ECE6370: Advanced Digital Design

Lab 4: FPGA-based Mental Binary Math Game

Rahul Verma
PS: 2251462

Contents

1.0. Introduction

This is a single-player mental binary math game implemented on Cyclone V- FPGA. For the game to start, the player should enter a 4-digit password. Until the password is authenticated the player cannot start entering the number. Log in status is indicated by a pair of LEDs as shown in the figure. After the password is authenticated, the timer will show 99 until the player hits “Game Start” button. The “Game Start” button will start the 99 sec timer, then the player has to hit the “Random Number” push-button to generate a random number which will be displayed on the second 7-segment display from the right. Then the Player has to compute a number in his mind by looking at the random number and enter the computed number which will add up to 1111. The Player’s number will be displayed on the right most 7-segment display the player presses the load button. The sum of these numbers will be displayed on the fourth 7-segment display from left. Two more LEDs on the left will indicate whether the sum is equal to 1111 or not. If the sum is 1111, matching LED will glow, otherwise non-matching will glow. The player can go as many rounds as he wants until the timer runs out after which the number of correct attempts will be displayed on the middle two 7-segment LEDs. The player can play multiple games as long as he is logged-in by pressing the “Game Restart” button. Player also has the option of resetting the password by pressing the password reset button before the game session starts. When the button is pressed, the game goes into password reset mode and will accept four 4-bits password by pressing password entering button after every digit. To logout, the player can press the logout button to logout when the game session hasn’t started.

The interface of the gaming system is shown in figure 1.0.1.

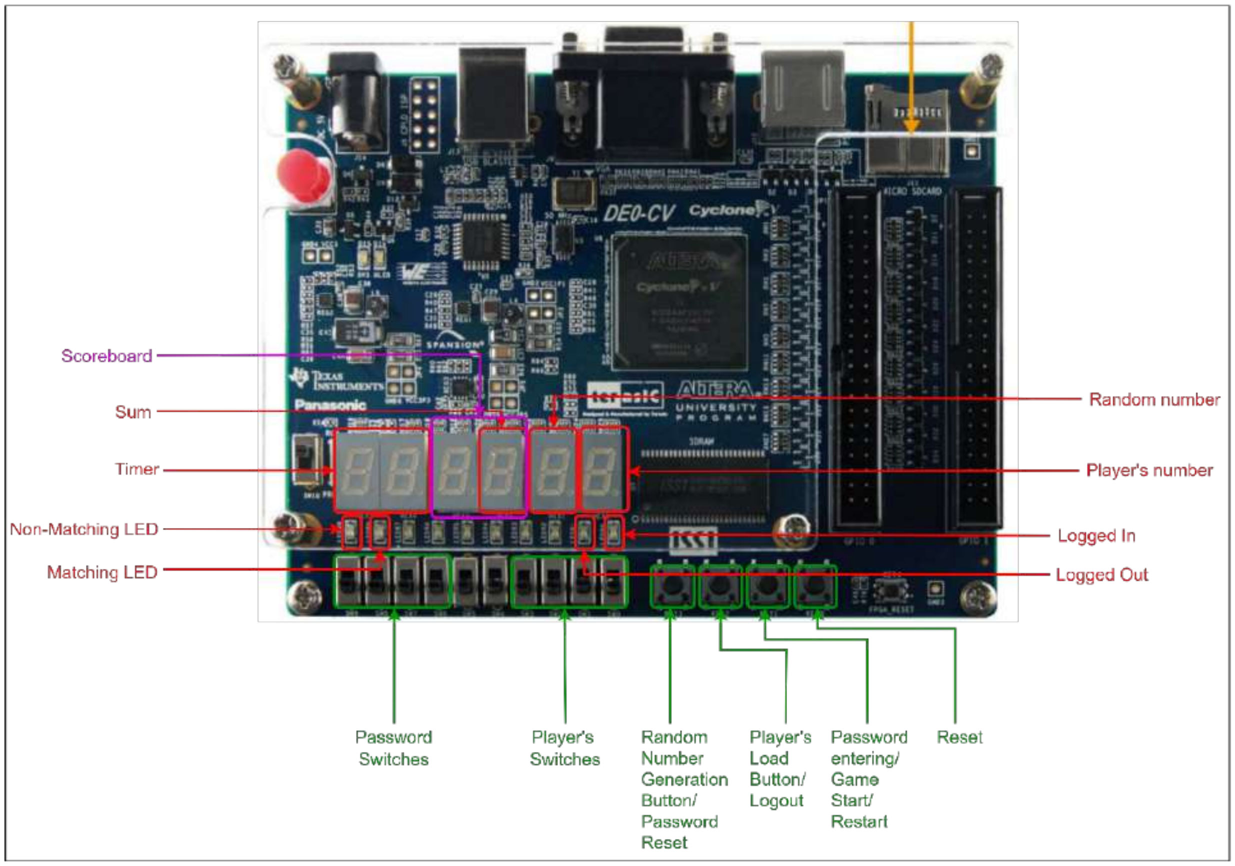


Figure 1.0.1: FPGA user interface

2.0. System Architecture

The system architecture of the design is shown in the figure 2.0.1. It consists of ten types of submodules- decoder_7seg module, adder module, verification module, buttonShaper module, loadRegister module, accessController module, rng module, two-digit timer module, scoreboard module, and mux_2_1 module. Top level module has seven inputs which includes clk and rst. The other inputs are two 4-bit inputs given by the player to enter a number and game creator to enter password using slide switches. Also, there are three push buttons used for loading the player's number, generating random number and loading the password which also doubles as game start/restart button. Top module has ten output signals which are six 7-segment displays to display player's number, random number, the sum of the two numbers, a pair to display the timer and a pair to display the scorecard at the end of the game. A pair of LEDs are used to indicate the log-in status and a pair of LEDs to indicate the sum status.

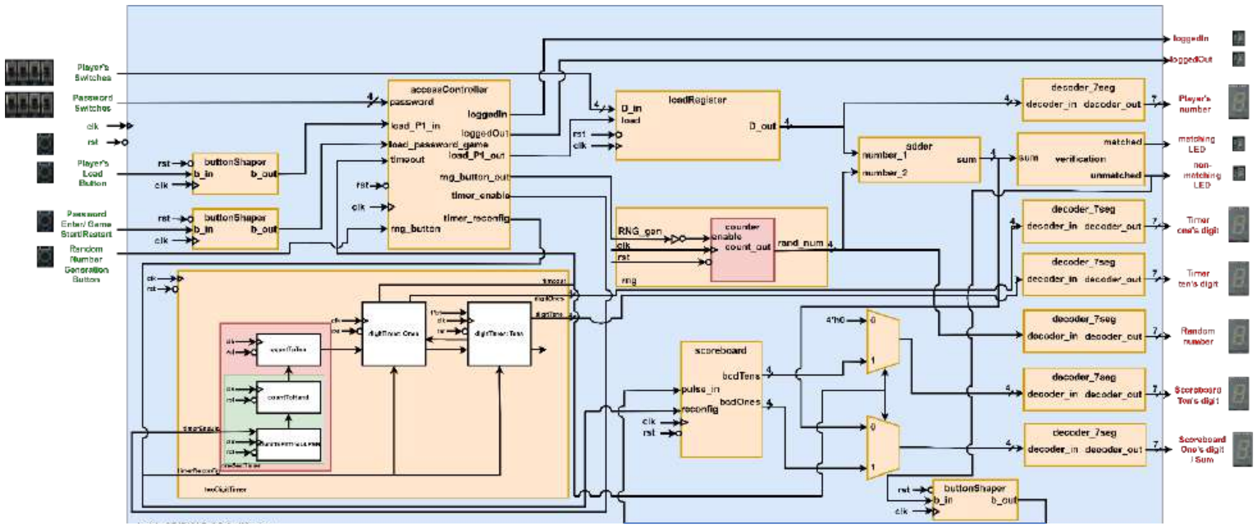


Figure 2.0.1: top-level module

Reset and clock signal definitions:

- rst→ this signal resets the system to the initial stage which is an active low signal.
- clk → is the clock signal which triggers the sequential logic at every positive edge, the frequency of the clock is 50MHz. This signal is important for synchronization.

2.1. Submodule: accessController

accessController module is responsible for authenticating the password entered by the player. This module accepts 4-digits password and asserts load_P1_out and rng_button when logged in, until then these outputs stay de-asserted. The module block diagram is shown in figure 2.1.1. This module also oversees the game operation and asserts the timer_enable and timer_reconfig signals to enable the timer when the game starts and reconfig the timer everytime the game is restarted.

accessController consists of two modules- authentication module and gameControl module. Authentication module takes care of validating the password entered by the player with the default password stored in the on-chip ROM in the reset password stored in the on-chip RAM.

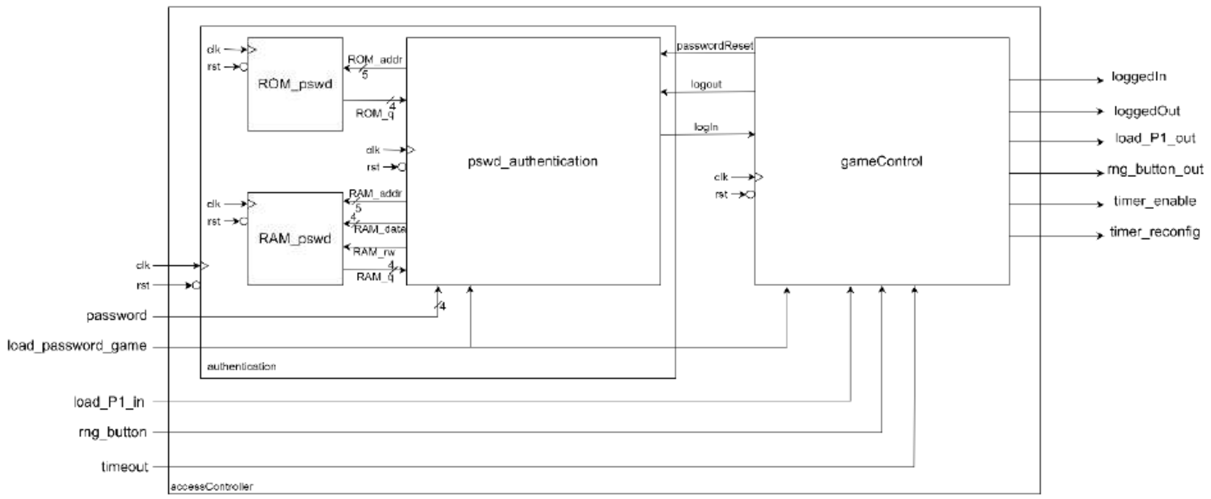


Figure 2.1.1: accessController module

The operation of the module is explained as shown in the finite state machine in figure 2.1.2(a) and figure 2.1.2(b). As we can see, authentication module implements a high-level FSM which has fourteen states. The states start from the INIT state and goes into CHECK_BUTTON state and waits until the player enters a password digit and this is compared with the digit stored in the ROM or RAM depending on if the player has reset the password or not.

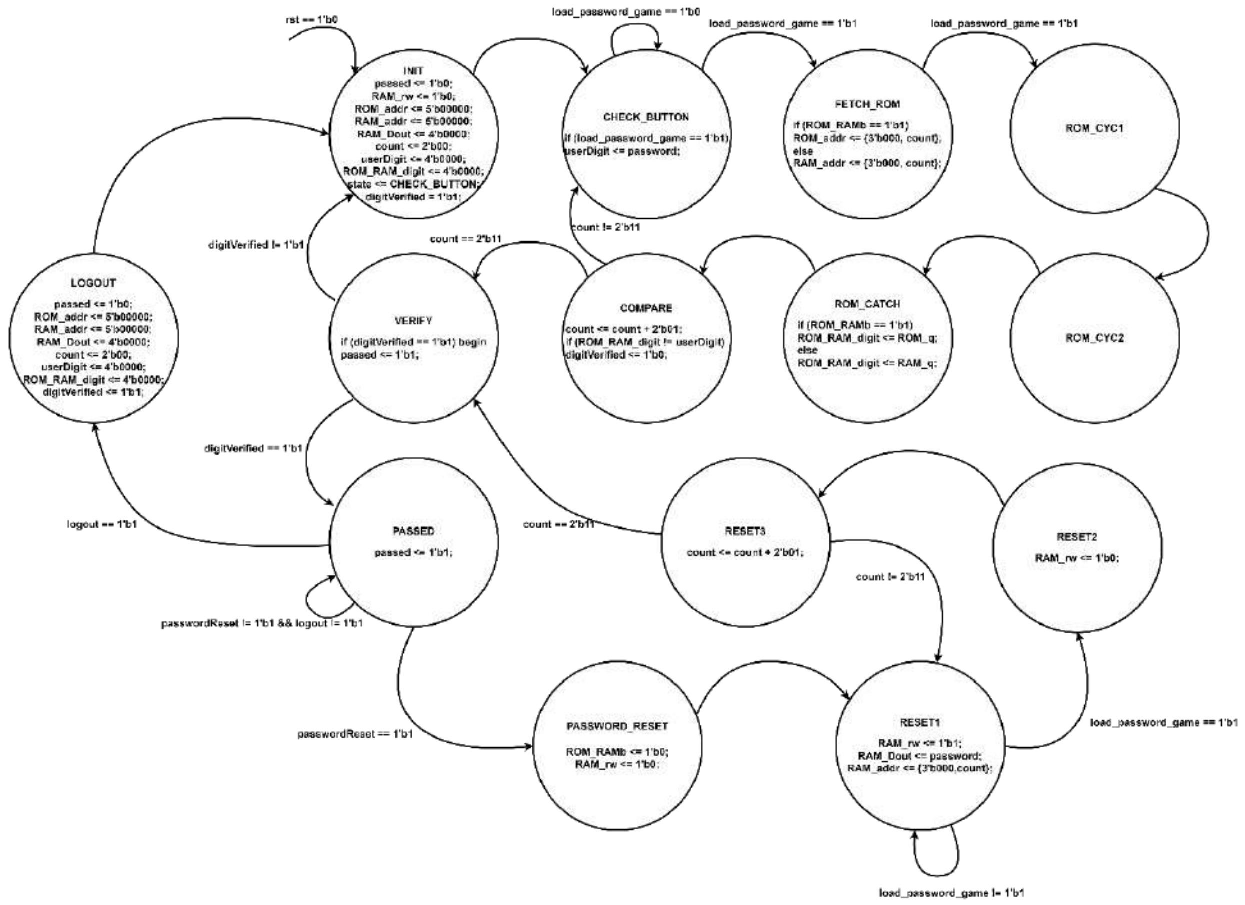


Figure 2.1.2 (a): authentication FSM

Figure 2.1.2(a) also shows the FMS to logout and password reset. When password reset signal is received, the state is transitioned into password PASSWORD_RESET and accepts new password from the player. Figure 2.1.2(b) shows the states and state transitions in the gameControl module. This module is responsible for taking the player inputs such as rng_button or player_load button. This module also interacts with the timer module to indicated when to start the timer and when the timer has runout. All of these states are defined in the diagram below.

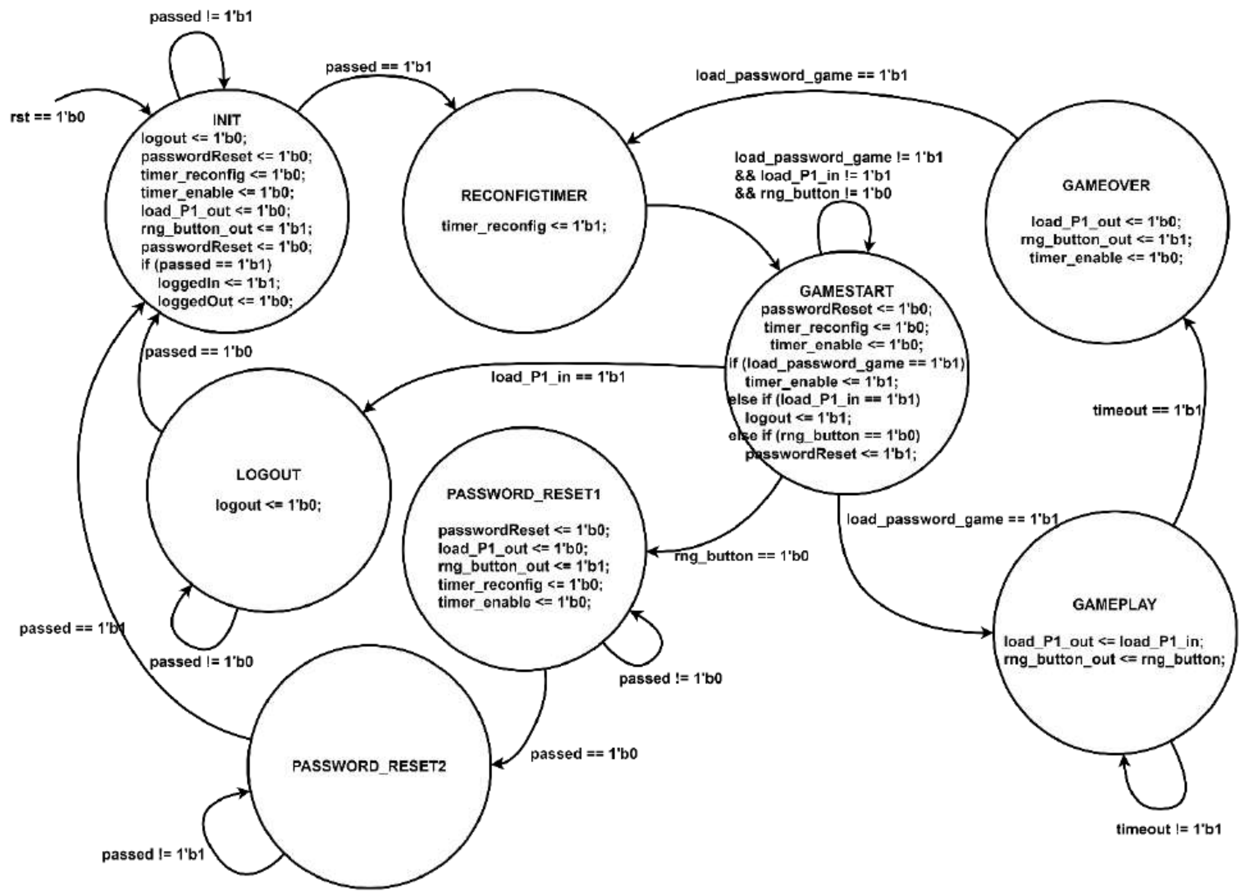


Figure 2.1.2 (b):gameControl FSM

2.2. Submodule: 1ms LFSR counter

1ms timer module outputs a pulse with a width of one clock pulse every 1ms. As we can see in the figure 2.2.1 below, module has three inputs – counter_in, clk and rst and has one output – pulse_out. This module is basically a counter which counts 50,000. However, since this module was designed based on Linear Feedback Shift Register (LFSR), the terminal value is 16’hA168.

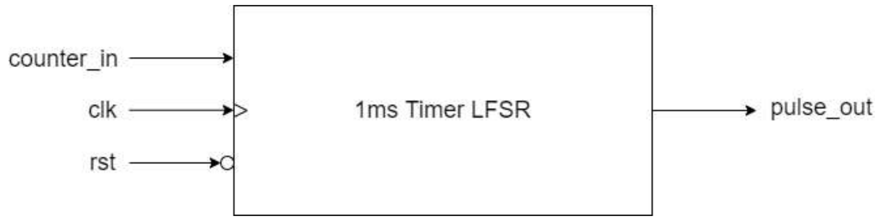


Figure 2.2.1: 1ms timer module

This module has three input signals:

- clk → is the clock signal which triggers the sequential logic at every positive edge, the frequency of the clock is 50MHz.
- rst→ this signal resets the module to the initial stage which is an active low signal
- counter_in → This is level triggering signal which will start and keep the counter running as long as the signal is high.

and has one output signal:

- pulse_out → a pulse signal every 1ms second with a width of one clock pulse.

2.3. Submodule: oneSecTimer

One-second timer module outputs a pulse with a width of one clock pulse every 1 sec. As we can see in the figure 2.3.1 below, there are three submodules -- countToTen, countToHund and countToFifThou. These submodules are basically counters with terminal values of 10, 100 and 50,000 respectively. However, for the ModelSim simulation sake, these terminal values where changed to 2, 3, and 4 respectively. The 50,000 counter module is designed using Linear Feedback Shift Register as shown in Figure 2.3.1.

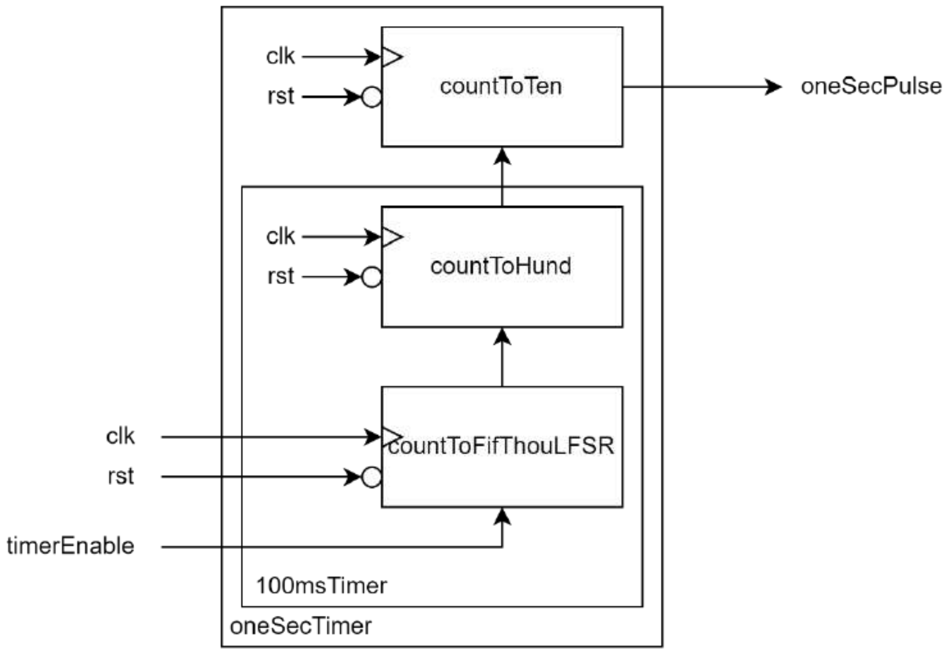


Figure 2.3.1: One-Second timer module

This module has three input signals:

- clk → is the clock signal which triggers the sequential logic at every positive edge, the frequency of the clock is 50MHz.
- rst→ this signal resets the module to the initial stage which is an active low signal
- enable → This is level triggering signal which will start and keep the counter running as long as the enable signal is high.

and has one output signal:

- oneSecPulse → a pulse signal every one second with a width of one clock pulse.

2.4. Submodule: twoDigitTimer

The twoDigitTimer module is a count-down timer that counts from 99 to 0 decremented every one second. The principle behind this module is a one second timer the outputs a pulse every one second which is passed to ones-digit timer that counts down from 9 to 0 for every pulse it receives and then sends a pulse to tens-digit timer until both the digits become zero after which the timeout signal is asserted indicating game over. The twoDigitTimer is shown in figure 2.4.1 below.

