ (posedge Clock)
31         begin
32             if(Reset == 1'b0 | LogOff == 1'b1)
33                 begin
34                     Passed <= 1'b0;
35                     Invalid <= 1'b0;
36                     Counter <= 5'b00000;
37                     State <= DIGIT;
38                     Wait <= 1'b0;
39                     LoggedIn <= 1'b0;
40                     LoggedOut <= 1'b1;
41                 end
42             else
43                 begin
44                     case (State)
45                     DIGIT:
46                         begin
47                             LoggedIn <= 1'b0;
48                             LoggedOut <= 1'b1;
49                             Passed <= 1'b0;
50                             if(Button == 1'b1)
51                                 begin
52                                     State <= ADDRESS;
53                                 end
54                             else
55                                 begin
56                                     State <= DIGIT;
57                                 end
58                         end
59                     ADDRESS:
60                         begin
61                             Address <= Counter;
62                             State <= RECIEVE;
63                         end
64                     RECIEVE:
65                         begin
66                             if(Wait == 1'b1)
67                                 begin
68                                     Wait <= 1'b0;
```

```
                                State <= COMPARE;
                            end
                        else
                            begin
                                Wait <= Wait + 1;
                            end
                    end
                COMPARE:
                    begin
                        if(Digit == PasswordDigit)
                            begin

                            end
                        else
                            begin
                                Invalid <= 1'b1;
                            end

                        if(Counter == 5'b00011)
                            begin
                                State <= VERIFY;
                            end
                        else
                            begin
                                Counter <= Counter + 1;
                                State <= DIGIT;
                            end
                    end
                VERIFY:
                    begin
                        if(Invalid == 1'b1)
```

```
                            begin
                                State <= DIGIT;
                            end
                        else
                            begin
                                State <= PASSED;
                            end

                        Counter <= 5'b00000;
                        Invalid <= 1'b0;
                    end
                PASSED:
                    begin
                        LoggedIn <= 1'b1;
                        LoggedOut <= 1'b0;
                        Passed <= 1'b1;
                    end
                default:
                    begin
                        Passed <= 1'b0;
                        Invalid <= 1'b0;
                        Counter <= 5'b00000;
                        LoggedIn <= 1'b0;
                        LoggedOut <= 1'b1;
                    end
                endcase
            end
        end
endmodule
```

Figure 70. *Code for Authentication Module*

Appendix Q: Game Controller Module

```
module GameController(Clock, Reset, Access, Button, TimeUp, Enable, Reconfig, LoadIn, LoadOut, RNGIn, RNGOut, LogOff);
    input Access;
    input Clock, Reset;
    input Button;
    input TimeUp;
    input LoadIn;
    input RNGIn;

    output Enable;
    output LoadOut;
    output Reconfig;
    output RNGOut;
    output LogOff;

    reg Enable;
    reg LoadOut;
    reg Reconfig;
    reg RNGOut;
    reg LogOff;

    parameter WAIT = 0, PASSED = 1, START = 2, GAMEPLAY = 3, GAMEOVER = 4;
    reg [2:0] State;

    always @ (posedge Clock)
        begin
            if(Reset == 1'b0)
                begin
                    Enable <= 1'b0;
                    LoadOut <= 1'b0;
                    Reconfig <= 1'b0;
                    RNGOut <= 1'b0;
                    LogOff <= 1'b0;
                end
            else
                begin
                    case (State)
                    WAIT:
                        begin
                            LogOff <= 1'b0;
                            if(Access == 1'b1)
                                begin
                                    State <= PASSED;
                                end
                            else
                                begin
                                    State <= WAIT;
                                end
                        end
                    PASSED:
                        begin
                            Reconfig <= 1'b1;
                            State <= START;
                        end
                    START:
                        begin
                            Reconfig <= 1'b0;
                            if(Button == 1'b1)
                                begin
                                    Enable <= 1'b1;
                                    State <= GAMEPLAY;
                                end
                            else if(LoadIn == 1'b1)
```

```
                                begin
                                    LogOff <= 1'b1;  //Signal sent to the access controller to turn access off
                                    State <= WAIT;
                                end
                            else
                                begin
                                    State <= START;
                                end
                        end
                    GAMEPLAY:
                        begin
                            //Give Access
                            LoadOut <= LoadIn;
                            RNGOut <= RNGIn;

                            if(TimeUp == 1'b0)
                                begin
                                    Enable <= 1'b0;
                                    State <= GAMEOVER;
                                end
                            else
                                begin
                                    State <= GAMEPLAY;
                                end
                        end
                    GAMEOVER:
                        begin
                            LoadOut <= 1'b0;
                            RNGOut <= 1'b0;
                            if(Button == 1'b1)
                                begin
                                    State <= PASSED;
                                end
                            else
                                begin
                                    State <= GAMEOVER;
                                end

                        end
                    default:
                        begin
                            LoadOut <= 1'b0;
                            RNGOut <= 1'b0;
                            State <= WAIT;
                        end
                    endcase
                end
        end

endmodule
```

Figure 71. *Code for Game Controller Module*

Appendix R: ROM Module

```
38  `timescale 1 ps / 1 ps
39  // synopsys translate_on
40  module ROM (
41      address,
42      clock,
43      q);
44
45      input   [4:0]  address;
46      input     clock;
47      output  [3:0]  q;
48  `ifndef ALTERA_RESERVED_QIS
49  // synopsys translate_off
50  `endif
51      tri1      clock;
52  `ifndef ALTERA_RESERVED_QIS
53  // synopsys translate_on
54  `endif
55
56      wire [3:0] sub_wire0;
57      wire [3:0] q = sub_wire0[3:0];
58
59      altsyncram  altsyncram_component (
60                  .address_a (address),
61                  .clock0 (clock),
62                  .q_a (sub_wire0),
63                  .aclr0 (1'b0),
64                  .aclr1 (1'b0),
65                  .address_b (1'b1),
66                  .addressstall_a (1'b0),
67                  .addressstall_b (1'b0),
68                  .byteena_a (1'b1),
```

```
                .byteena_b (1'b1),
                .clock1 (1'b1),
                .clocken0 (1'b1),
                .clocken1 (1'b1),
                .clocken2 (1'b1),
                .clocken3 (1'b1),
                .data_a ({4{1'b1}}),
                .data_b (1'b1),
                .eccstatus (),
                .q_b (),
                .rden_a (1'b1),
                .rden_b (1'b1),
                .wren_a (1'b0),
                .wren_b (1'b0));
    defparam
        altsyncram_component.address_aclr_a = "NONE",
        altsyncram_component.clock_enable_input_a = "BYPASS",
        altsyncram_component.clock_enable_output_a = "BYPASS",
`ifdef NO_PLI
        altsyncram_component.init_file = "ROM.rif"
`else
        altsyncram_component.init_file = "ROM.hex"
`endif
,
        altsyncram_component.intended_device_family = "Cyclone V",
        altsyncram_component.lpm_hint = "ENABLE_RUNTIME_MOD=NO",
        altsyncram_component.lpm_type = "altsyncram",
        altsyncram_component.numwords_a = 32,
        altsyncram_component.operation_mode = "ROM",
        altsyncram_component.outdata_aclr_a = "NONE",
        altsyncram_component.outdata_reg_a = "CLOCK0",
        altsyncram_component.widthad_a = 5,
        altsyncram_component.width_a = 4,
        altsyncram_component.width_byteena_a = 1;


endmodule
```

Figure 72. *Code for ROM Module*