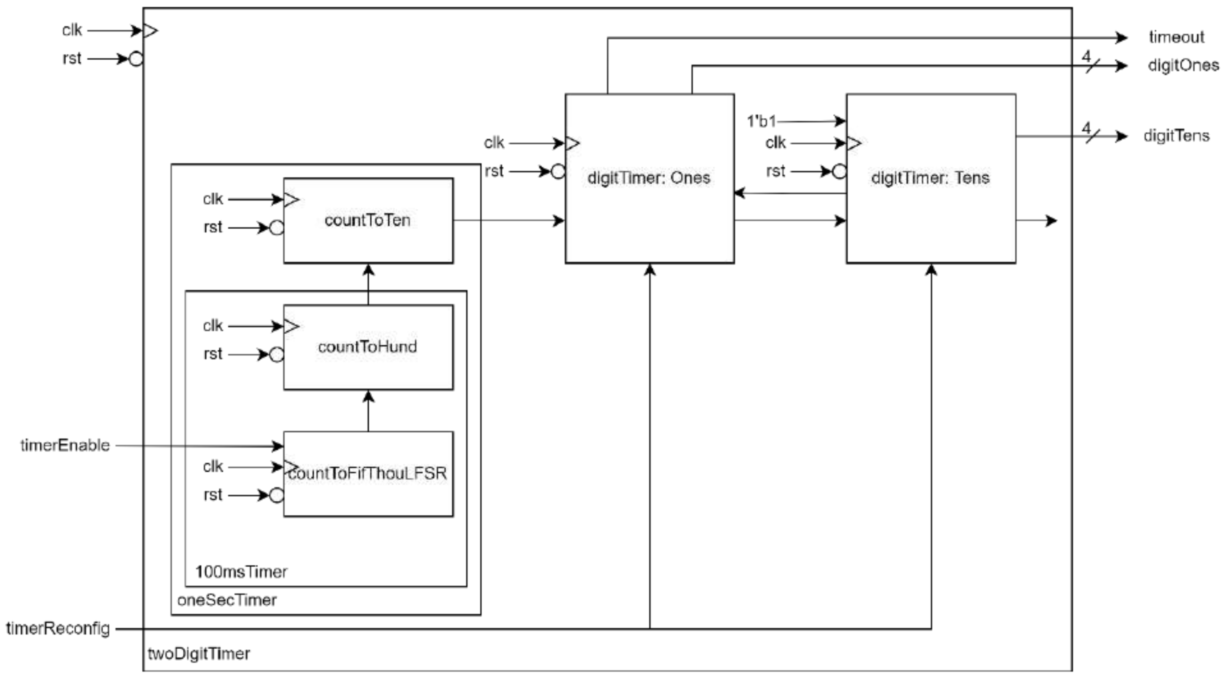


Figure 2.4.1: twoDigitTimer module

This module has two inputs apart from clk and rst:

- timerEnable → this is a 1-bit signal, source of this signal is the accessController module and is asserted high when the game starts and becomes low when the timer runs out.
- timerReconfig → this is a 1-bit pulse signal, which will set the digits on the digit counter to 99, making the timer ready for counting down when game starts.

and three output signals:

- digitOnes and digitTens → these are 4-bit binary numbers generated by the two digitTimer modules present inside the twoDigitTimer module. These numbers are displayed on the 7-segment display.
- timeout → this is a 1-bit signal which is asserted high when the digits become 00 and the timer stops.

2.5. Submodule: rng

The rng module is the module which generates a random number. The principle behind the random number generator is a 4-bit counter module which starts counting as long as the enable signal is high as shown in the figure 2.5.1. The inverted raw signal from a push-button is passed to the counter module as the enable signal, and hence the duration of the push will determine the counter's final value in turn the random number.

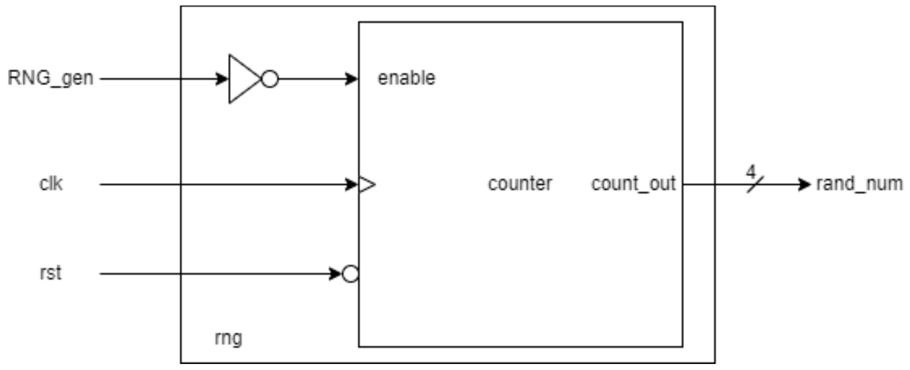


Figure 2.5.1: rng module

This module has one input apart from clk and rst:

- RNG_gen → this is a 1-bit signal, source of this signal is the inverted raw signal from the push-button which is an active low signal.

and one output signal:

- rand_num → this is a 4-bit binary number generated by the counter present inside the rng module. This number is displayed on the 7-segment display.

2.6. Submodule: scoreboard

The scoreboard module keeps track of the player's number of correct responses. The internal structure of scoreboard is two bcd counters which increments every time the player presses the "load player number" button and the sum is 0xF. The scoreboard module is shown in figure 2.6.1.

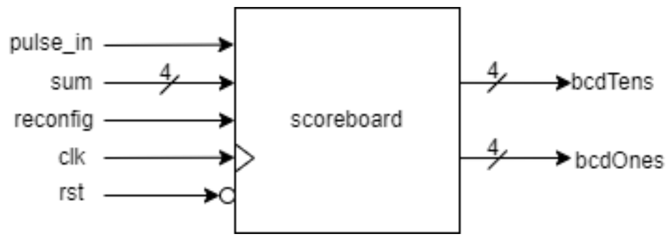


Figure 2.6.1: scoreboard module

This module has three inputs apart from clk and rst:

- sum → this is a 4-bit input signal received from the adder module. This is used to determine whether the number entered by the player is correct or not.
- pulse_in → this is a 1-bit pulse signal, when this signal is high, the internal bcd counter will increment by 1.
- reconfig → this is a 1-bit pulse signal, which will set the digits on the digit counter to 00, making the scorebaord ready for counting when game starts.

and two output signals:

- bcdOnes and bcdTens → these are 4-bit binary numbers generated by the two bcd counter modules present inside the scoreboard module. These numbers are displayed on the 7-segment display.

2.7. Submodule: buttonShaper

The frequency on FPGAs is around 50MHz, which is 20ns per clock cycle. Since human is not fast enough to push and release the button within a clock cycle, each press might take about thousands of cycles, so passing the raw signal from the push button to the load register will load the register multiple times which is not feasible in power consumption perspective. Button shaper module takes the raw 1-bit signal from the push button and generates a single pulse of 1-bit with a width of one clock cycle i.e., for each push-release action only a single pulse is generated. The block diagram of the button shaper module is shown in Figure 2.7.1 below.

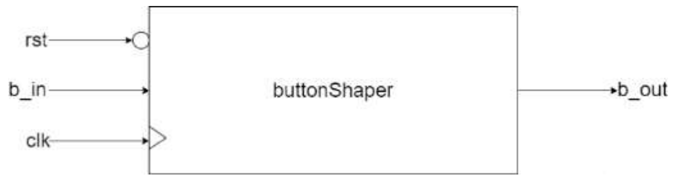


Figure 2.7.1: Button shaper module

This module has a 1-bit input:

- b_in → raw signal from the push button which is an active low signal

and one 1-bit output:

- b_out → this is a single pulse signal generated by the module for every push-release action which is an active high 1-bit signal.

The finite state machine shows the state transitions happening in the module as shown in the Figure 2.7.2.

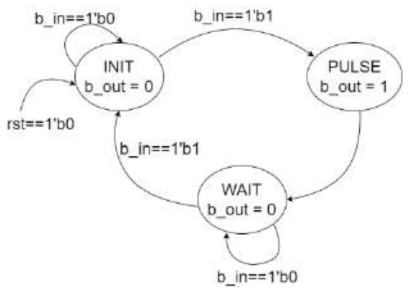


Figure 2.7.2: buttonShaper FSM

As we can see in the above diagram, button shaper has three states:

- INIT: This is the initial state of the module, and the output during this state is low as long as the b_in is not high, in which case the state transition happens and goes to PULSE state.
- PULSE: The module stays in this state for exactly one clock cycle during which the b_out is asserted to high after the b_out is pulled down to low and the state transits to WAIT.
- WAIT: This state is responsible for keeping the d_out low even when the button is being pressed after the pulse signal is generated. The module stays in this state as long as the b_in does not go back to high, after which it goes back to INIT state.

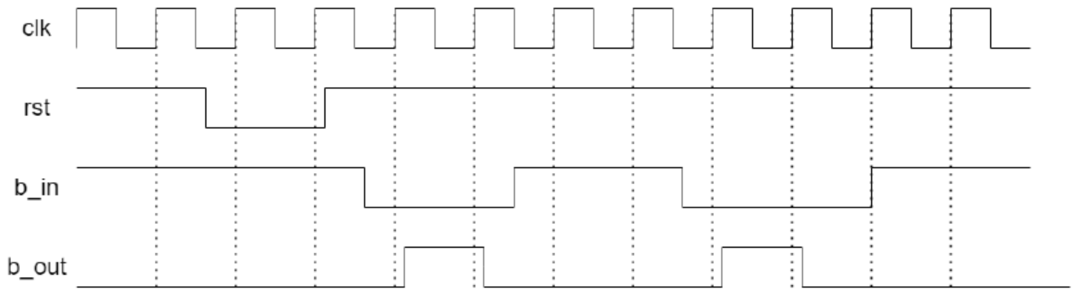


Figure 2.7.3: Expected Timing Diagram

Figure 2.7.3 shows the expected timing diagram of the button shaper module. We expect to get a single pulse signal every time a push-release action is performed. We also expect to keep the b_out low after the pulse is generated and until b_in is asserted again.

2.8. Submodule: loadRegister

Load register is a special register which updates its value only when a load signal is asserted. This register is used between player's slide switches and 7-segment display so that when the players use the slide switches to enter the number, it doesn't show up in the display as intermittent values, rather a load button is used to confirm the player's input. This block is shown in figure 2.8.1.

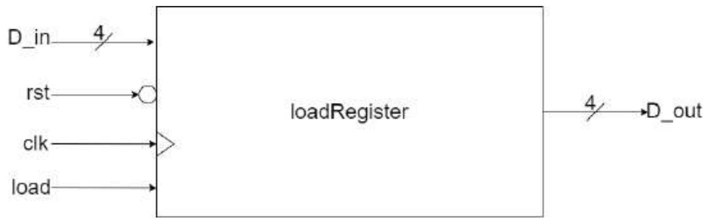


Figure:2.8.1: loadRegister module

This module has two inputs:

- D_in → this is a 4-bit signal, source of this signal is the slide switches used by the players to enter the number.
- load → this signal is responsible for updating the register value when the load signal is high. The source of this signal is accessController block, it's a one-pulse active high signal.

and one output signal:

- D_out → this signal gets updated at every load signal and will retain its value the load signal is low.

2.9. Submodule: decoder_7seg

This is a decoder_7seg module which is responsible for generating the 7-bit binary number. This module has one 4-bit input signal decoder_in, and outputs a 7-bit decoder_out signal which drives the 7-segment display as shown in figure 2.9.1.

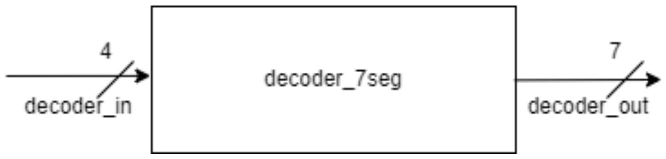


Figure 2.9.1: decoder_7seg module

2.10. Submodule: adder

The adder module takes in two 4-bit inputs and computes the sum and outputs a 4-bits binary number shown in figure 2.10.1. Since the output is just 4-bits binary number, when the sum exceeds binary 1111, it is rolled over to binary 0000, and the carry bit is discarded.

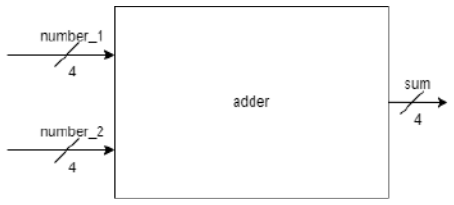


Figure 2.10.1: adder module

2.11. Submodule: verification

The verification module is incorporated into the design to indicate whether the sum is binary 1111 or not. If it is, matched signal is asserted to high and unmatched is asserted to low and vice versa. These signals as used to drive the two LEDs shown in figure 1.0.1. The verification module is shown in the figure 2.11.1.

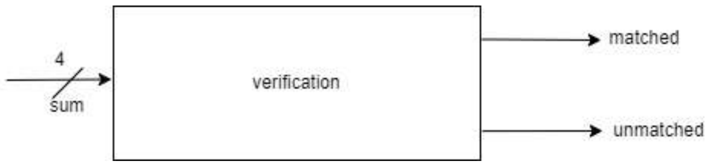


Figure 2.11.1: verification module

3.0. Simulation Results

All the submodules were simulated using ModelSim and the results are as follows.

3.1. Submodule: accessController

Figure 3.1.1 shows the simulation results of the accessController module. Here the one gameplay round is simulated by entering the password, resetting the password and logging-in again with new password. Player's load button, random number generating button, game start/restart button where also simulated and we can see that these signals are not passed to the output until the player is logged in and game start button is pressed.

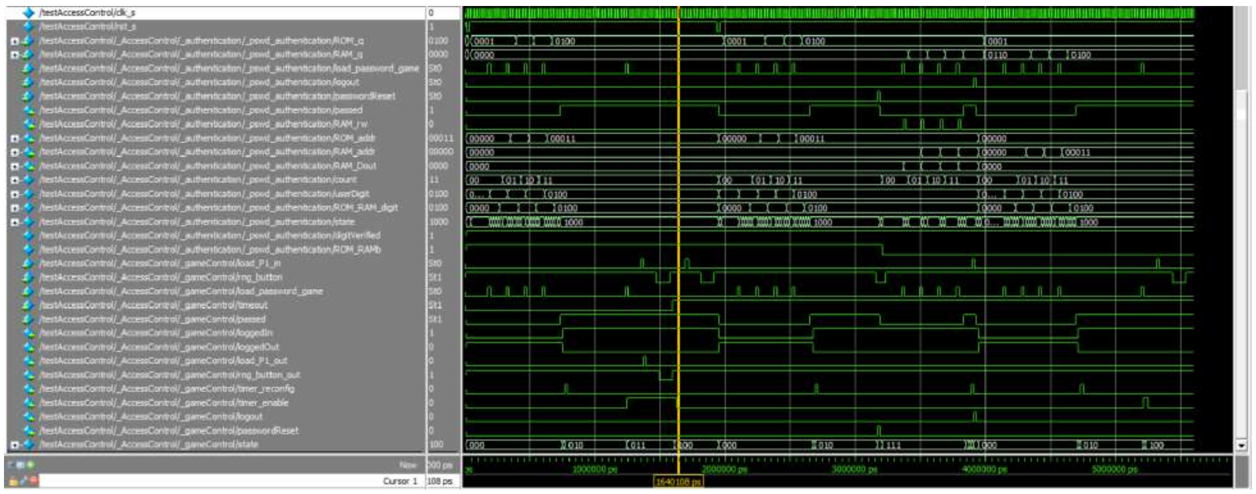


Figure 3.1.1: accessController module simulation result

3.2. Submodule: 1ms LFSR counter

For determining the terminal value of the LFSR counter, 16-bit binary counter was used. In the Figure 3.2.1 we can see that the pulse_out_s_1 signal become high after the value 49,999 in the 16-bit binary counter. The corresponding value at 16-bit LFSR counter is 16'hA168. This value was included in the design module as the terminal value, and we can see in the figure 3.2.2 that the pulse_out_s_2 also become high when the pulse_out_s_1 become high, verifying that the LFSR module is operating properly and can be replaced for 16-bit binary counter.

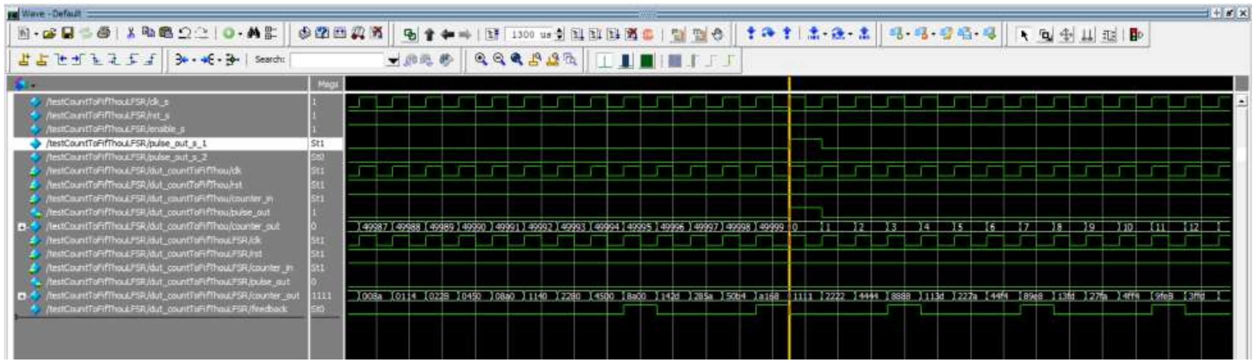


Figure 3.2.1: Simulation results of 1ms timer for determining the terminal value for LFSR from ModelSim

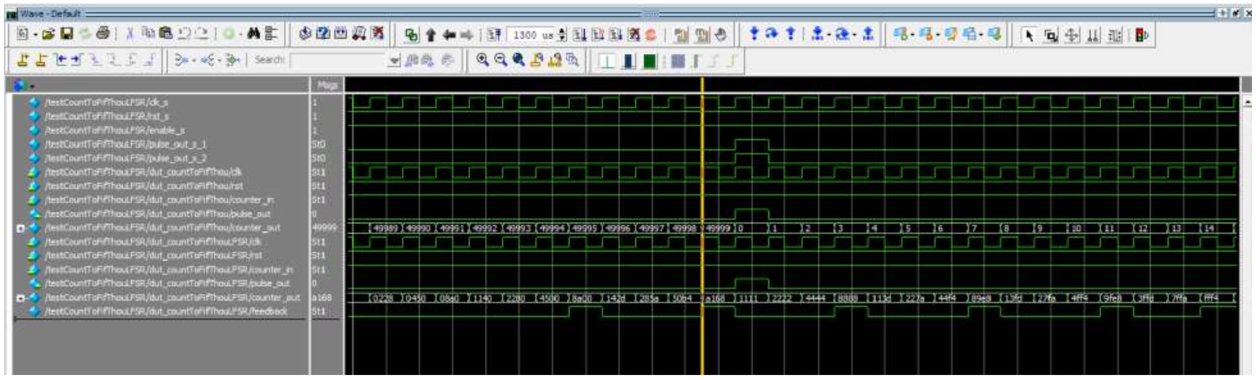


Figure 3.2.2: Simulation results of 1ms timer after setting the terminal value for LFSR from ModelSim

3.3. Submodule: rng

Figure 3.3.1 shows the simulation results of random number generator module. Here we can see that when RNG_gen signal is asserted for unknown duration, the internal counter will start incrementing by 1. When the counter value reaches 0xF, the output is rolled back to 0x0 and continues to count up as long as the RNG_gen signal is asserted low.

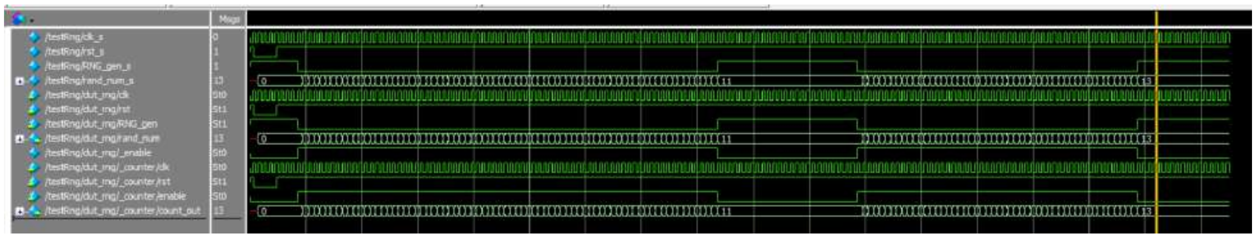


Figure 3.3.1: rng module simulation result

3.4. Submodule: twoDigitTimer

The twoDigitTimer module consists of two digitTimers which counts down from 9 to 0 every time it receives a one clock cycle pulse. We can see that when both one's digit and tens digit is set to 9 when a pulse of recongif signal is received. When the timer enable signal is asserted high, the timer starts running and one's digit starts decrementing every one second (changed to every 24 clock cycles for the sake of simulation). This is shown in figure 3.4.1.

In the figure 3.4.2 we can see that when one's digit becomes 0, a pulse is sent out from one's digit module to ten's digit module and then the one's digit is rolled back to 9, and the pulse is sent to the ten's digit module which will be used for decrementing the value.

In the figure 3.4.3 we see that when ten's digit becomes 0, the ten's digit module asserts noBorrowUp signal, and when one's digit reaches 0, the borrowUp pulse is not asserted rather, the timeout signal is asserted high.

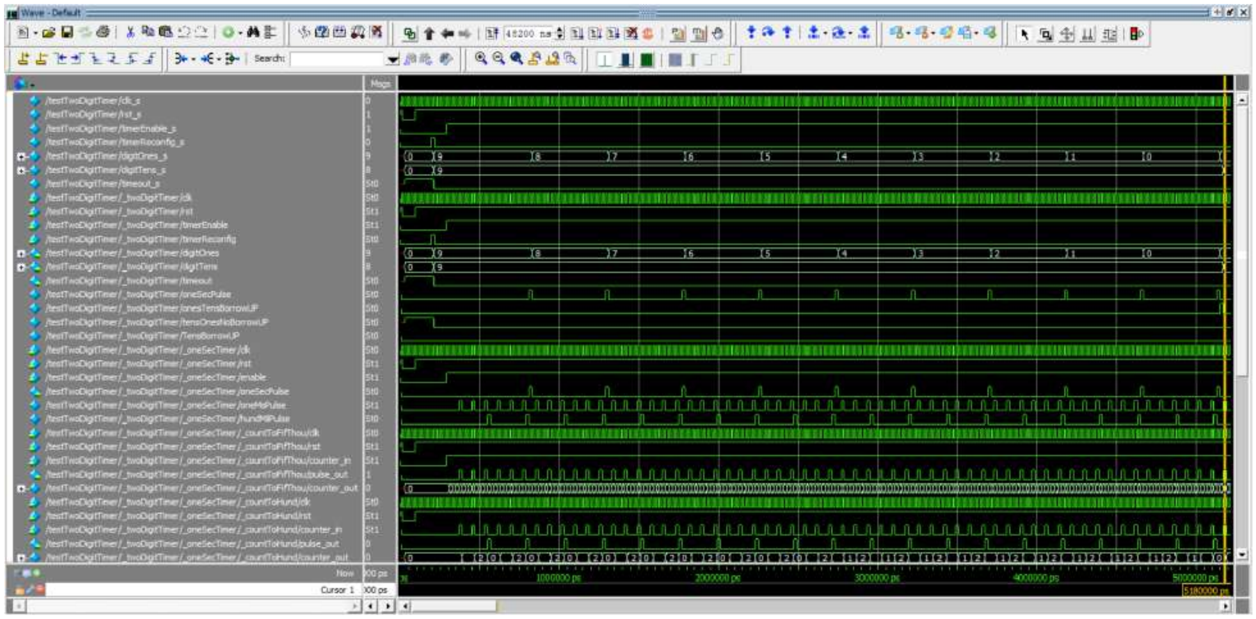


Figure 3.4.1: twoDigitTimer module simulation result (1 of 3)

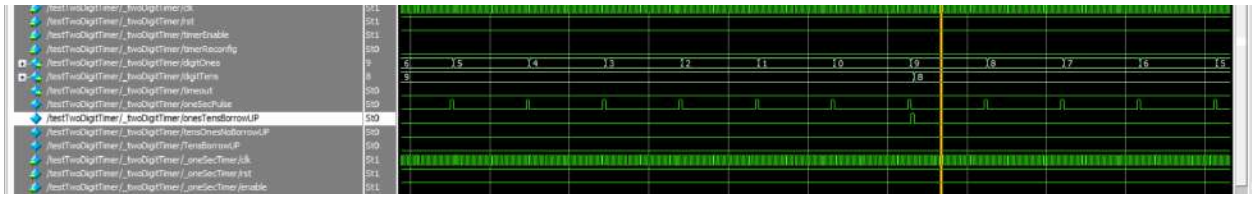


Figure 3.4.2: twoDigitTimer module simulation result (2 of 3)

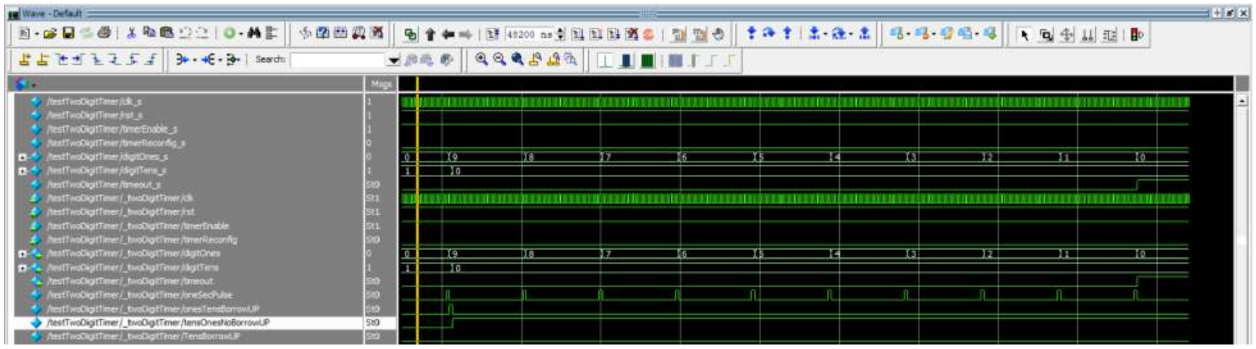


Figure 3.4.3: twoDigitTimer module simulation result (3 of 3)

3.5. Submodule: digitTimer

Figure 3.5.1 shows the simulation results of digit timer, we can see that when the digit timer is reset, the value becomes 0 until a reconfig pulse is received. At every borrow down singal, the digit timer counts down to 0 and stays 0 until the noBorrowUP signal is high, otherwise, the digit timer will roll back to 9.

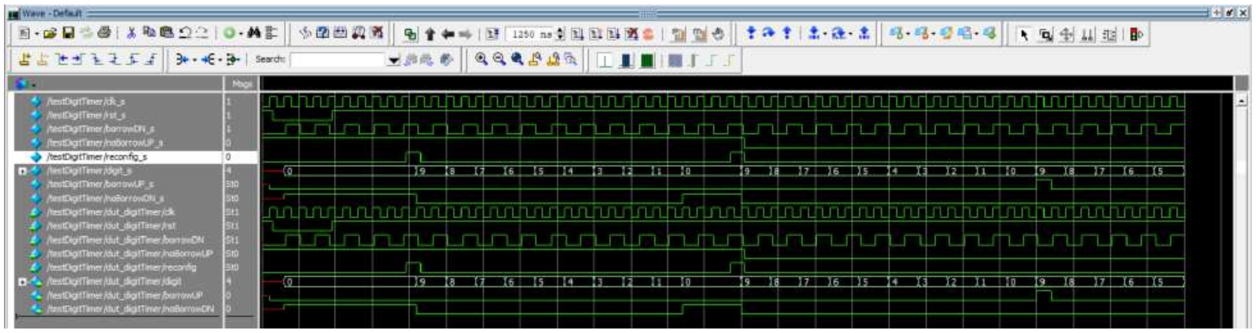


Figure 3.5.1: digitTimer module simulation result

3.6. Submodule: oneSecTimer

For the purpose of simulation's sake, the terminal values of the counters were reduced to 4, 3 and 2 for the submodules countToFifThou, countToHund and countToTen respectively. From the simulation result shown in the figure 3.6.1 we can see that when the enable signal is high, the pulse_out signal from countToFifThou is a pulse signal every 1 ms. This pulse is the driving signal for the coutToHund module. The output of coutToHund is high every 100 ms for a width of one clock pulse. This is signal in turn drives the countToTen module which outputs a high pulse every 10 counts i.e. one second. However, as mentioned above, since the terminal values are changed, the output of the one-second timer is high every 24 clock cycles i.e, 4 times 3 times 2.

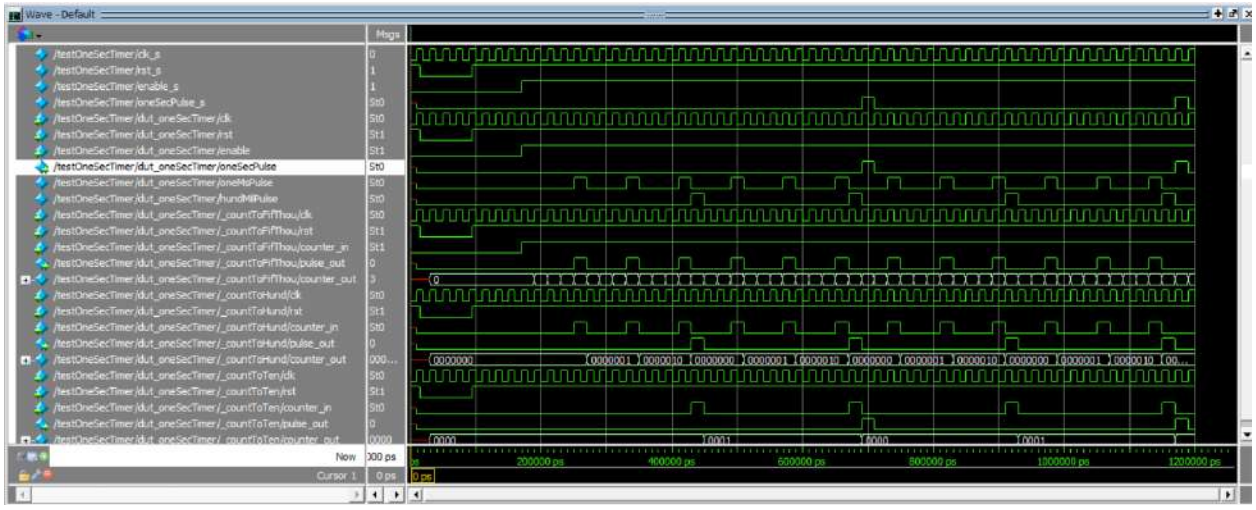


Figure 3.6.1: Simulation results of one-second timer from ModelSim

3.7. Submodule: buttonShaper

Figure 3.7.1 below shows the simulation results from ModelSim. We can see that when b_in is asserted to low, a pulse is generated at the b_out for one cycle. We also see that the b_out stays low until the b_in goes low again.

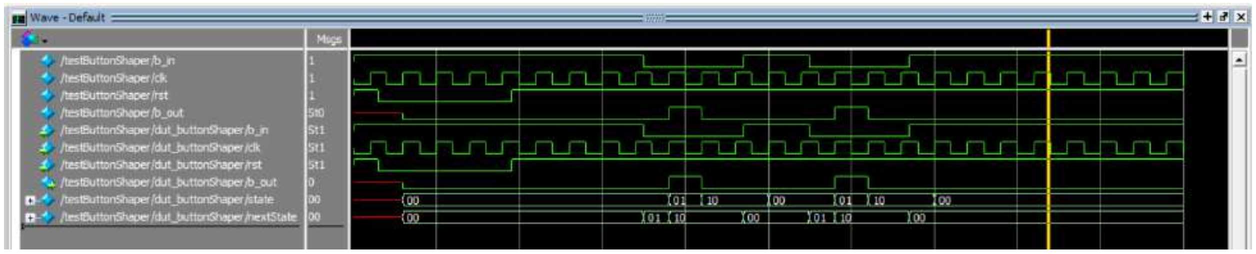

Wave - Default
/testButtonShaper/b_in
/testButtonShaper/clk
/testButtonShaper/rst
/testButtonShaper/b_out
/testButtonShaper/dut_buttonShaper/b_in
/testButtonShaper/dut_buttonShaper/clk
/testButtonShaper/dut_buttonShaper/rst
/testButtonShaper/dut_buttonShaper/b_out
/testButtonShaper/dut_buttonShaper/state
/testButtonShaper/dut_buttonShaper/nextState

Figure 3.7.1: buttonShaper module Simulation result

3.8. Submodule: loadRegister

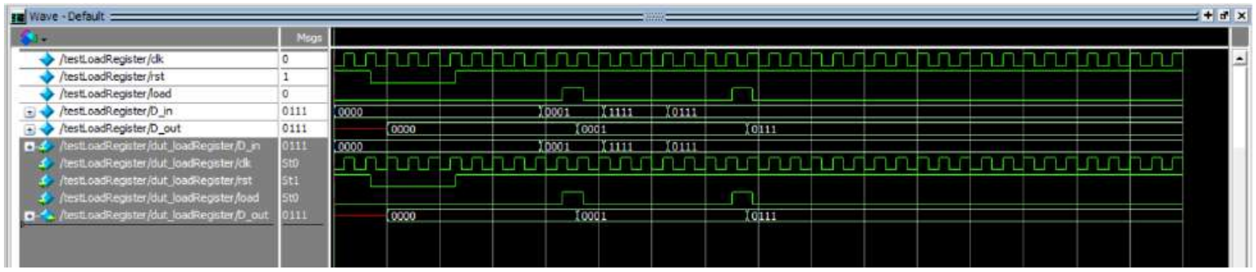

Wave - Default
/testLoadRegister/clk
/testLoadRegister/rst
/testLoadRegister/load
/testLoadRegister/D_in
/testLoadRegister/D_out
/testLoadRegister/dut_loadRegister/D_in
/testLoadRegister/dut_loadRegister/clk
/testLoadRegister/dut_loadRegister/rst
/testLoadRegister/dut_loadRegister/load
/testLoadRegister/dut_loadRegister/D_out

Figure 3.8.1: loadRegister module simulation result

From the above simulation result (figure 3.8.1), it shows that when D_in is asserted, the value at output does not change unless the load is high.

Initially we see that the D_out is 4’bX until the rst pin is asserted to low, after which it goes to 4’b0. We also observe that when D_in is 4’h1 the D_out does not change unless the load signal is asserted. This can be verified when we see the D_in is changed to 4’hF and then to 4’h7 and we only see the D_out changes to 4’h7 because the load signal was only asserted for the later.

3.9. Submodule: decoder_7seg

The timing waveform is shown in the figure 3.9.1. Here 16 test cases have been implemented for the decoder_7seg module since the input to the decoder is 4-bits wide. Each segment can be turned ON or OFF by driving logic low or high respectively. Let us take an example to display 0x0- we can see that the output of the decoder is 1000000 which means the segment 6 is OFF and the rest are turned ON as shown in the figure 3.9.2. below.

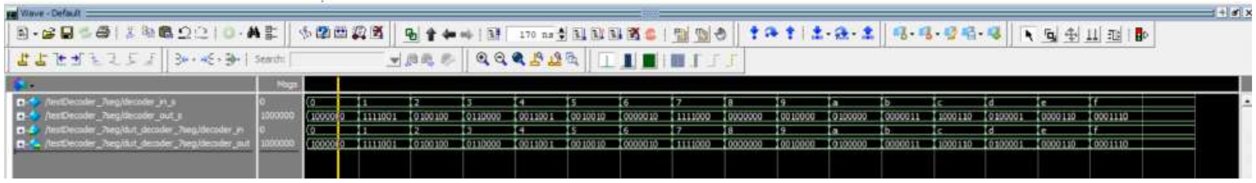

Wave - Default
/testDecoder_7seg/decoder_in_s
/testDecoder_7seg/decoder_out_s
/testDecoder_7seg/dut_decoder_7seg/decoder_in
/testDecoder_7seg/dut_decoder_7seg/decoder_out
0 1 2 3 4 5 6 7 8 9 a b c d e f
1000000 1111001 0100100 0110000 0011001 0010010 0000010 1111000 0000000 0010000 0100000 0000011 1000110 0100001 0000110 0001110

Figure 3.9.1: decoder_7seg module simulation results

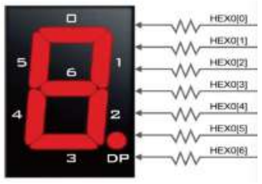


Figure 3.9.2: 7-Segment display

3.10. Submodule: adder

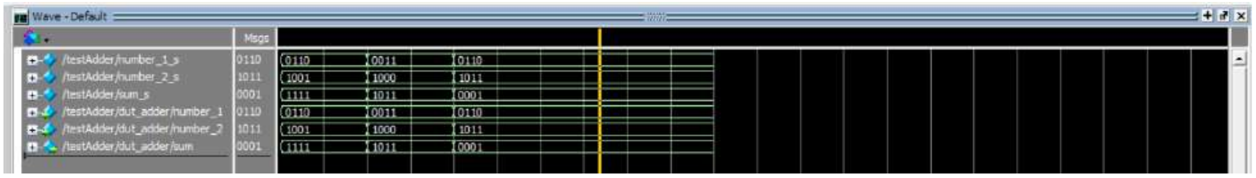

Wave - Default
Msgs
/testAdder/number_1_s 0110 0110 0011 0110
/testAdder/number_2_s 1011 1001 1000 1011
/testAdder/sum_s 0001 1111 1011 0001
/testAdder/dut_adder/number_1 0110 0110 0011 0110
/testAdder/dut_adder/number_2 1011 1001 1000 1011
/testAdder/dut_adder/sum 0001 1111 1011 0001

Figure 3.10.1: adder module simulation result

Here 3 test cases have been implemented:
1. Matching criteria: where the two numbers add up to 0xF.
2. Non-matching criteria: where the two numbers don't add up to 0xF but is less than 0xF.
3. Non-matching criteria: where the two numbers don't add up to 0xF but is more than 0xF.

Note that in testcase 3, when number_1 and number_2 (0x6 and 0xB) adds up to give the sum more than 4-bits (0x11), the sum rolls over to zero and a carry is generated but since we are discarding the carry, the output is only 4-bits (0x1).

4.0. FPGA Board Testing
FPGA board testing is done as follows:

1. FPGA was set to run mode and the default state is shown in figure 4.0.1.
2. The player enters the password to login. The timer is configured to 99 and waits for the player to press Game Start button. This is shown in figure 4.0.2(a).
3. Player enters Game Start button and the timer starts counting down. This is shown in figure 4.0.3.
4. Player pushes the random number generation button to generate a random number. This is shown in figure 4.0.4.
5. Player enters a number to make the sum 0xF. The sum is displayed, and the matching LED glows as shown in figure 4.0.5.
6. As the countdown of timer continues, the player presses the random number generation button again to generate a new random number. When this happens, the matching LED is turned OFF and the non-matching LED glows as shown in figure 4.0.6.
7. Player enters a new number to match the new random number which is shown in figure 4.0.7. With this, the player has matched the random number twice in this round.
8. The timer has run out and the player can enter a number nor generate a new random number. The FPGA will now display the scoreboard and wait for the player to restart the game. This is displayed in figure 4.0.8.
9. Player pushes the restart button which resets the timer back to 99 as shown in figure 4.0.9.
10. Player pushes Game Start button again for round 2 and timer starts to count down as shown in figure 4.0.10.
11. The player will logout in between the game sessions by pressing the Player's Load/Logout button and can log back in with the password. This is shown in figure 4.0.2(b).
12. The player should press the Random Number/Password Reset button to reset the password in between the sessions and can set the password similar to entering the password for logging in.

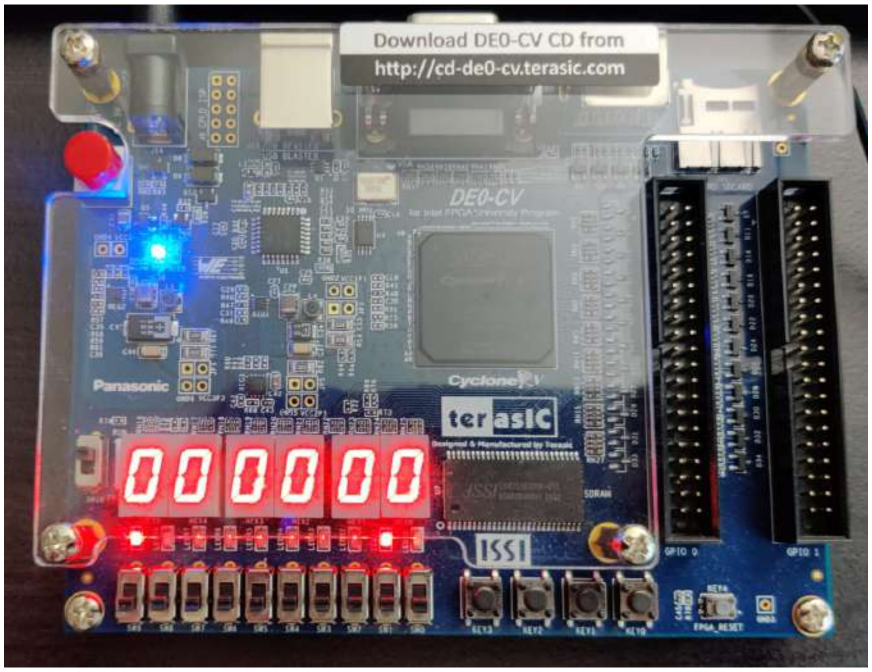


Figure 4.0.1: Default state

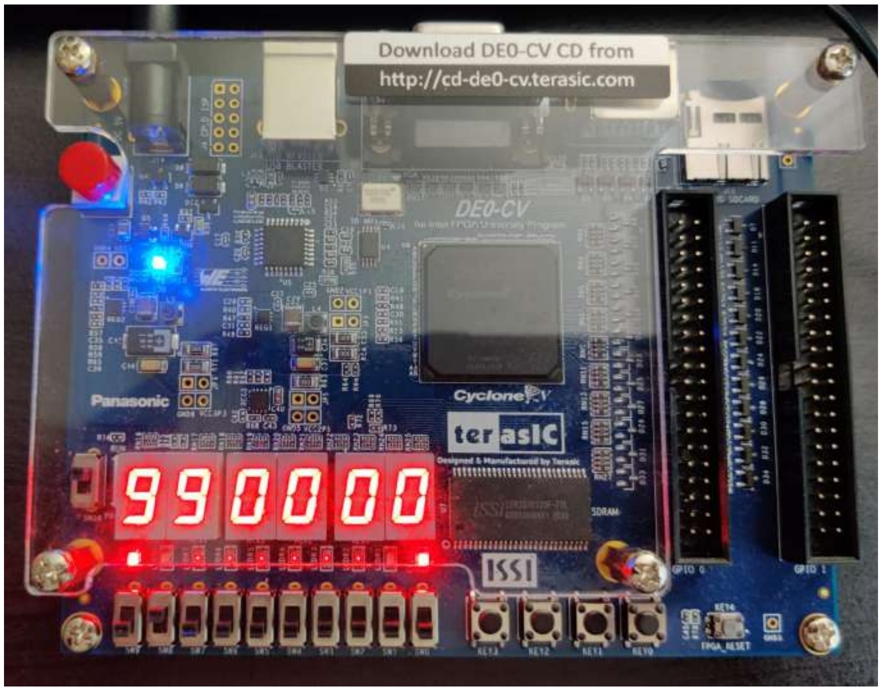


Figure 4.0.2 (a): Logged In by player

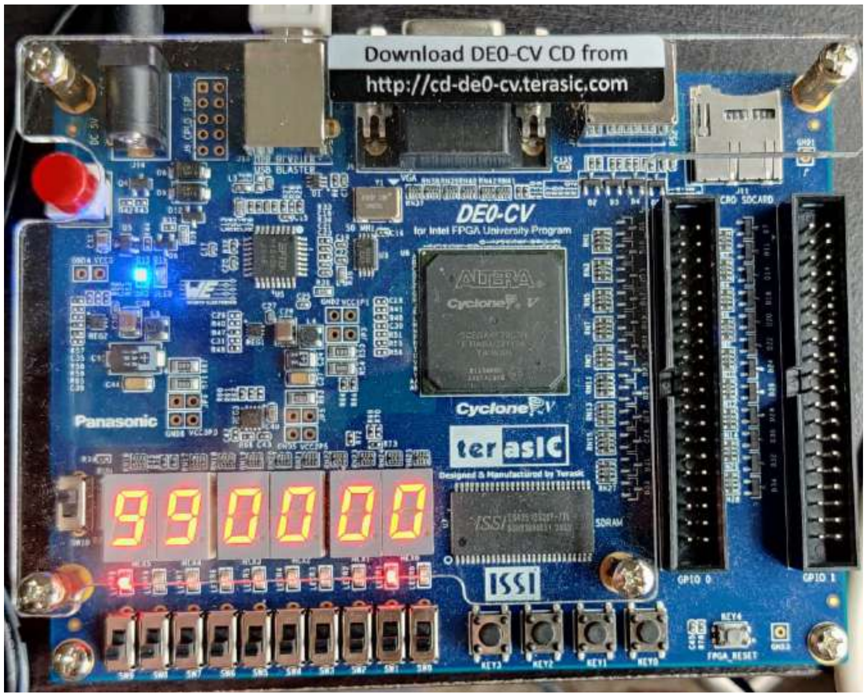


Figure 4.0.2 (b): Logged Out by player

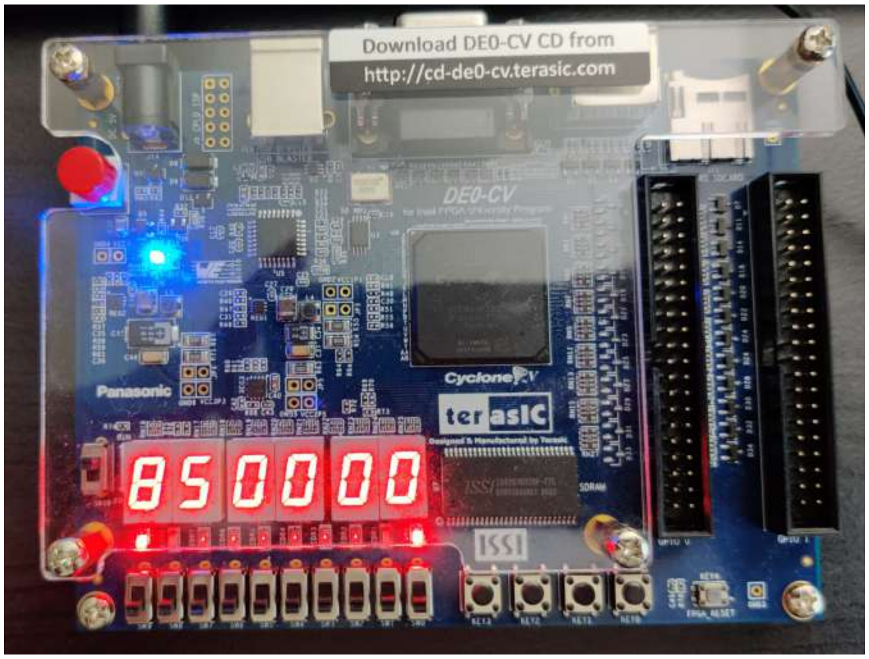


Figure 4.0.3: Player starts the game, timer counts down

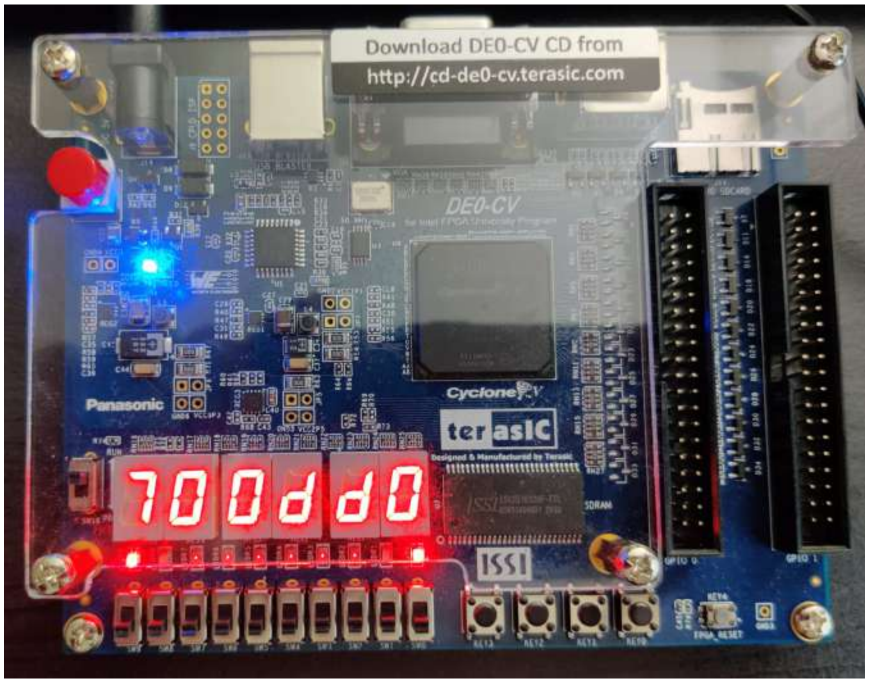


Figure 4.0.4: Player pushes random number generation button

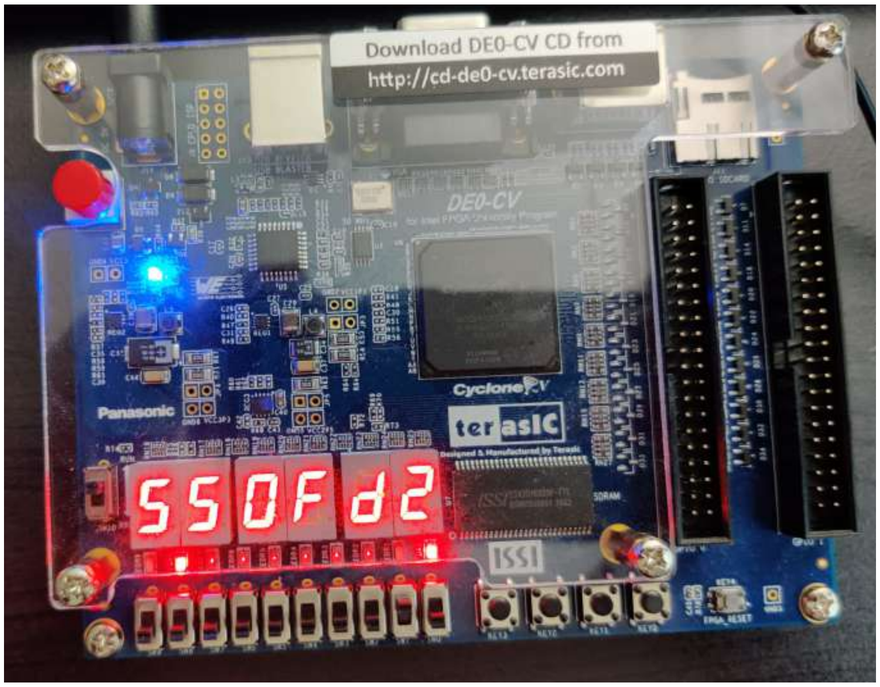


Figure 4.0.5: Player enters a number to match the random number

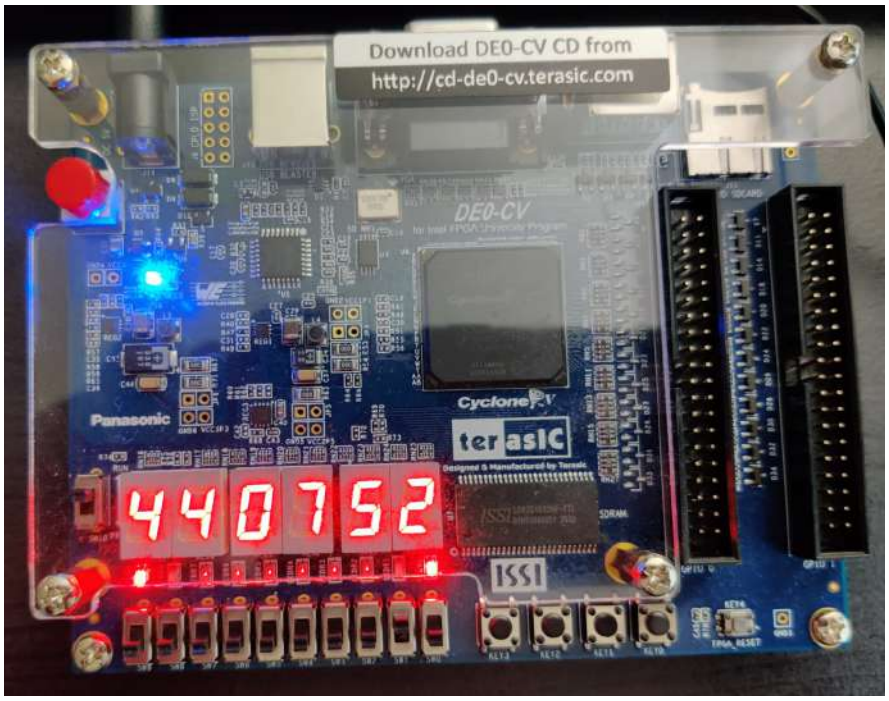


Figure 4.0.6: Player pushes the random number generation button again

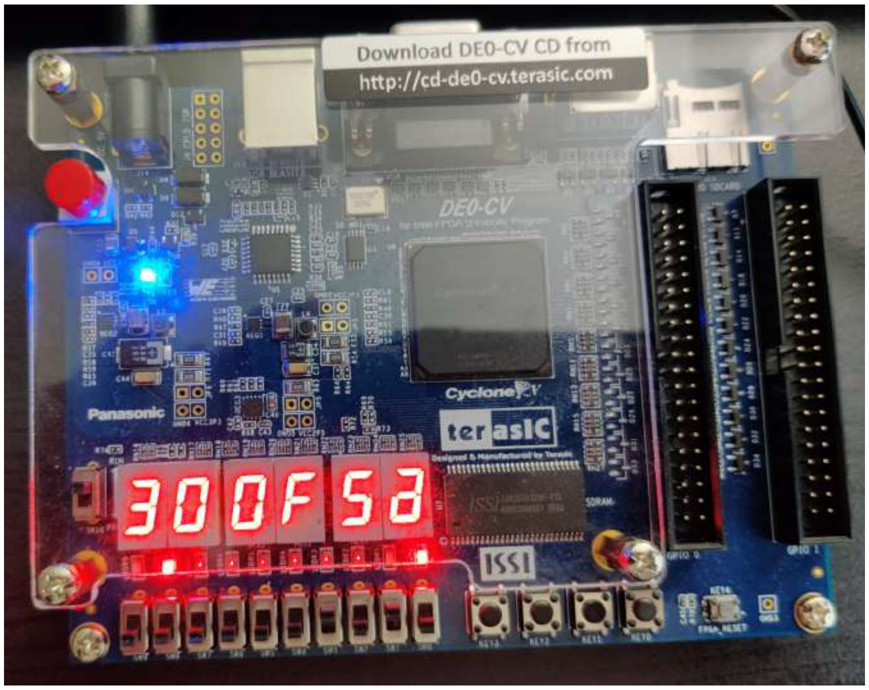


Figure 4.0.7: Player matches the new random number again

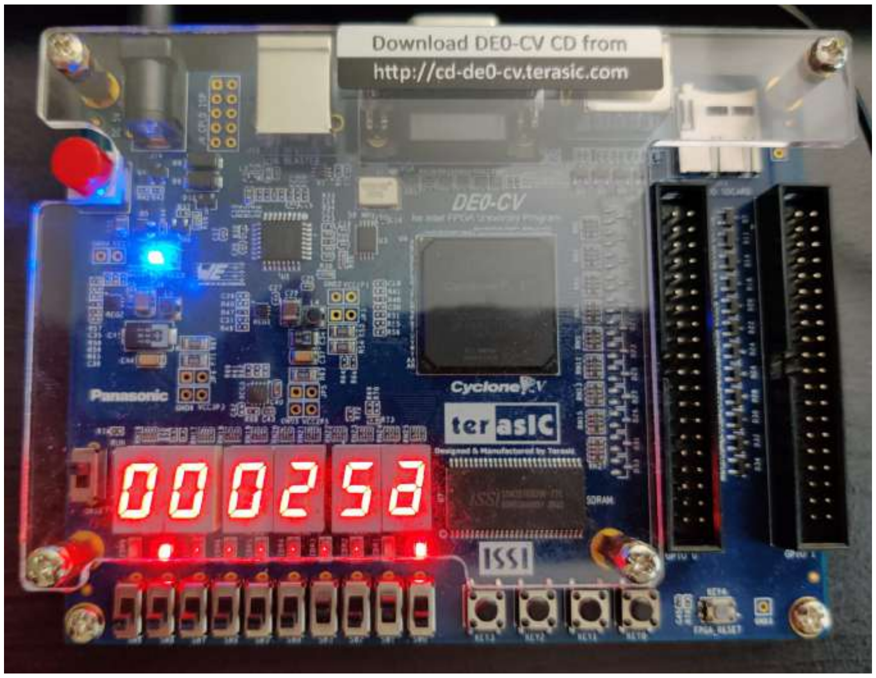


Figure 4.0.8: FPGA displays the scoreboard

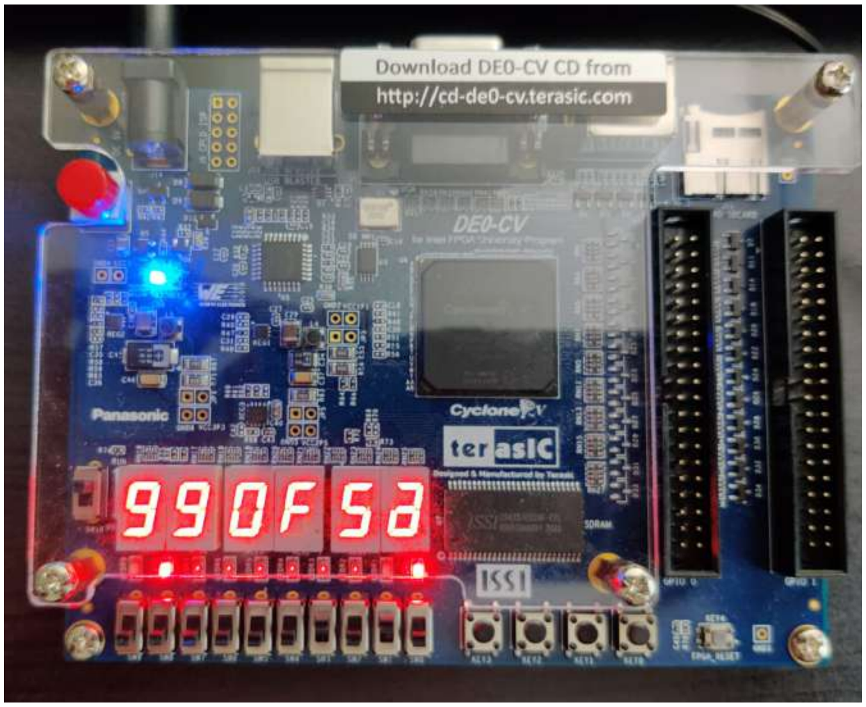


Figure 4.0.9: Player Resets the game

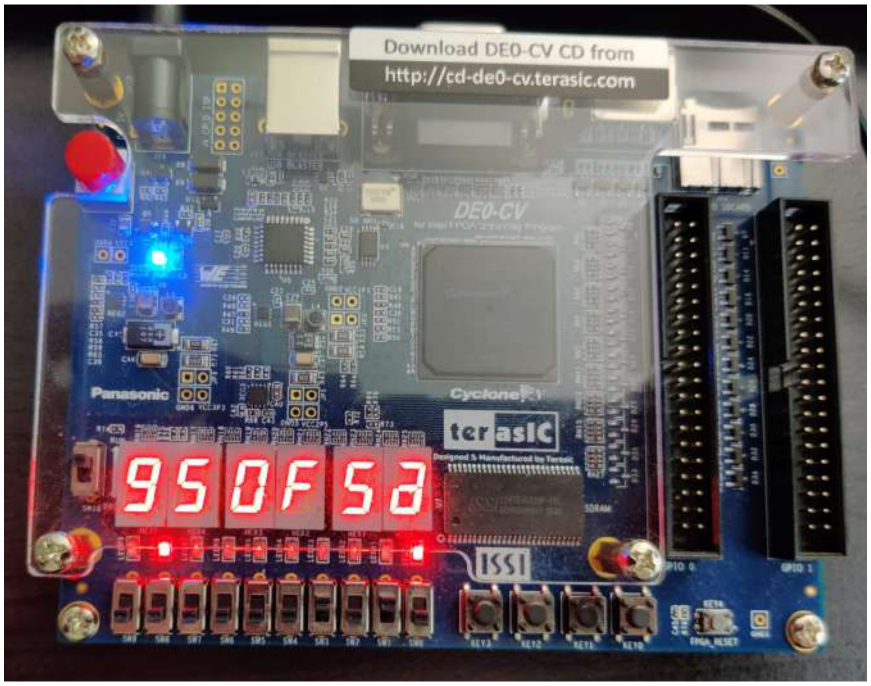


Figure 4.0.10: Player starts a new game

5.0. Video Demo

https://uofh-my.sharepoint.com/personal/rverma7_cougarnet_uh_edu/_layouts/15/stream.aspx?id=%2Fpersonal%2Frverma7%5Fcougarnet%5Fuh%5Fedu%2FDocuments%2FADD%20videos%2FLab4%5Fvideos%2FIMG%5F8805%2EMOV&referrer=StreamWebApp%2EWeb&referrerScenario=AddressBarCopied%2Eview

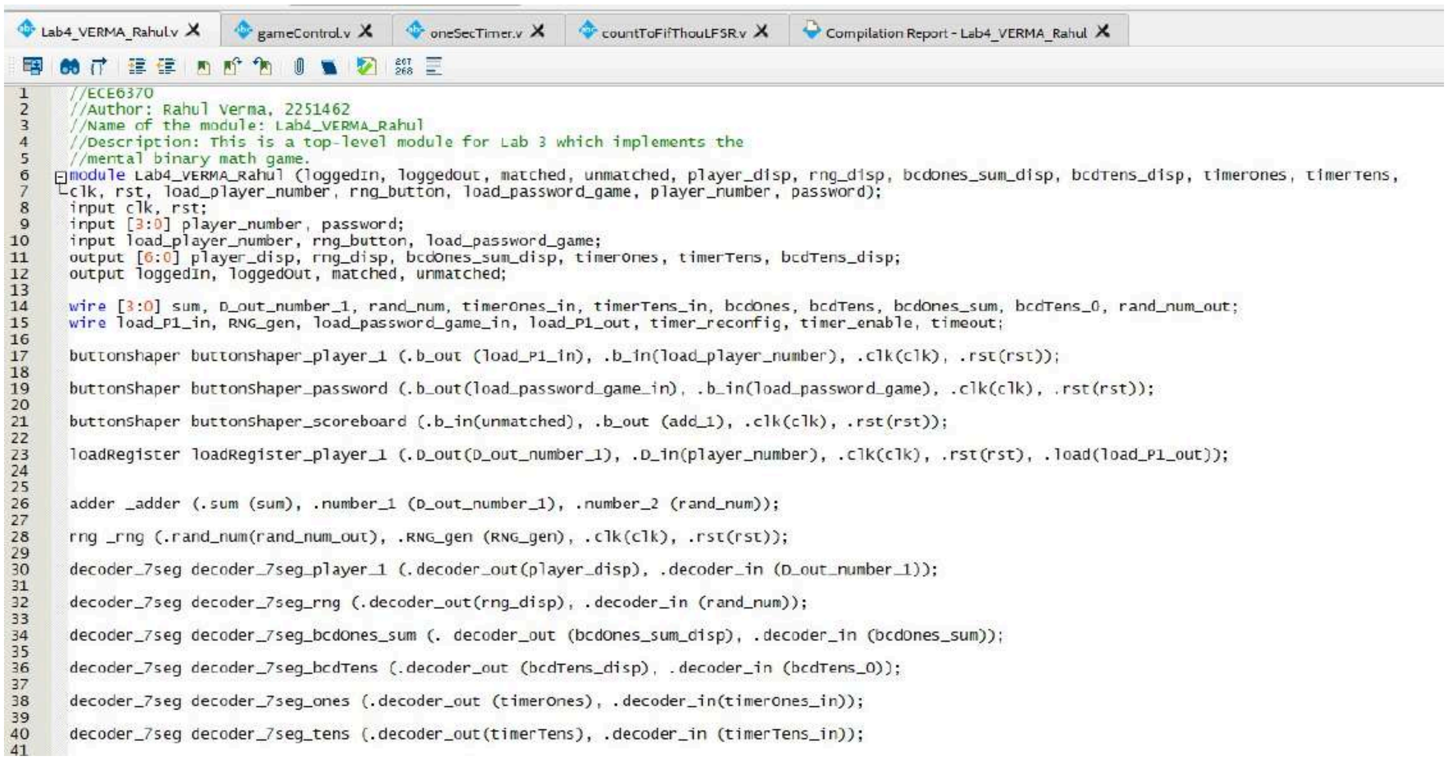
6.0. Conclusion

Mental binary math game was designed and tested with all the aforementioned features which also includes scorecard, interfacing with on-chip ROM and RAM for storing the password, implemented logout function and password reset function. Modified the 1ms Timer, replaced the conventional up-counter with more power efficient LFSR counter. The result of the tests conducted on the FPGA board is as follows-

- Player's default password stored in the ROM is accessible and if player resets their password, the new password is stored in the RAM is retrieved for logging in.
- The player can logout and log back in with either default password or with the updated password.
- 1 sec timer was designed which gives out a pulse every one second, with LFSR counter as its core for 1ms pulses.
- The player can start playing the game only after entering the correct password, until then the player's number or random number will not be displayed even if the load button or random number generator button is pressed.
- True random number generator is designed based on the duration of pressing the button.
- Two-digit timer was designed which decrements by 1 every 1 sec.
- The board keeps track of the number of correct attempts and displays the scorecard at the end of the game.

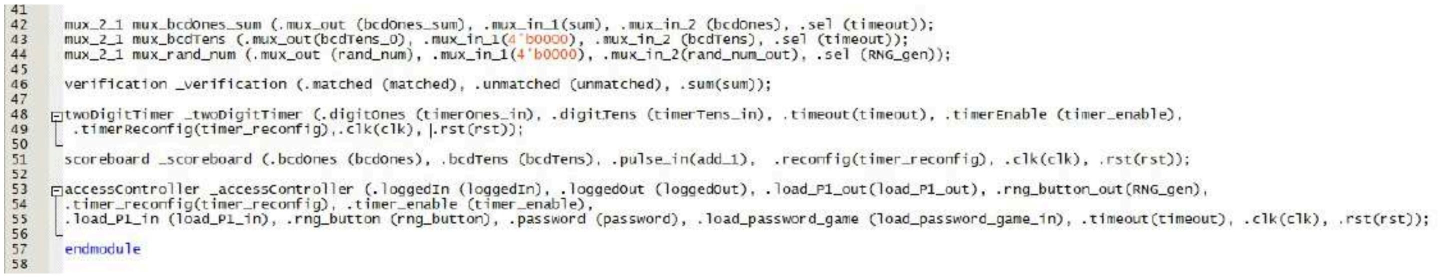
Overall, the Mental Binary Math Game is a success.

Appendix



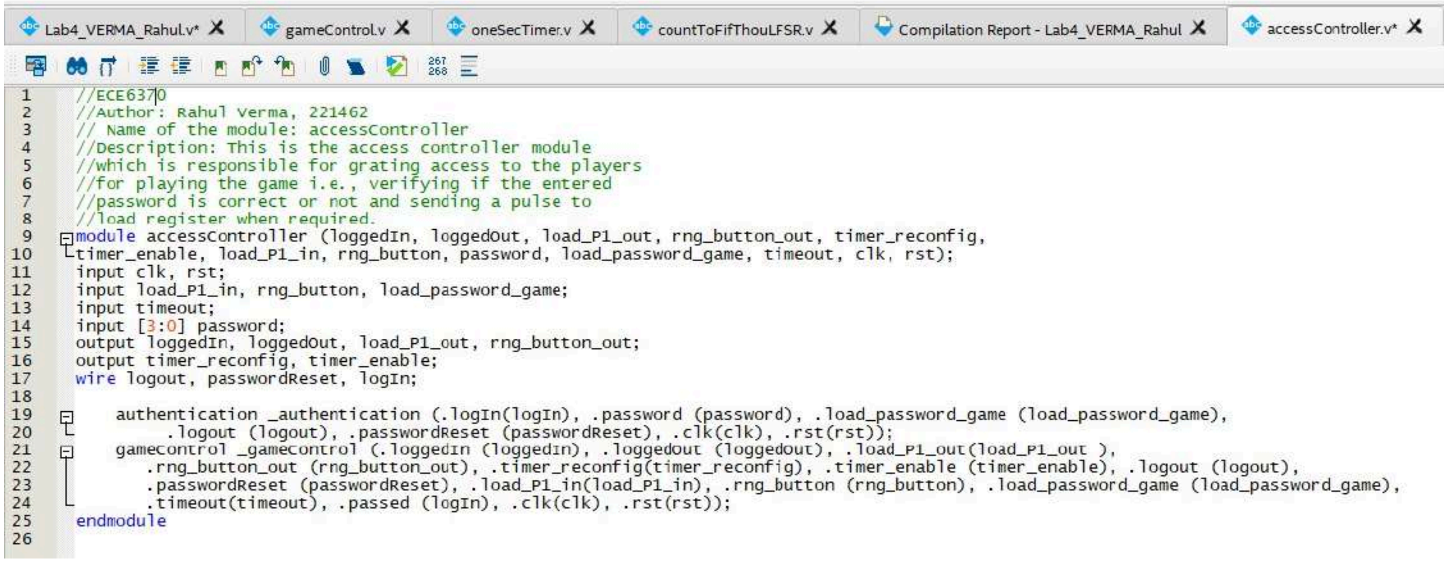
```
//ECE6370
//Author: Rahul Verma, 2251462
//Name of the module: Lab4_VERMA_Rahul
//Description: This is a top-level module for Lab 3 which implements the
//mental binary math game.
module Lab4_VERMA_Rahul (loggedIn, loggedOut, matched, unmatched, player_disp, rng_disp, bcdOnes_sum_disp, bcdTens_disp, timerOnes, timerTens,
clk, rst, load_player_number, rng_button, load_password_game, player_number, password);
 input clk, rst;
 input [3:0] player_number, password;
 input load_player_number, rng_button, load_password_game;
 output [6:0] player_disp, rng_disp, bcdOnes_sum_disp, timerOnes, timerTens, bcdTens_disp;
 output loggedIn, loggedOut, matched, unmatched;

 wire [3:0] sum, D_out_number_1, rand_num, timerOnes_in, timerTens_in, bcdOnes, bcdTens, bcdOnes_sum, bcdTens_0, rand_num_out;
 wire load_P1_in, RNG_gen, load_password_game_in, load_P1_out, timer_reconfig, timer_enable, timeout;

 buttonShaper buttonShaper_player_1 (.b_out (load_P1_in), .b_in(load_player_number), .clk(clk), .rst(rst));

 buttonShaper buttonShaper_password (.b_out(load_password_game_in), .b_in(load_password_game), .clk(clk), .rst(rst));

 buttonShaper buttonShaper_scoreboard (.b_in(unmatched), .b_out (add_1), .clk(clk), .rst(rst));

 loadRegister loadRegister_player_1 (.D_out(D_out_number_1), .D_in(player_number), .clk(clk), .rst(rst), .load(load_P1_out));


 adder _adder (.sum (sum), .number_1 (D_out_number_1), .number_2 (rand_num));

 rng _rng (.rand_num(rand_num_out), .RNG_gen (RNG_gen), .clk(clk), .rst(rst));

 decoder_7seg decoder_7seg_player_1 (.decoder_out(player_disp), .decoder_in (D_out_number_1));

 decoder_7seg decoder_7seg_rng (.decoder_out(rng_disp), .decoder_in (rand_num));

 decoder_7seg decoder_7seg_bcdOnes_sum (. decoder_out (bcdOnes_sum_disp), .decoder_in (bcdOnes_sum));

 decoder_7seg decoder_7seg_bcdTens (.decoder_out (bcdTens_disp), .decoder_in (bcdTens_0));

 decoder_7seg decoder_7seg_ones (.decoder_out (timerOnes), .decoder_in(timerOnes_in));

 decoder_7seg decoder_7seg_tens (.decoder_out(timerTens), .decoder_in (timerTens_in));

```

Figure 7.0.1: Top Module (1 of 2)



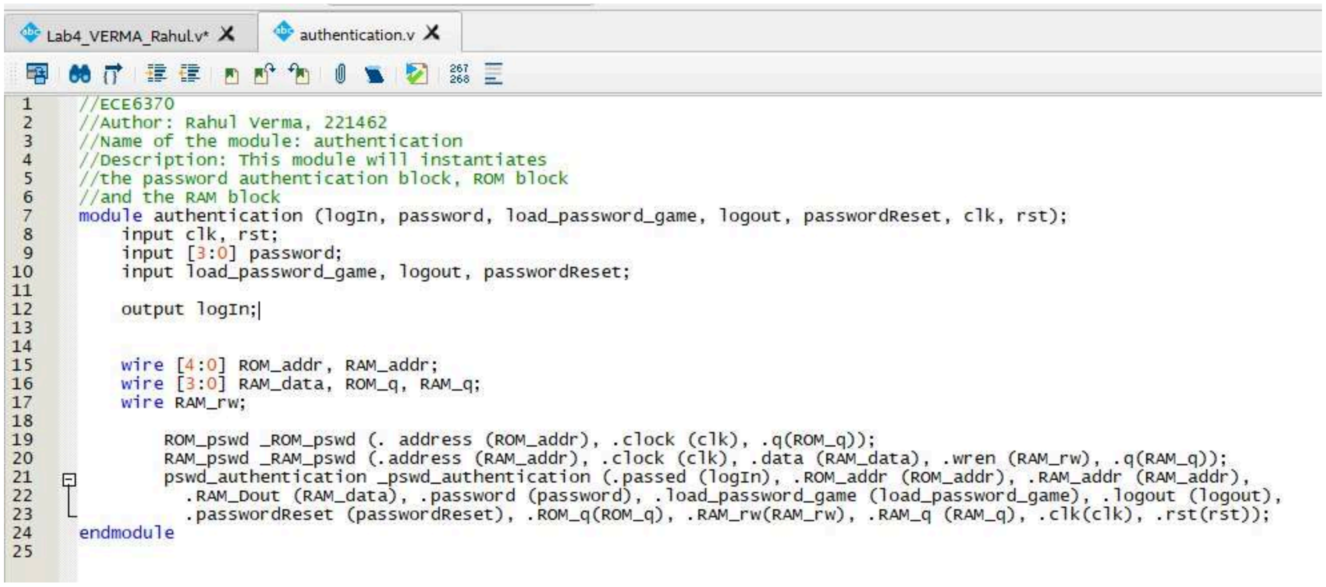
```

 mux_2_1 mux_bcdOnes_sum (.mux_out (bcdOnes_sum), .mux_in_1(sum), .mux_in_2 (bcdOnes), .sel (timeout));
 mux_2_1 mux_bcdTens (.mux_out(bcdTens_0), .mux_in_1(4'b0000), .mux_in_2 (bcdTens), .sel (timeout));
 mux_2_1 mux_rand_num (.mux_out (rand_num), .mux_in_1(4'b0000), .mux_in_2(rand_num_out), .sel (RNG_gen));

 verification _verification (.matched (matched), .unmatched (unmatched), .sum(sum));

twoDigitTimer _twoDigitTimer (.digitOnes (timerOnes_in), .digitTens (timerTens_in), .timeout(timeout), .timerEnable (timer_enable),
  .timerReconfig(timer_reconfig),.clk(clk), .rst(rst));

 scoreboard _scoreboard (.bcdOnes (bcdOnes), .bcdTens (bcdTens), .pulse_in(add_1),  .reconfig(timer_reconfig), .clk(clk), .rst(rst));

accessController _accessController (.loggedIn (loggedIn), .loggedOut (loggedOut), .load_P1_out(load_P1_out), .rng_button_out(RNG_gen),
 .timer_reconfig(timer_reconfig), .timer_enable (timer_enable),
 .load_P1_in (load_P1_in), .rng_button (rng_button), .password (password), .load_password_game (load_password_game_in), .timeout(timeout), .clk(clk), .rst(rst));

 endmodule

```

Figure 7.0.1: Top Module (2 of 2)



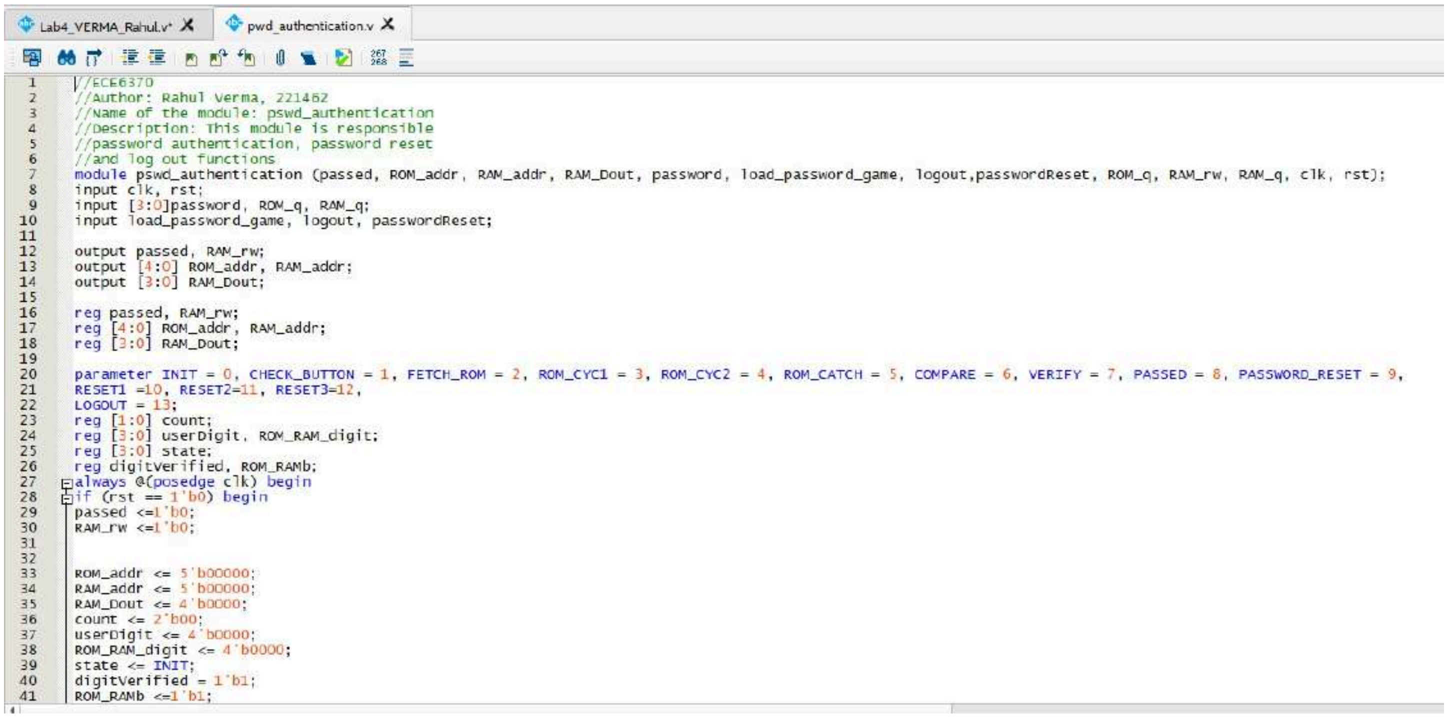
```
 //ECE6370
 //Author: Rahul Verma, 221462
 // Name of the module: accessController
 //Description: This is the access controller module
 //which is responsible for grating access to the players
 //for playing the game i.e., verifying if the entered
 //password is correct or not and sending a pulse to
 //load register when required.
module accessController (loggedIn, loggedOut, load_P1_out, rng_button_out, timer_reconfig,
timer_enable, load_P1_in, rng_button, password, load_password_game, timeout, clk, rst);
 input clk, rst;
 input load_P1_in, rng_button, load_password_game;
 input timeout;
 input [3:0] password;
 output loggedIn, loggedOut, load_P1_out, rng_button_out;
 output timer_reconfig, timer_enable;
 wire logout, passwordReset, logIn;

     authentication _authentication (.logIn(logIn), .password (password), .load_password_game (load_password_game),
             .logout (logout), .passwordReset (passwordReset), .clk(clk), .rst(rst));
     gameControl _gameControl (.loggedIn (loggedIn), .loggedOut (loggedOut), .load_P1_out(load_P1_out ),
        .rng_button_out (rng_button_out), .timer_reconfig(timer_reconfig), .timer_enable (timer_enable), .logout (logout),
        .passwordReset (passwordReset), .load_P1_in(load_P1_in), .rng_button (rng_button), .load_password_game (load_password_game),
        .timeout(timeout), .passed (logIn), .clk(clk), .rst(rst));
 endmodule

```

Figure 7.0.2: Access controller



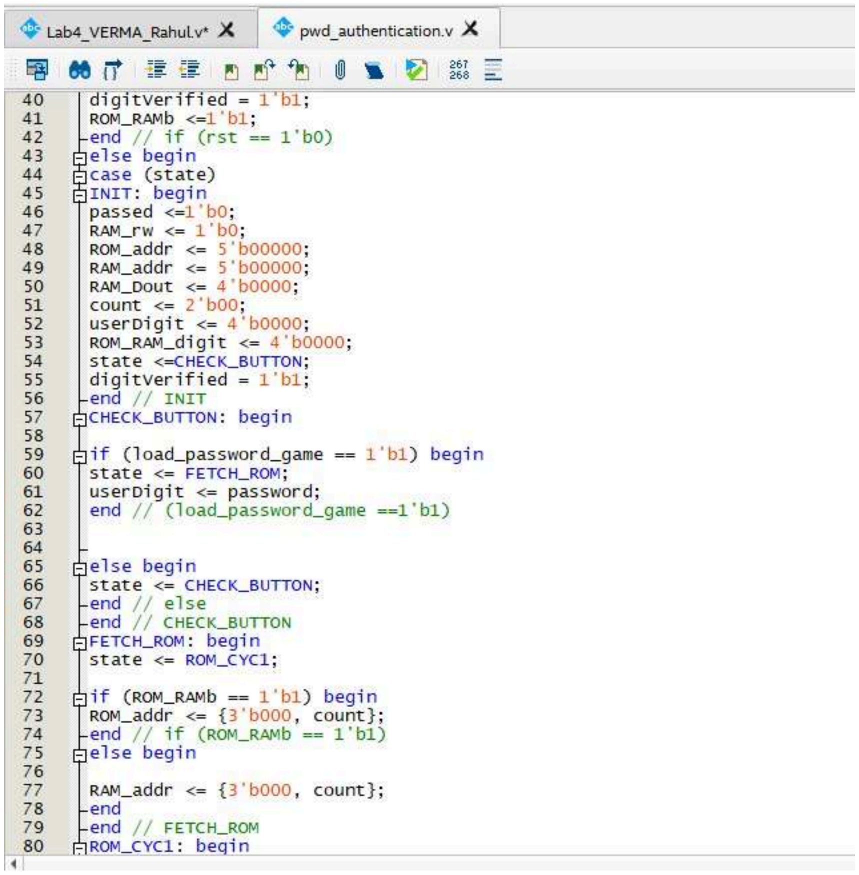
```verilog
//ECE6370
//Author: Rahul Verma, 221462
//Name of the module: authentication
//Description: This module will instantiates
//the password authentication block, ROM block
//and the RAM block
module authentication (logIn, password, load_password_game, logout, passwordReset, clk, rst);
    input clk, rst;
    input [3:0] password;
    input load_password_game, logout, passwordReset;

    output logIn;


    wire [4:0] ROM_addr, RAM_addr;
    wire [3:0] RAM_data, ROM_q, RAM_q;
    wire RAM_rw;

        ROM_pswd _ROM_pswd (. address (ROM_addr), .clock (clk), .q(ROM_q));
        RAM_pswd _RAM_pswd (.address (RAM_addr), .clock (clk), .data (RAM_data), .wren (RAM_rw), .q(RAM_q));
        pswd_authentication _pswd_authentication (.passed (logIn), .ROM_addr (ROM_addr), .RAM_addr (RAM_addr),
          .RAM_Dout (RAM_data), .password (password), .load_password_game (load_password_game), .logout (logout),
          .passwordReset (passwordReset), .ROM_q(ROM_q), .RAM_rw(RAM_rw), .RAM_q (RAM_q), .clk(clk), .rst(rst));
endmodule

```

Figure 7.0.3: Authentication



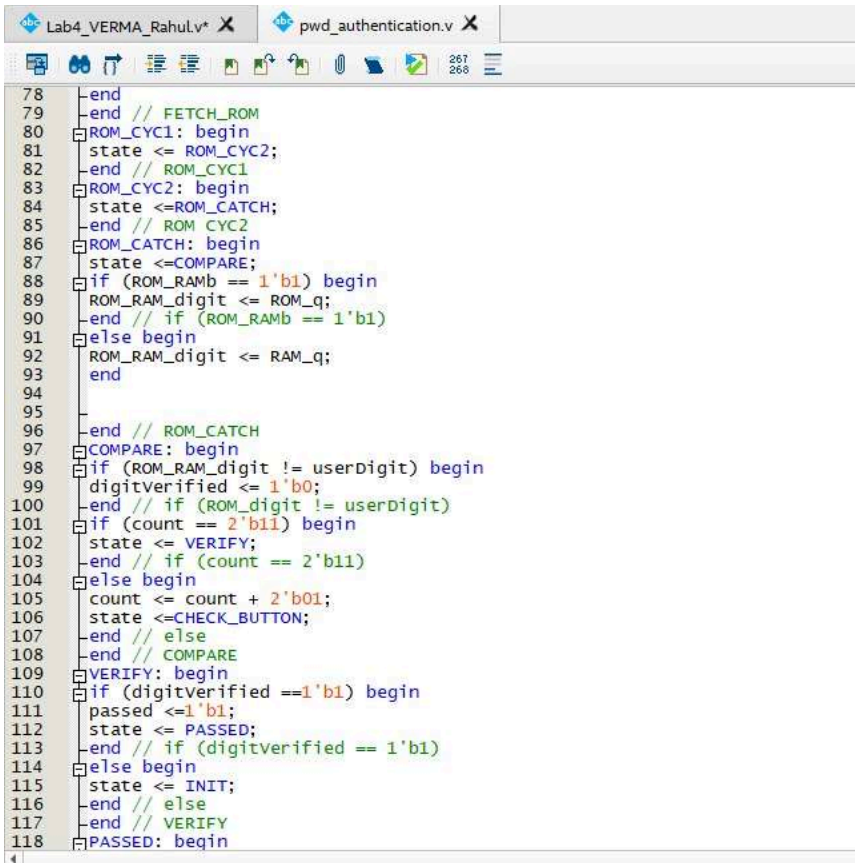
```verilog
//ECE6370
//Author: Rahul Verma, 221462
//Name of the module: pswd_authentication
//Description: This module is responsible
//password authentication, password reset
//and log out functions
module pswd_authentication (passed, ROM_addr, RAM_addr, RAM_Dout, password, load_password_game, logout,passwordReset, ROM_q, RAM_rw, RAM_q, clk, rst);
input clk, rst;
input [3:0]password, ROM_q, RAM_q;
input load_password_game, logout, passwordReset;

output passed, RAM_rw;
output [4:0] ROM_addr, RAM_addr;
output [3:0] RAM_Dout;

reg passed, RAM_rw;
reg [4:0] ROM_addr, RAM_addr;
reg [3:0] RAM_Dout;

parameter INIT = 0, CHECK_BUTTON = 1, FETCH_ROM = 2, ROM_CYC1 = 3, ROM_CYC2 = 4, ROM_CATCH = 5, COMPARE = 6, VERIFY = 7, PASSED = 8, PASSWORD_RESET = 9,
RESET1 =10, RESET2=11, RESET3=12,
LOGOUT = 13;
reg [1:0] count;
reg [3:0] userDigit, ROM_RAM_digit;
reg [3:0] state;
reg digitVerified, ROM_RAMb;
always @(posedge clk) begin
if (rst == 1'b0) begin
passed <=1'b0;
RAM_rw <=1'b0;


ROM_addr <= 5'b00000;
RAM_addr <= 5'b00000;
RAM_Dout <= 4'b0000;
count <= 2'b00;
userDigit <= 4'b0000;
ROM_RAM_digit <= 4'b0000;
state <= INIT;
digitVerified = 1'b1;
ROM_RAMb <=1'b1;
```

Figure 7.0.4: Password Authentication (1 of 7)



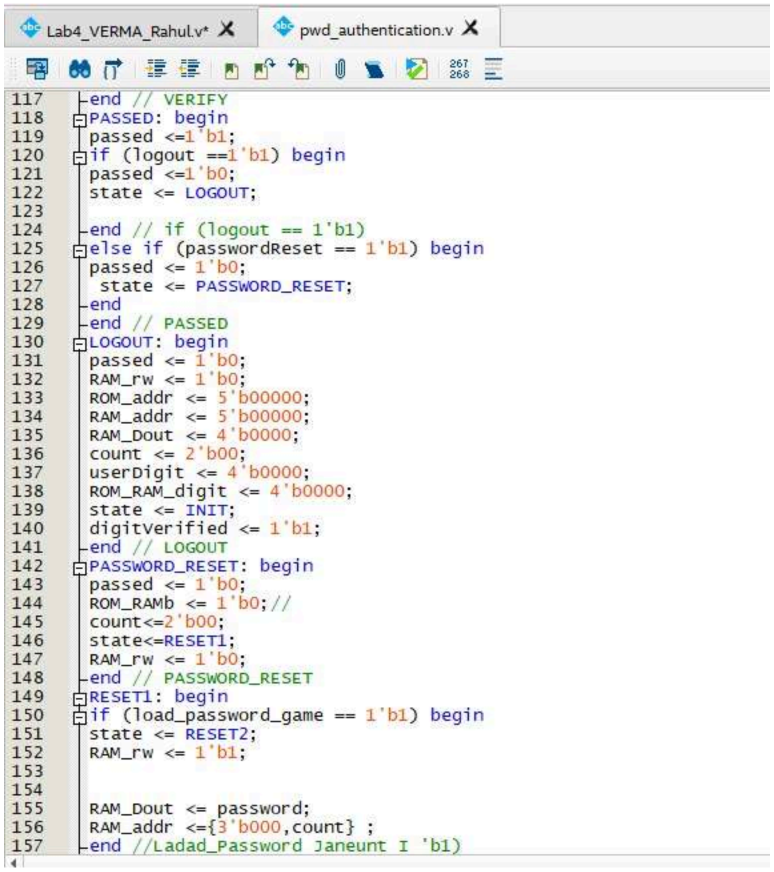
```
digitVerified = 1'b1;
ROM_RAMb <=1'b1;
end // if (rst == 1'b0)
else begin
case (state)
INIT: begin
passed <=1'b0;
RAM_rw <= 1'b0;
ROM_addr <= 5'b00000;
RAM_addr <= 5'b00000;
RAM_Dout <= 4'b0000;
count <= 2'b00;
userDigit <= 4'b0000;
ROM_RAM_digit <= 4'b0000;
state <=CHECK_BUTTON;
digitVerified = 1'b1;
end // INIT
CHECK_BUTTON: begin

if (load_password_game == 1'b1) begin
state <= FETCH_ROM;
userDigit <= password;
end // (load_password_game ==1'b1)


else begin
state <= CHECK_BUTTON;
end // else
end // CHECK_BUTTON
FETCH_ROM: begin
state <= ROM_CYC1;

if (ROM_RAMb == 1'b1) begin
ROM_addr <= {3'b000, count};
end // if (ROM_RAMb == 1'b1)
else begin

RAM_addr <= {3'b000, count};
end
end // FETCH_ROM
ROM_CYC1: begin
```

Figure 7.0.5: Password Authentication (2 of 7)



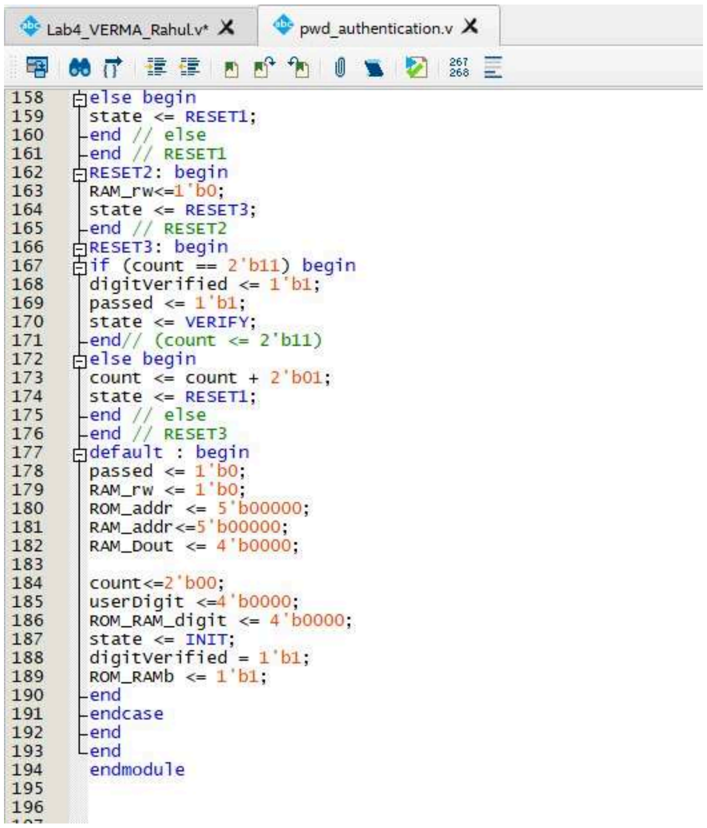
```
end
end // FETCH_ROM
ROM_CYC1: begin
state <= ROM_CYC2;
end // ROM_CYC1
ROM_CYC2: begin
state <=ROM_CATCH;
end // ROM CYC2
ROM_CATCH: begin
state <=COMPARE;
if (ROM_RAMb == 1'b1) begin
ROM_RAM_digit <= ROM_q;
end // if (ROM_RAMb == 1'b1)
else begin
ROM_RAM_digit <= RAM_q;
end


end // ROM_CATCH
COMPARE: begin
if (ROM_RAM_digit != userDigit) begin
digitVerified <= 1'b0;
end // if (ROM_digit != userDigit)
if (count == 2'b11) begin
state <= VERIFY;
end // if (count == 2'b11)
else begin
count <= count + 2'b01;
state <=CHECK_BUTTON;
end // else
end // COMPARE
VERIFY: begin
if (digitVerified ==1'b1) begin
passed <=1'b1;
state <= PASSED;
end // if (digitVerified == 1'b1)
else begin
state <= INIT;
end // else
end // VERIFY
PASSED: begin
```

Figure 7.0.6: Password Authentication (3 of 7)



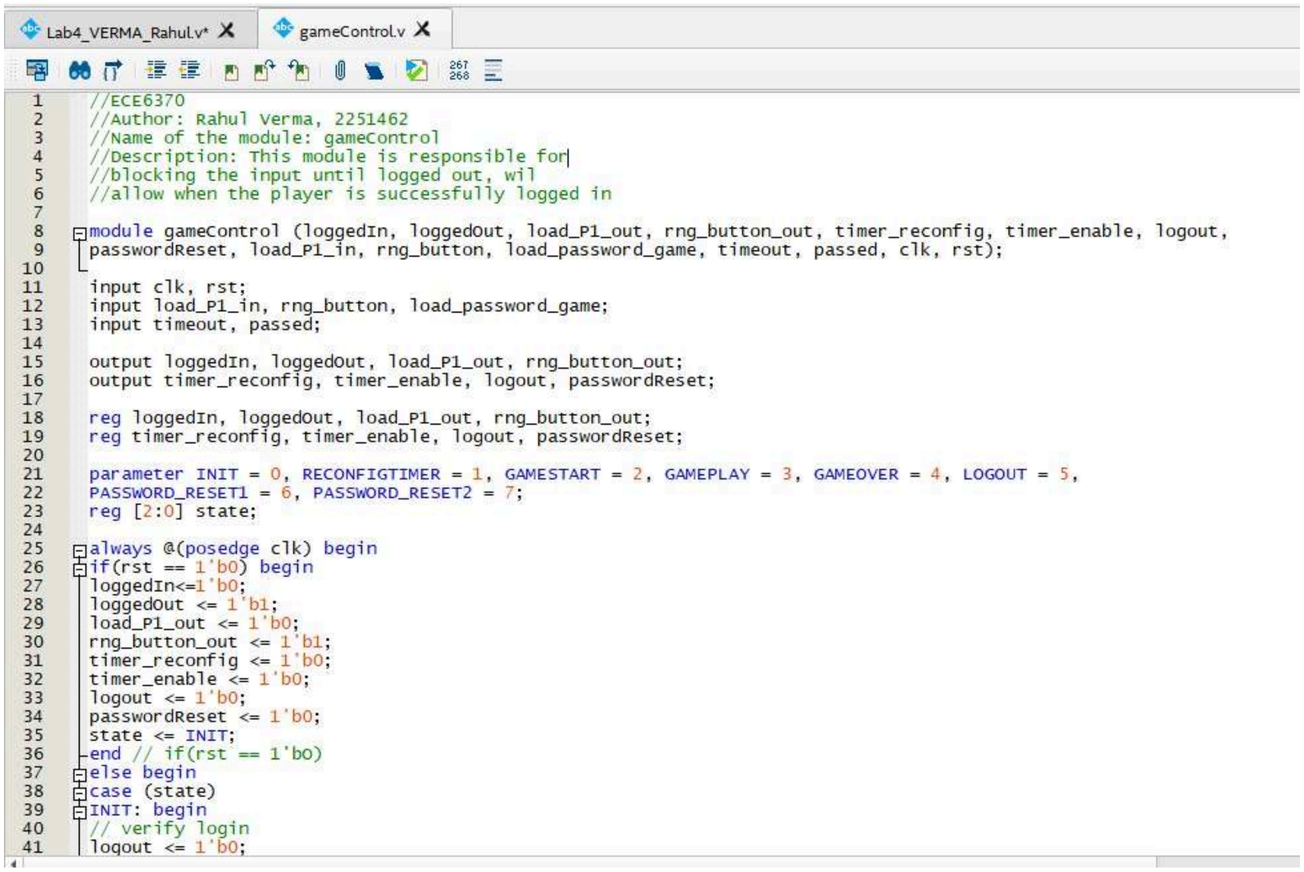
```verilog
end // VERIFY
PASSED: begin
passed <=1'b1;
if (logout ==1'b1) begin
passed <=1'b0;
state <= LOGOUT;

end // if (logout == 1'b1)
else if (passwordReset == 1'b1) begin
passed <= 1'b0;
 state <= PASSWORD_RESET;
end
end // PASSED
LOGOUT: begin
passed <= 1'b0;
RAM_rw <= 1'b0;
ROM_addr <= 5'b00000;
RAM_addr <= 5'b00000;
RAM_Dout <= 4'b0000;
count <= 2'b00;
userDigit <= 4'b0000;
ROM_RAM_digit <= 4'b0000;
state <= INIT;
digitVerified <= 1'b1;
end // LOGOUT
PASSWORD_RESET: begin
passed <= 1'b0;
ROM_RAMb <= 1'b0;//
count<=2'b00;
state<=RESET1;
RAM_rw <= 1'b0;
end // PASSWORD_RESET
RESET1: begin
if (load_password_game == 1'b1) begin
state <= RESET2;
RAM_rw <= 1'b1;


RAM_Dout <= password;
RAM_addr <={3'b000,count} ;
end //Ladad_Password Janeunt I 'b1)
```

Figure 7.0.7: Password Authentication (4 of 7)



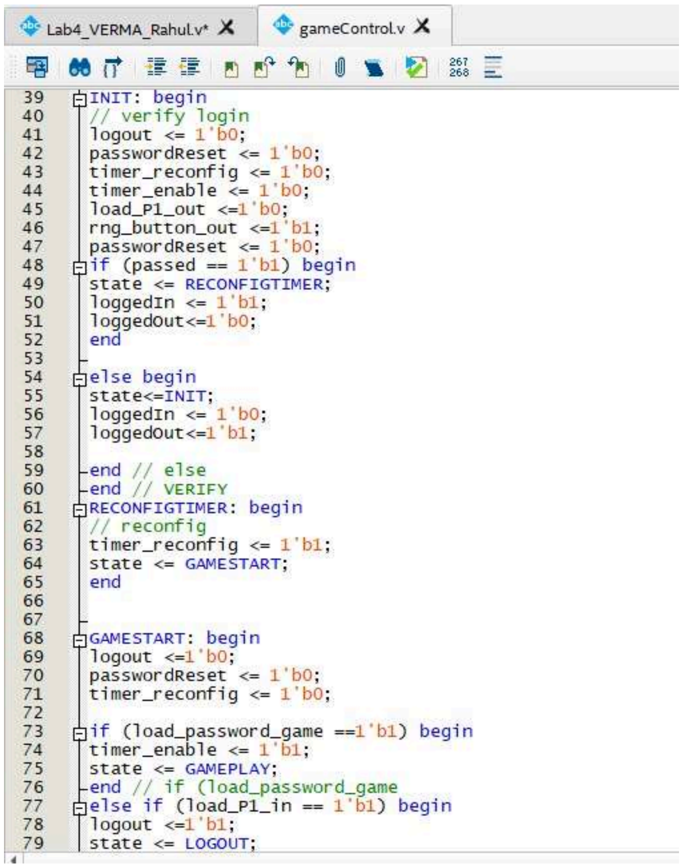
```verilog
else begin
state <= RESET1;
end // else
end // RESET1
RESET2: begin
RAM_rw<=1'b0;
state <= RESET3;
end // RESET2
RESET3: begin
if (count == 2'b11) begin
digitVerified <= 1'b1;
passed <= 1'b1;
state <= VERIFY;
end// (count <= 2'b11)
else begin
count <= count + 2'b01;
state <= RESET1;
end // else
end // RESET3
default : begin
passed <= 1'b0;
RAM_rw <= 1'b0;
ROM_addr <= 5'b00000;
RAM_addr<=5'b00000;
RAM_Dout <= 4'b0000;

count<=2'b00;
userDigit <=4'b0000;
ROM_RAM_digit <= 4'b0000;
state <= INIT;
digitVerified = 1'b1;
ROM_RAMb <= 1'b1;
end
endcase
end
end
endmodule
```

Figure 7.0.8: Password Authentication (5 of 7)



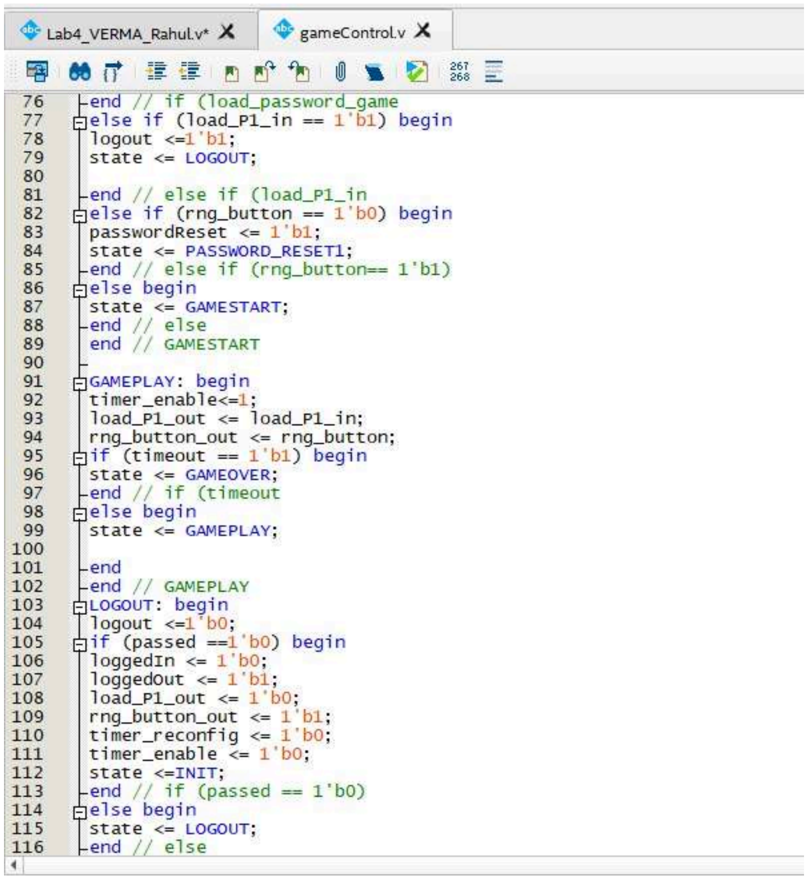
```
//ECE6370
//Author: Rahul Verma, 2251462
//Name of the module: gameControl
//Description: This module is responsible for
//blocking the input until logged out, wil
//allow when the player is successfully logged in

module gameControl (loggedIn, loggedOut, load_P1_out, rng_button_out, timer_reconfig, timer_enable, logout,
passwordReset, load_P1_in, rng_button, load_password_game, timeout, passed, clk, rst);

input clk, rst;
input load_P1_in, rng_button, load_password_game;
input timeout, passed;

output loggedIn, loggedOut, load_P1_out, rng_button_out;
output timer_reconfig, timer_enable, logout, passwordReset;

reg loggedIn, loggedOut, load_P1_out, rng_button_out;
reg timer_reconfig, timer_enable, logout, passwordReset;

parameter INIT = 0, RECONFIGTIMER = 1, GAMESTART = 2, GAMEPLAY = 3, GAMEOVER = 4, LOGOUT = 5,
PASSWORD_RESET1 = 6, PASSWORD_RESET2 = 7;
reg [2:0] state;

always @(posedge clk) begin
if(rst == 1'b0) begin
loggedIn<=1'b0;
loggedOut <= 1'b1;
load_P1_out <= 1'b0;
rng_button_out <= 1'b1;
timer_reconfig <= 1'b0;
timer_enable <= 1'b0;
logout <= 1'b0;
passwordReset <= 1'b0;
state <= INIT;
end // if(rst == 1'bo)
else begin
case (state)
INIT: begin
// verify login
logout <= 1'b0;
```

Figure 7.0.11: Game Control (1 of 6)



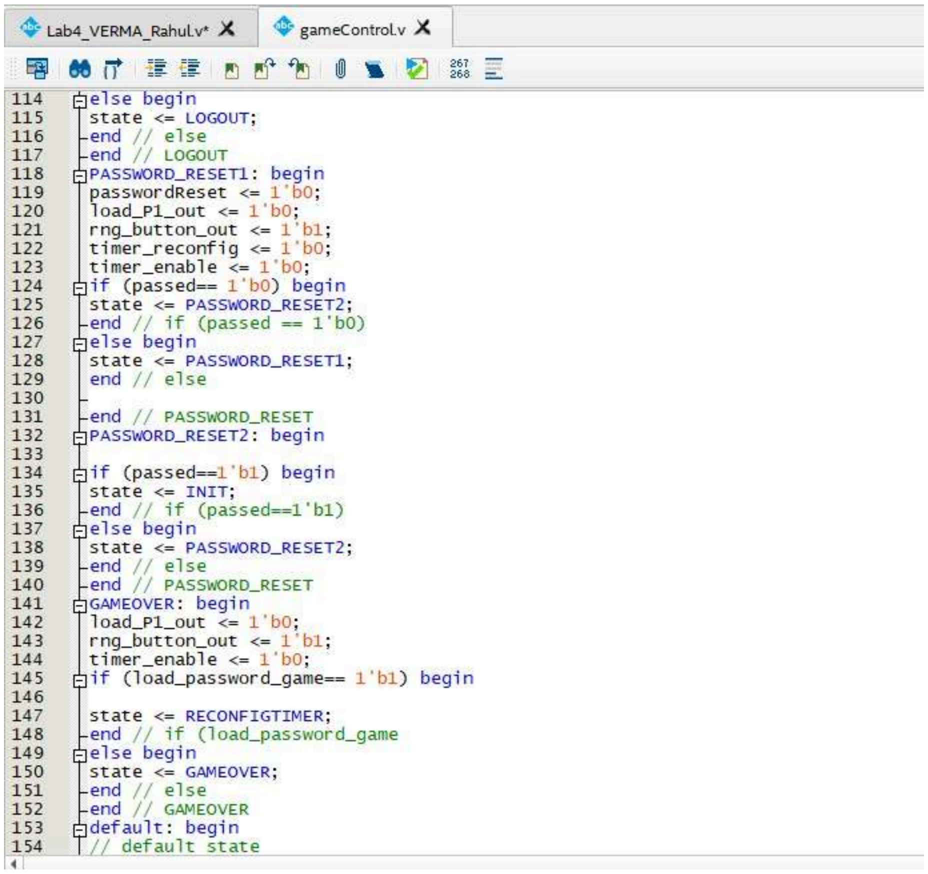
```
INIT: begin
// verify login
logout <= 1'b0;
passwordReset <= 1'b0;
timer_reconfig <= 1'b0;
timer_enable <= 1'b0;
load_P1_out <=1'b0;
rng_button_out <=1'b1;
passwordReset <= 1'b0;
if (passed == 1'b1) begin
state <= RECONFIGTIMER;
loggedIn <= 1'b1;
loggedOut<=1'b0;
end

else begin
state<=INIT;
loggedIn <= 1'b0;
loggedOut<=1'b1;

end // else
end // VERIFY
RECONFIGTIMER: begin
// reconfig
timer_reconfig <= 1'b1;
state <= GAMESTART;
end


GAMESTART: begin
logout <=1'b0;
passwordReset <= 1'b0;
timer_reconfig <= 1'b0;

if (load_password_game ==1'b1) begin
timer_enable <= 1'b1;
state <= GAMEPLAY;
end // if (load_password_game
else if (load_P1_in == 1'b1) begin
logout <=1'b1;
state <= LOGOUT;
```

Figure 7.0.12: Game Control (2 of 6)



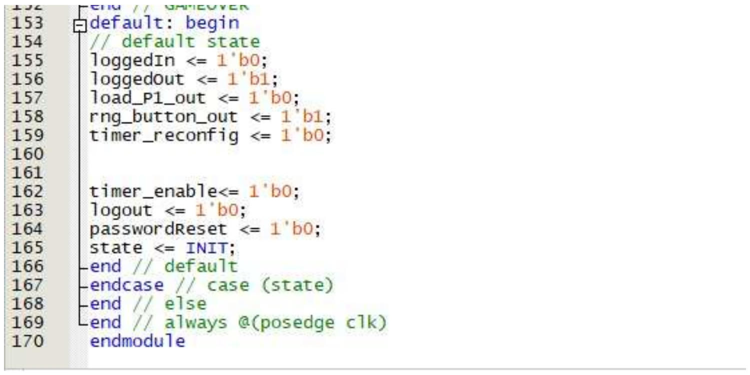
```
end // if (load_password_game
else if (load_P1_in == 1'b1) begin
logout <=1'b1;
state <= LOGOUT;

end // else if (load_P1_in
else if (rng_button == 1'b0) begin
passwordReset <= 1'b1;
state <= PASSWORD_RESET1;
end // else if (rng_button== 1'b1)
else begin
state <= GAMESTART;
end // else
end // GAMESTART

GAMEPLAY: begin
timer_enable<=1;
load_P1_out <= load_P1_in;
rng_button_out <= rng_button;
if (timeout == 1'b1) begin
state <= GAMEOVER;
end // if (timeout
else begin
state <= GAMEPLAY;

end
end // GAMEPLAY
LOGOUT: begin
logout <=1'b0;
if (passed ==1'b0) begin
loggedIn <= 1'b0;
loggedOut <= 1'b1;
load_P1_out <= 1'b0;
rng_button_out <= 1'b1;
timer_reconfig <= 1'b0;
timer_enable <= 1'b0;
state <=INIT;
end // if (passed == 1'b0)
else begin
state <= LOGOUT;
end // else
```

Figure 7.0.13: Game Control (3 of 6)



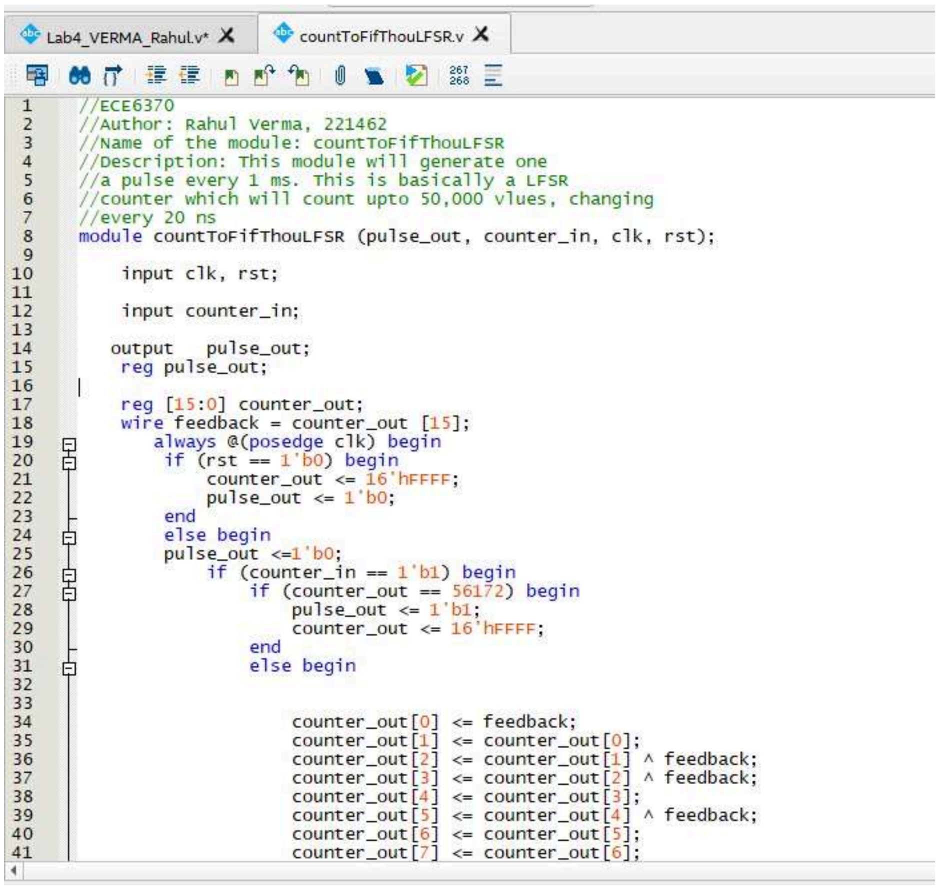
```
else begin
state <= LOGOUT;
end // else
end // LOGOUT
PASSWORD_RESET1: begin
passwordReset <= 1'b0;
load_P1_out <= 1'b0;
rng_button_out <= 1'b1;
timer_reconfig <= 1'b0;
timer_enable <= 1'b0;
if (passed== 1'b0) begin
state <= PASSWORD_RESET2;
end // if (passed == 1'b0)
else begin
state <= PASSWORD_RESET1;
end // else

end // PASSWORD_RESET
PASSWORD_RESET2: begin

if (passed==1'b1) begin
state <= INIT;
end // if (passed==1'b1)
else begin
state <= PASSWORD_RESET2;
end // else
end // PASSWORD_RESET
GAMEOVER: begin
load_P1_out <= 1'b0;
rng_button_out <= 1'b1;
timer_enable <= 1'b0;
if (load_password_game== 1'b1) begin

state <= RECONFIGTIMER;
end // if (load_password_game
else begin
state <= GAMEOVER;
end // else
end // GAMEOVER
default: begin
// default state
```

Figure 7.0.14: Game Control (4 of 6)



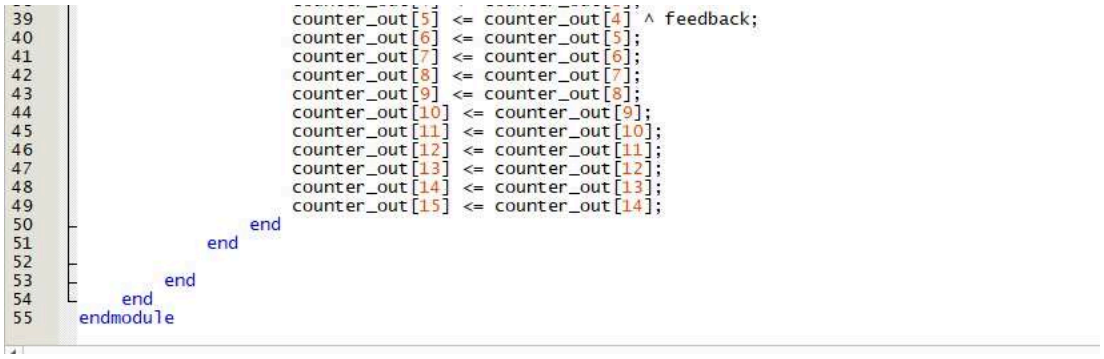
```
default: begin
// default state
loggedIn <= 1'b0;
loggedOut <= 1'b1;
load_P1_out <= 1'b0;
rng_button_out <= 1'b1;
timer_reconfig <= 1'b0;


timer_enable<= 1'b0;
logout <= 1'b0;
passwordReset <= 1'b0;
state <= INIT;
end // default
endcase // case (state)
end // else
end // always @(posedge clk)
endmodule
```

Figure 7.0.15: Game Control (5 of 6)



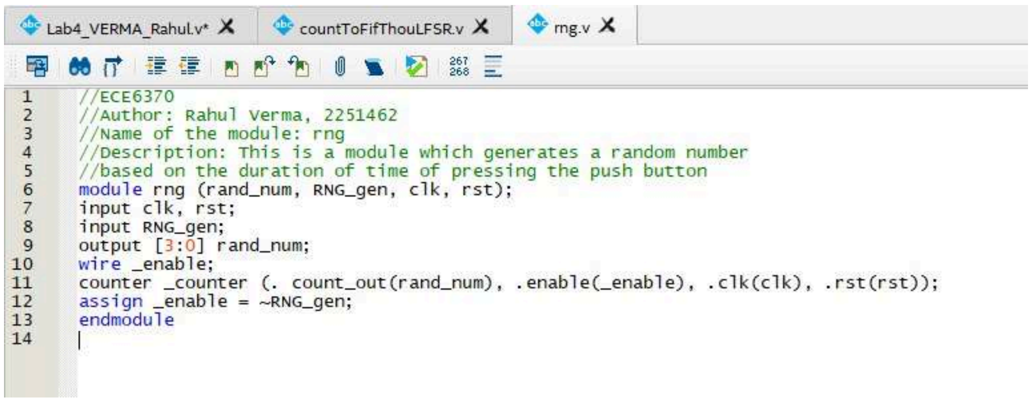
```
//ECE6370
//Author: Rahul Verma, 221462
//Name of the module: countToFifThouLFSR
//Description: This module will generate one
//a pulse every 1 ms. This is basically a LFSR
//counter which will count upto 50,000 vlues, changing
//every 20 ns
module countToFifThouLFSR (pulse_out, counter_in, clk, rst);

    input clk, rst;

    input counter_in;

   output    pulse_out;
    reg pulse_out;

    reg [15:0] counter_out;
    wire feedback = counter_out [15];
       always @(posedge clk) begin
        if (rst == 1'b0) begin
            counter_out <= 16'hFFFF;
            pulse_out <= 1'b0;
        end
        else begin
        pulse_out <=1'b0;
            if (counter_in == 1'b1) begin
                if (counter_out == 56172) begin
                    pulse_out <= 1'b1;
                    counter_out <= 16'hFFFF;
                end
                else begin


                    counter_out[0] <= feedback;
                    counter_out[1] <= counter_out[0];
                    counter_out[2] <= counter_out[1] ^ feedback;
                    counter_out[3] <= counter_out[2] ^ feedback;
                    counter_out[4] <= counter_out[3];
                    counter_out[5] <= counter_out[4] ^ feedback;
                    counter_out[6] <= counter_out[5];
                    counter_out[7] <= counter_out[6];
```

Figure 7.0.23: 50,000 LFSR counter (1 of 2)



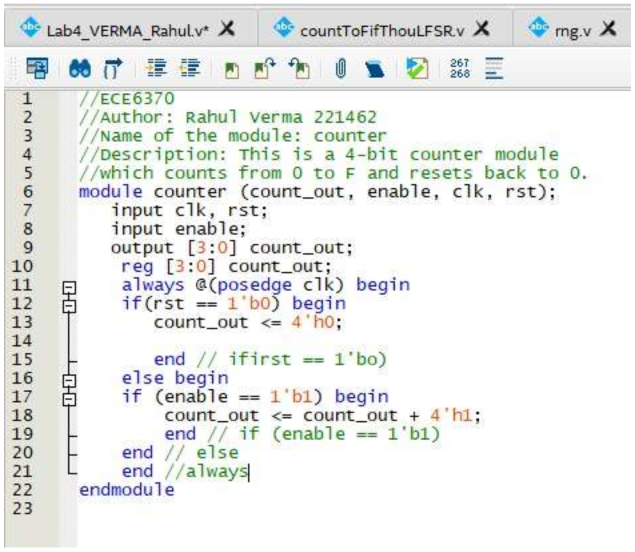
```
                    counter_out[5] <= counter_out[4] ^ feedback;
                    counter_out[6] <= counter_out[5];
                    counter_out[7] <= counter_out[6];
                    counter_out[8] <= counter_out[7];
                    counter_out[9] <= counter_out[8];
                    counter_out[10] <= counter_out[9];
                    counter_out[11] <= counter_out[10];
                    counter_out[12] <= counter_out[11];
                    counter_out[13] <= counter_out[12];
                    counter_out[14] <= counter_out[13];
                    counter_out[15] <= counter_out[14];
                end
            end

        end
    end
endmodule
```

Figure 7.0.24: 50,000 LFSR counter (2 of 2)



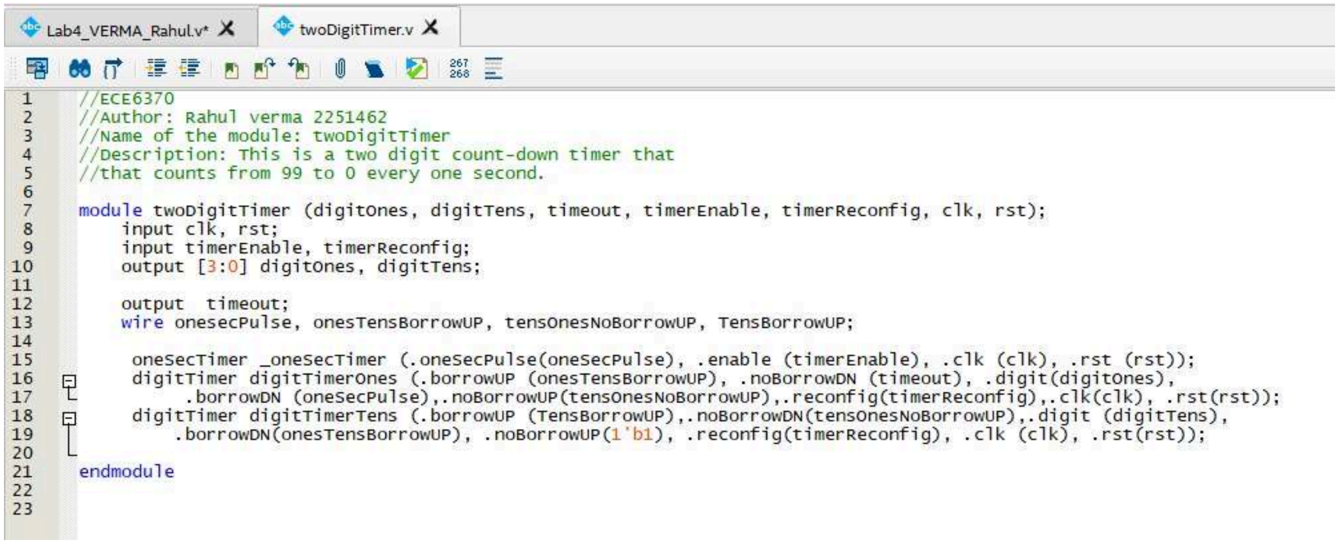
```
//ECE6370
//Author: Rahul Verma, 2251462
//Name of the module: rng
//Description: This is a module which generates a random number
//based on the duration of time of pressing the push button
module rng (rand_num, RNG_gen, clk, rst);
input clk, rst;
input RNG_gen;
output [3:0] rand_num;
wire _enable;
counter _counter (. count_out(rand_num), .enable(_enable), .clk(clk), .rst(rst));
assign _enable = ~RNG_gen;
endmodule
```

Figure 7.0.26: Random number generator



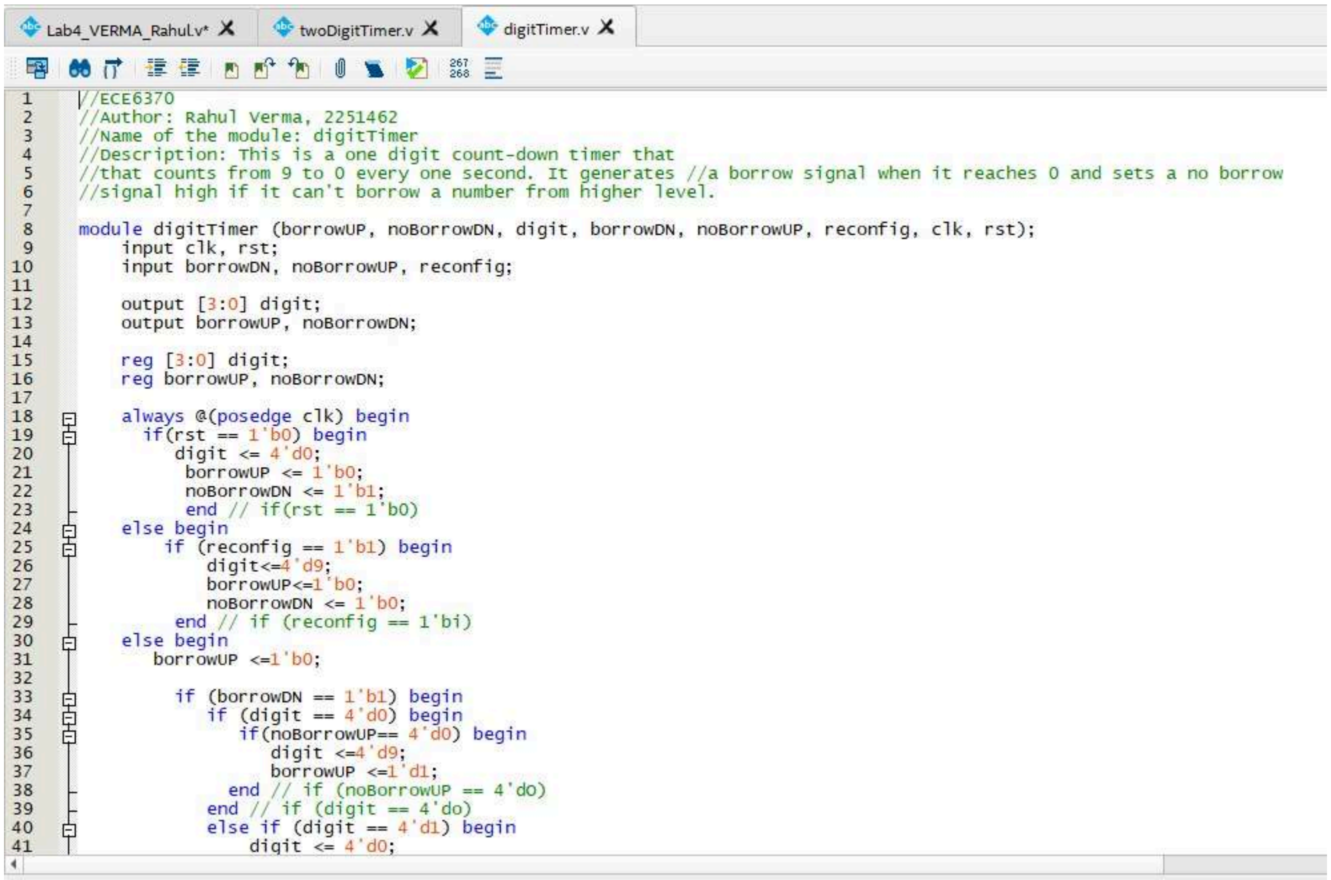
```
//ECE6370
//Author: Rahul Verma 221462
//Name of the module: counter
//Description: This is a 4-bit counter module
//which counts from 0 to F and resets back to 0.
module counter (count_out, enable, clk, rst);
   input clk, rst;
   input enable;
   output [3:0] count_out;
    reg [3:0] count_out;
    always @(posedge clk) begin
    if(rst == 1'b0) begin
       count_out <= 4'h0;

       end // ifirst == 1'bo)
    else begin
    if (enable == 1'b1) begin
        count_out <= count_out + 4'h1;
        end // if (enable == 1'b1)
    end // else
    end //always
endmodule
```

Figure 7.0.28: 4-bit Counter



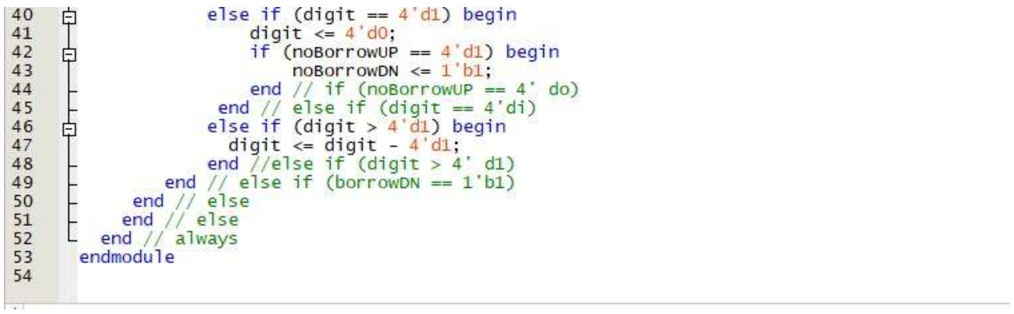
```
//ECE6370
//Author: Rahul verma 2251462
//Name of the module: twoDigitTimer
//Description: This is a two digit count-down timer that
//that counts from 99 to 0 every one second.

module twoDigitTimer (digitOnes, digitTens, timeout, timerEnable, timerReconfig, clk, rst);
    input clk, rst;
    input timerEnable, timerReconfig;
    output [3:0] digitOnes, digitTens;

    output  timeout;
    wire onesecPulse, onesTensBorrowUP, tensOnesNoBorrowUP, TensBorrowUP;

     oneSecTimer _oneSecTimer (.oneSecPulse(oneSecPulse), .enable (timerEnable), .clk (clk), .rst (rst));
     digitTimer digitTimerOnes (.borrowUP (onesTensBorrowUP), .noBorrowDN (timeout), .digit(digitOnes),
          .borrowDN (oneSecPulse),.noBorrowUP(tensOnesNoBorrowUP),.reconfig(timerReconfig),.clk(clk), .rst(rst));
     digitTimer digitTimerTens (.borrowUP (TensBorrowUP),.noBorrowDN(tensOnesNoBorrowUP),.digit (digitTens),
         .borrowDN(onesTensBorrowUP), .noBorrowUP(1'b1), .reconfig(timerReconfig), .clk (clk), .rst(rst));

endmodule
```

Figure 7.0.29: Two-digit timer



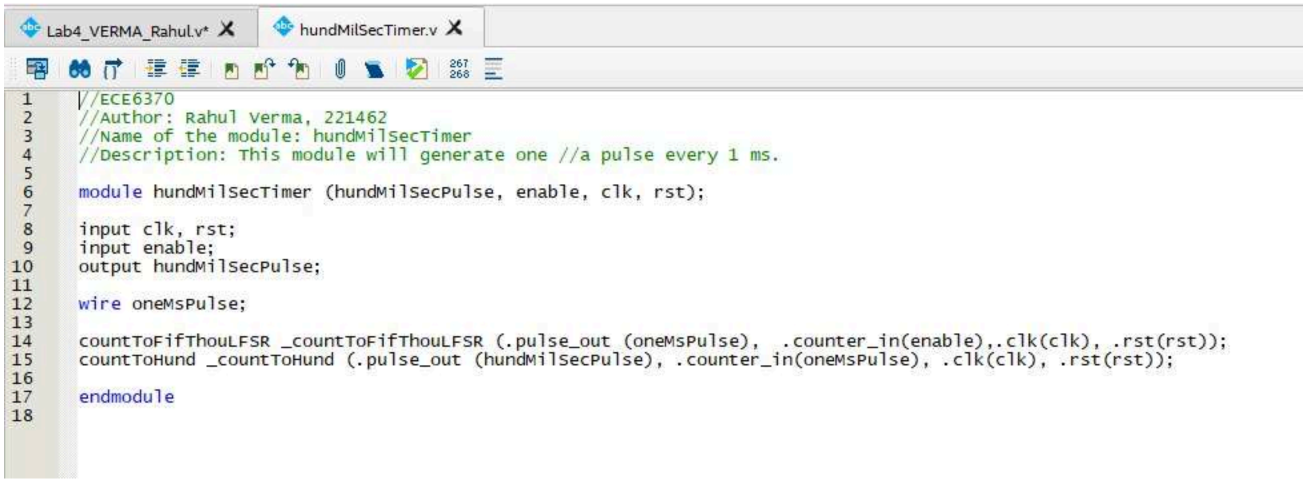
```
//ECE6370
//Author: Rahul Verma, 2251462
//Name of the module: digitTimer
//Description: This is a one digit count-down timer that
//that counts from 9 to 0 every one second. It generates //a borrow signal when it reaches 0 and sets a no borrow
//signal high if it can't borrow a number from higher level.

module digitTimer (borrowUP, noBorrowDN, digit, borrowDN, noBorrowUP, reconfig, clk, rst);
    input clk, rst;
    input borrowDN, noBorrowUP, reconfig;

    output [3:0] digit;
    output borrowUP, noBorrowDN;

    reg [3:0] digit;
    reg borrowUP, noBorrowDN;

    always @(posedge clk) begin
      if(rst == 1'b0) begin
         digit <= 4'd0;
          borrowUP <= 1'b0;
          noBorrowDN <= 1'b1;
          end // if(rst == 1'b0)
    else begin
        if (reconfig == 1'b1) begin
            digit<=4'd9;
            borrowUP<=1'b0;
            noBorrowDN <= 1'b0;
         end // if (reconfig == 1'bi)
    else begin
       borrowUP <=1'b0;

         if (borrowDN == 1'b1) begin
            if (digit == 4'd0) begin
               if(noBorrowUP== 4'd0) begin
                  digit <=4'd9;
                  borrowUP <=1'd1;
              end // if (noBorrowUP == 4'do)
            end // if (digit == 4'do)
            else if (digit == 4'd1) begin
               digit <= 4'd0;
```

Figure 7.0.31: Digit timer (1 of 2)



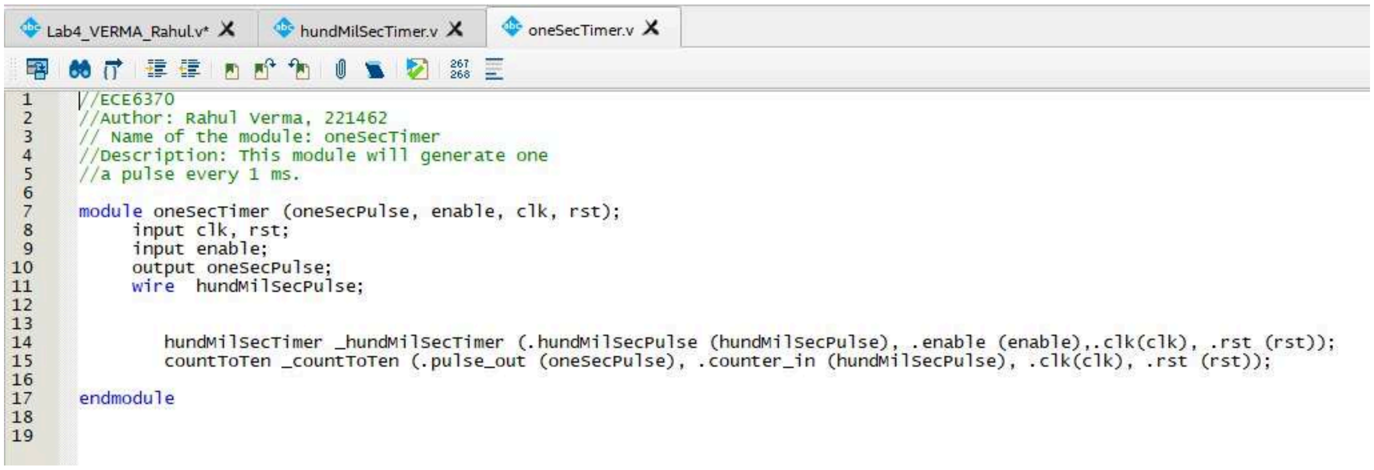
```
            else if (digit == 4'd1) begin
               digit <= 4'd0;
               if (noBorrowUP == 4'd1) begin
                   noBorrowDN <= 1'b1;
               end // if (noBorrowUP == 4' do)
             end // else if (digit == 4'di)
            else if (digit > 4'd1) begin
              digit <= digit - 4'd1;
            end //else if (digit > 4' d1)
         end // else if (borrowDN == 1'b1)
      end // else
     end // else
   end // always
endmodule
```

Figure 7.0.32: Digit timer (2 of 2)



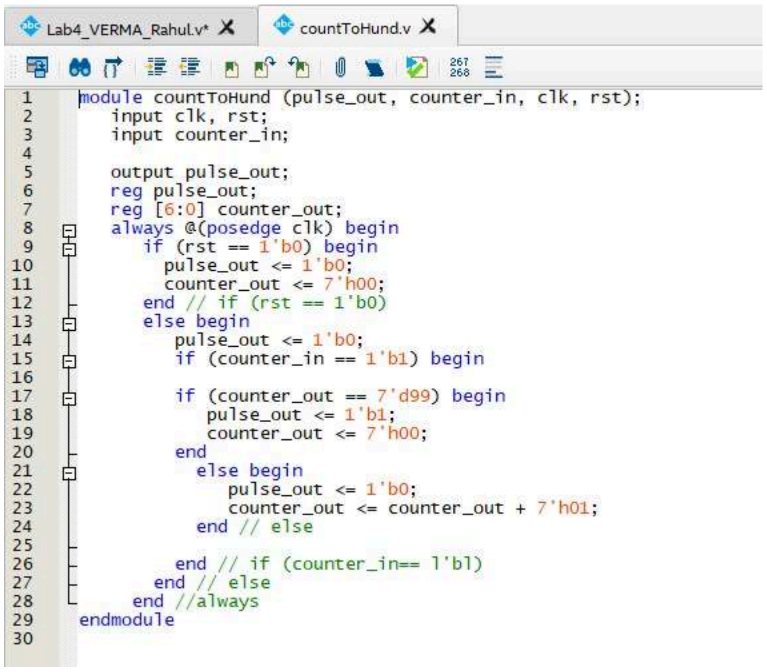
```
//ECE6370
//Author: Rahul Verma, 221462
//Name of the module: hundMilSecTimer
//Description: This module will generate one //a pulse every 1 ms.

module hundMilSecTimer (hundMilSecPulse, enable, clk, rst);

input clk, rst;
input enable;
output hundMilSecPulse;

wire oneMsPulse;

countToFifThouLFSR _countToFifThouLFSR (.pulse_out (oneMsPulse),  .counter_in(enable),.clk(clk), .rst(rst));
countToHund _countToHund (.pulse_out (hundMilSecPulse), .counter_in(oneMsPulse), .clk(clk), .rst(rst));

endmodule
```

Figure 7.0.35: 100 ms Second timer



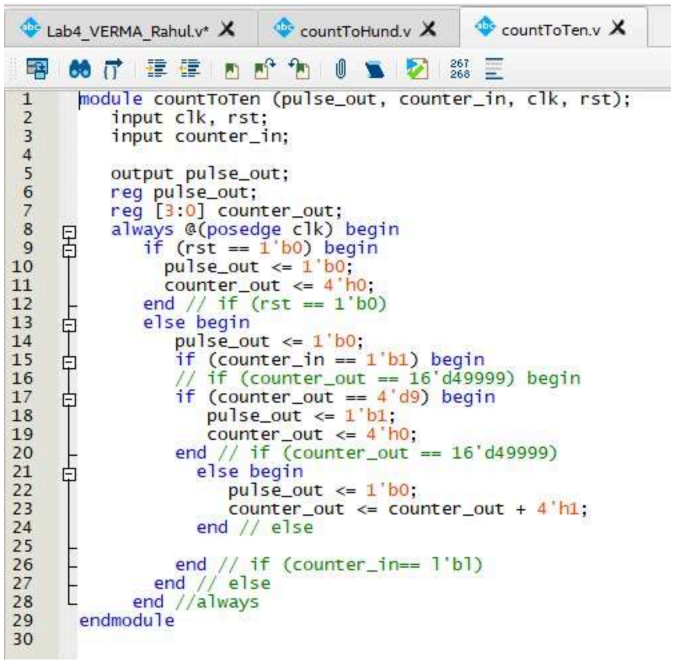
```
//ECE6370
//Author: Rahul Verma, 221462
// Name of the module: oneSecTimer
//Description: This module will generate one
//a pulse every 1 ms.

module oneSecTimer (oneSecPulse, enable, clk, rst);
     input clk, rst;
     input enable;
     output oneSecPulse;
     wire  hundMilSecPulse;


        hundMilSecTimer _hundMilSecTimer (.hundMilSecPulse (hundMilSecPulse), .enable (enable),.clk(clk), .rst (rst));
        countToTen _countToTen (.pulse_out (oneSecPulse), .counter_in (hundMilSecPulse), .clk(clk), .rst (rst));

endmodule
```

Figure 7.0.36: One-Second timer



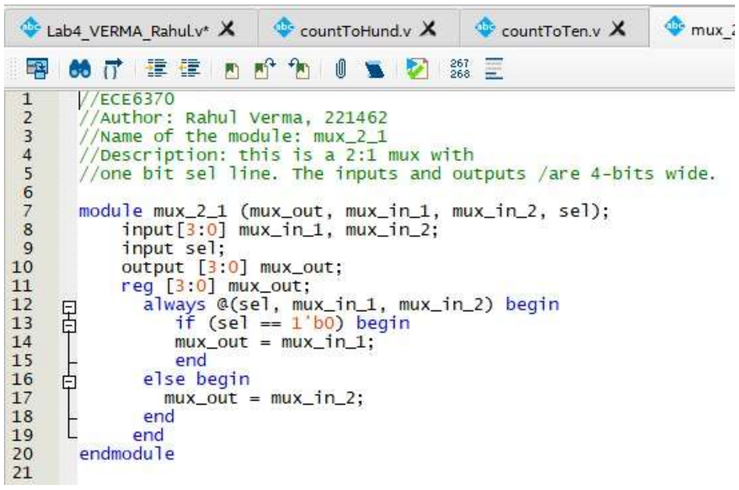
```
module countToHund (pulse_out, counter_in, clk, rst);
   input clk, rst;
   input counter_in;

   output pulse_out;
   reg pulse_out;
   reg [6:0] counter_out;
   always @(posedge clk) begin
      if (rst == 1'b0) begin
        pulse_out <= 1'b0;
        counter_out <= 7'h00;
      end // if (rst == 1'b0)
      else begin
         pulse_out <= 1'b0;
         if (counter_in == 1'b1) begin

         if (counter_out == 7'd99) begin
            pulse_out <= 1'b1;
            counter_out <= 7'h00;
         end
           else begin
              pulse_out <= 1'b0;
              counter_out <= counter_out + 7'h01;
           end // else

         end // if (counter_in== 1'b1)
       end // else
     end //always
endmodule
```

Figure 7.0.38: Count to 100 counter



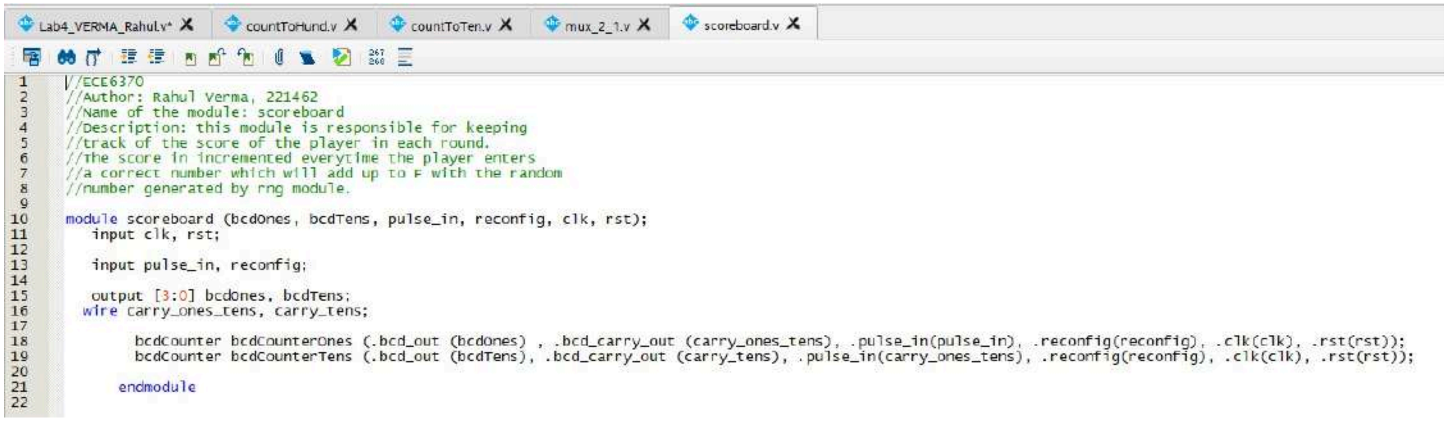
```
module countToTen (pulse_out, counter_in, clk, rst);
   input clk, rst;
   input counter_in;

   output pulse_out;
   reg pulse_out;
   reg [3:0] counter_out;
   always @(posedge clk) begin
      if (rst == 1'b0) begin
        pulse_out <= 1'b0;
        counter_out <= 4'h0;
      end // if (rst == 1'b0)
      else begin
         pulse_out <= 1'b0;
         if (counter_in == 1'b1) begin
         // if (counter_out == 16'd49999) begin
         if (counter_out == 4'd9) begin
            pulse_out <= 1'b1;
            counter_out <= 4'h0;
         end // if (counter_out == 16'd49999)
           else begin
              pulse_out <= 1'b0;
              counter_out <= counter_out + 4'h1;
           end // else

         end // if (counter_in== 1'b1)
       end // else
     end //always
endmodule
```

Figure 7.0.39: Count to 10 counter



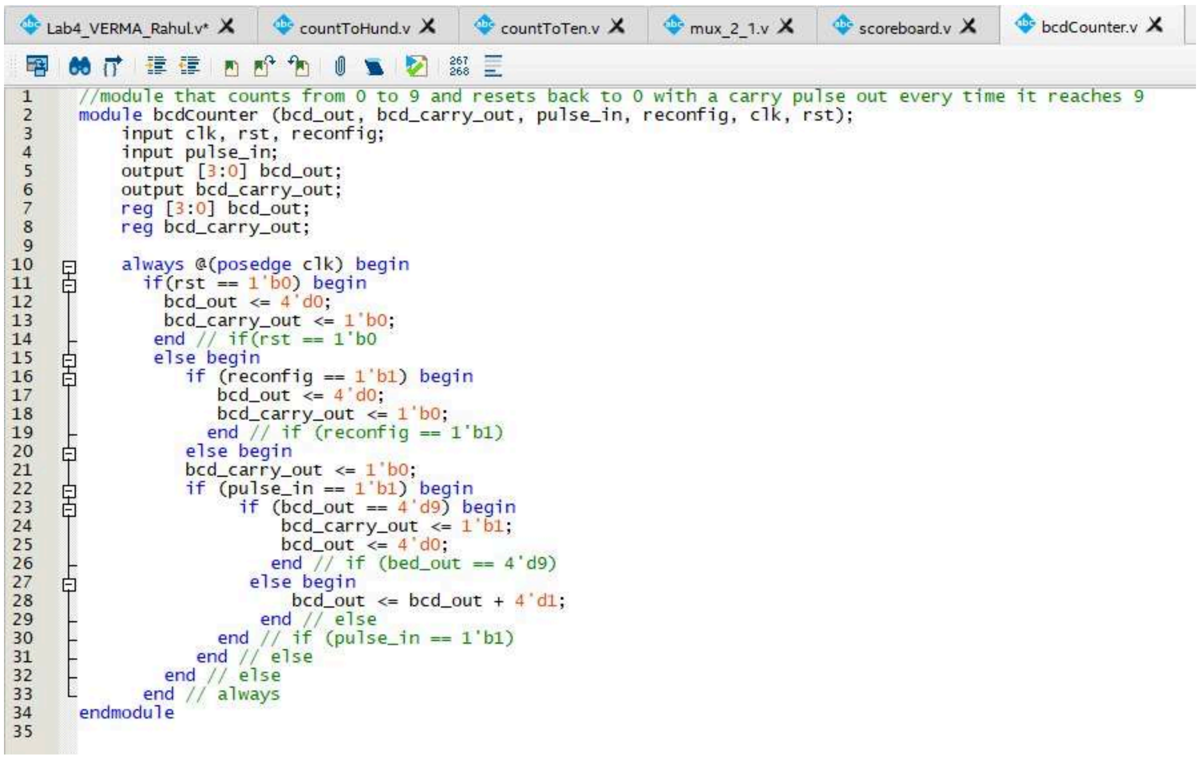
```
//ECE6370
//Author: Rahul Verma, 221462
//Name of the module: mux_2_1
//Description: this is a 2:1 mux with
//one bit sel line. The inputs and outputs /are 4-bits wide.

module mux_2_1 (mux_out, mux_in_1, mux_in_2, sel);
    input[3:0] mux_in_1, mux_in_2;
    input sel;
    output [3:0] mux_out;
    reg [3:0] mux_out;
      always @(sel, mux_in_1, mux_in_2) begin
         if (sel == 1'b0) begin
         mux_out = mux_in_1;
         end
      else begin
        mux_out = mux_in_2;
      end
     end
endmodule
```

Figure 7.0.40: 2x1 MUX



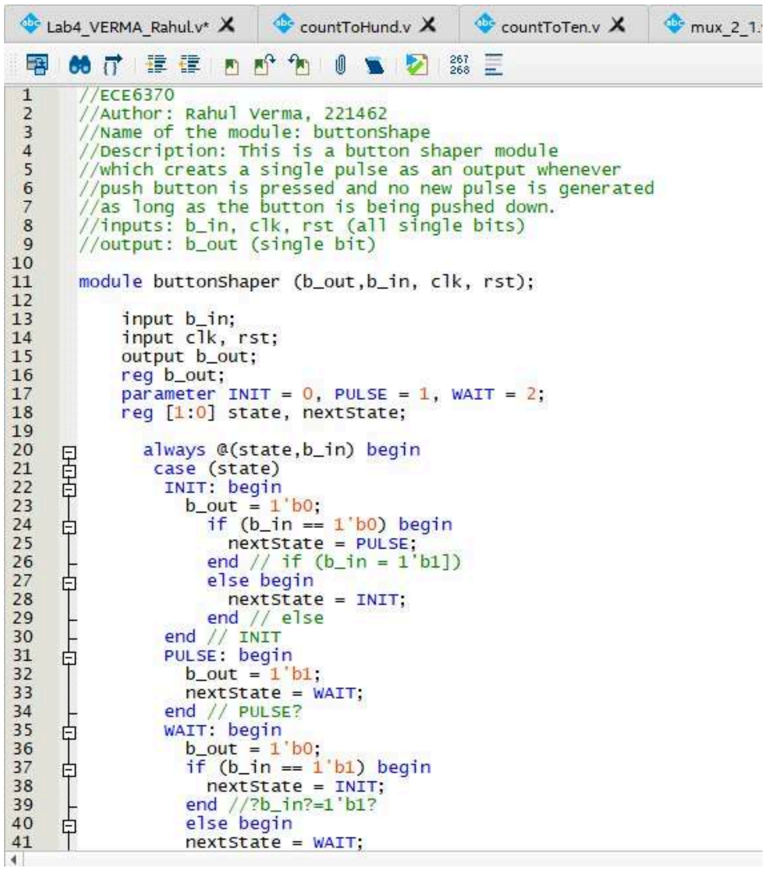
```
//ECE6370
//Author: Rahul Verma, 221462
//Name of the module: scoreboard
//Description: this module is responsible for keeping
//track of the score of the player in each round.
//The score in incremented everytime the player enters
//a correct number which will add up to F with the random
//number generated by rng module.

module scoreboard (bcdOnes, bcdTens, pulse_in, reconfig, clk, rst);
   input clk, rst;

   input pulse_in, reconfig;

   output [3:0] bcdOnes, bcdTens;
  wire carry_ones_tens, carry_tens;

        bcdCounter bcdCounterOnes (.bcd_out (bcdOnes) , .bcd_carry_out (carry_ones_tens), .pulse_in(pulse_in), .reconfig(reconfig), .clk(clk), .rst(rst));
        bcdCounter bcdCounterTens (.bcd_out (bcdTens), .bcd_carry_out (carry_tens), .pulse_in(carry_ones_tens), .reconfig(reconfig), .clk(clk), .rst(rst));

      endmodule
```

Figure 7.0.41: Scoreboard



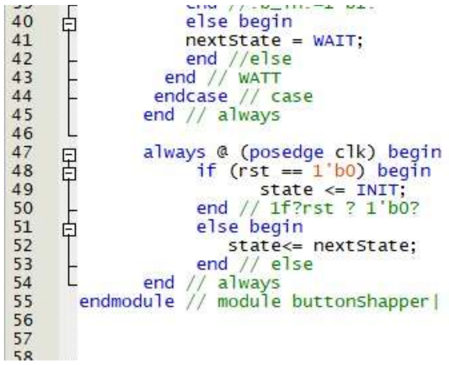
```
//module that counts from 0 to 9 and resets back to 0 with a carry pulse out every time it reaches 9
module bcdCounter (bcd_out, bcd_carry_out, pulse_in, reconfig, clk, rst);
    input clk, rst, reconfig;
    input pulse_in;
    output [3:0] bcd_out;
    output bcd_carry_out;
    reg [3:0] bcd_out;
    reg bcd_carry_out;

    always @(posedge clk) begin
      if(rst == 1'b0) begin
        bcd_out <= 4'd0;
        bcd_carry_out <= 1'b0;
       end // if(rst == 1'b0
       else begin
          if (reconfig == 1'b1) begin
             bcd_out <= 4'd0;
             bcd_carry_out <= 1'b0;
            end // if (reconfig == 1'b1)
          else begin
          bcd_carry_out <= 1'b0;
          if (pulse_in == 1'b1) begin
               if (bcd_out == 4'd9) begin
                   bcd_carry_out <= 1'b1;
                   bcd_out <= 4'd0;
                  end // if (bed_out == 4'd9)
                else begin
                    bcd_out <= bcd_out + 4'd1;
                 end // else
             end // if (pulse_in == 1'b1)
           end // else
        end // else
      end // always
endmodule
```

Figure 7.0.42: BCD counter



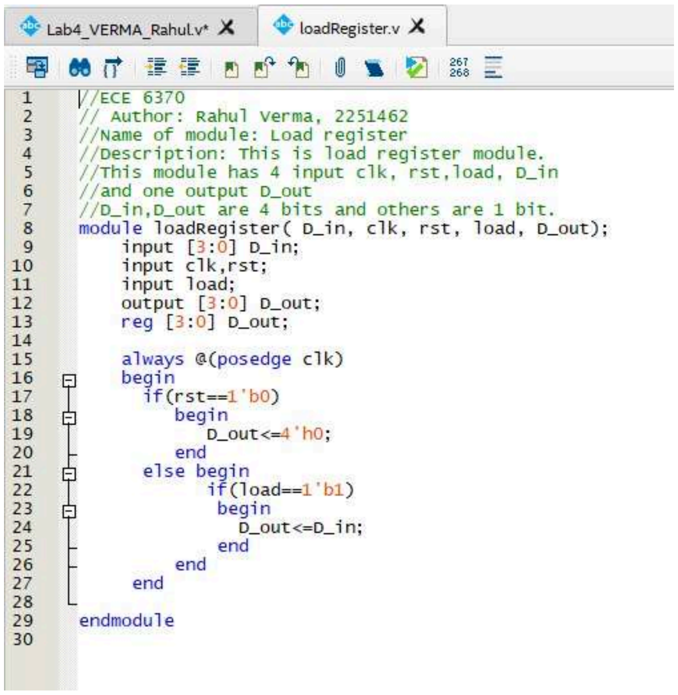
```
//ECE6370
//Author: Rahul Verma, 221462
//Name of the module: buttonShape
//Description: This is a button shaper module
//which creats a single pulse as an output whenever
//push button is pressed and no new pulse is generated
//as long as the button is being pushed down.
//inputs: b_in, clk, rst (all single bits)
//output: b_out (single bit)

module buttonShaper (b_out,b_in, clk, rst);

    input b_in;
    input clk, rst;
    output b_out;
    reg b_out;
    parameter INIT = 0, PULSE = 1, WAIT = 2;
    reg [1:0] state, nextState;

      always @(state,b_in) begin
       case (state)
        INIT: begin
          b_out = 1'b0;
            if (b_in == 1'b0) begin
              nextState = PULSE;
            end // if (b_in = 1'b1])
            else begin
              nextState = INIT;
            end // else
        end // INIT
        PULSE: begin
          b_out = 1'b1;
          nextState = WAIT;
        end // PULSE?
        WAIT: begin
          b_out = 1'b0;
          if (b_in == 1'b1) begin
            nextState = INIT;
          end //?b_in?=1'b1?
          else begin
          nextState = WAIT;
```

Figure 7.0.43: Button shaper (1 of 2)



```
          else begin
          nextState = WAIT;
          end //else
        end // WATT
       endcase // case
      end // always

      always @ (posedge clk) begin
           if (rst == 1'b0) begin
                 state <= INIT;
           end // 1f?rst ? 1'b0?
           else begin
              state<= nextState;
           end // else
      end // always
endmodule // module buttonShapper
```

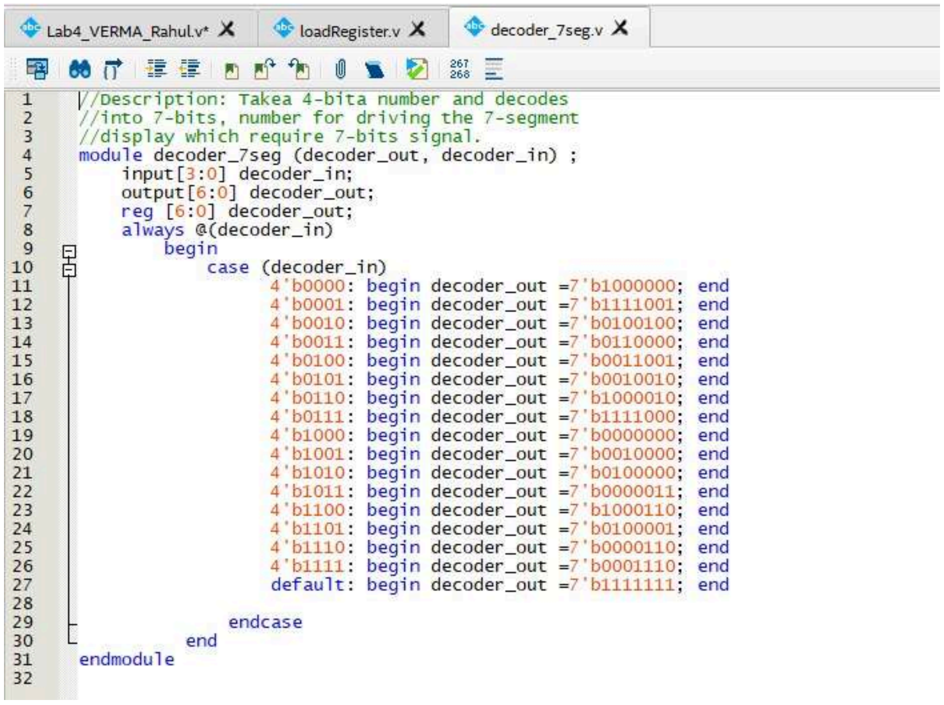
Figure 7.0.44: Button shaper (2 of 2)



Lab4_VERMA_Rahul.v* loadRegister.v

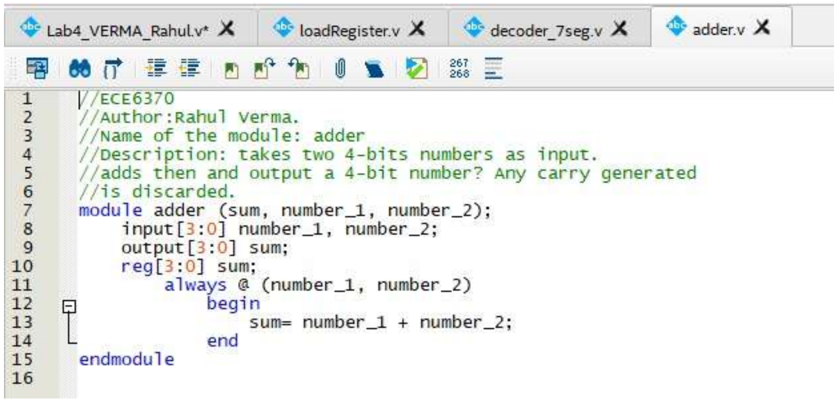
```
//ECE 6370
// Author: Rahul Verma, 2251462
//Name of module: Load register
//Description: This is load register module.
//This module has 4 input clk, rst,load, D_in
//and one output D_out
//D_in,D_out are 4 bits and others are 1 bit.
module loadRegister( D_in, clk, rst, load, D_out);
    input [3:0] D_in;
    input clk,rst;
    input load;
    output [3:0] D_out;
    reg [3:0] D_out;

    always @(posedge clk)
    begin
      if(rst==1'b0)
         begin
            D_out<=4'h0;
         end
      else begin
            if(load==1'b1)
             begin
               D_out<=D_in;
             end
         end
     end

endmodule
```

Figure 7.0.46: Load register



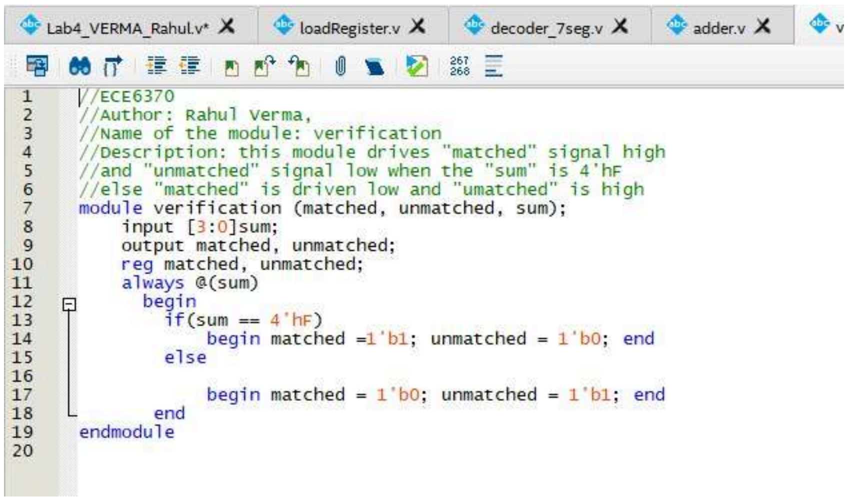
```verilog
//Description: Takea 4-bita number and decodes
//into 7-bits, number for driving the 7-segment
//display which require 7-bits signal.
module decoder_7seg (decoder_out, decoder_in) ;
    input[3:0] decoder_in;
    output[6:0] decoder_out;
    reg [6:0] decoder_out;
    always @(decoder_in)
        begin
            case (decoder_in)
                  4'b0000: begin decoder_out =7'b1000000; end
                  4'b0001: begin decoder_out =7'b1111001; end
                  4'b0010: begin decoder_out =7'b0100100; end
                  4'b0011: begin decoder_out =7'b0110000; end
                  4'b0100: begin decoder_out =7'b0011001; end
                  4'b0101: begin decoder_out =7'b0010010; end
                  4'b0110: begin decoder_out =7'b1000010; end
                  4'b0111: begin decoder_out =7'b1111000; end
                  4'b1000: begin decoder_out =7'b0000000; end
                  4'b1001: begin decoder_out =7'b0010000; end
                  4'b1010: begin decoder_out =7'b0100000; end
                  4'b1011: begin decoder_out =7'b0000011; end
                  4'b1100: begin decoder_out =7'b1000110; end
                  4'b1101: begin decoder_out =7'b0100001; end
                  4'b1110: begin decoder_out =7'b0000110; end
                  4'b1111: begin decoder_out =7'b0001110; end
                  default: begin decoder_out =7'b1111111; end

              endcase
          end
endmodule
```

Figure 7.0.49: 7-segment decoder



```verilog
//ECE6370
//Author:Rahul Verma.
//Name of the module: adder
//Description: takes two 4-bits numbers as input.
//adds then and output a 4-bit number? Any carry generated
//is discarded.
module adder (sum, number_1, number_2);
    input[3:0] number_1, number_2;
    output[3:0] sum;
    reg[3:0] sum;
        always @ (number_1, number_2)
            begin
                sum= number_1 + number_2;
            end
endmodule
```

Figure 7.0.51: Adder



```verilog
//ECE6370
//Author: Rahul Verma,
//Name of the module: verification
//Description: this module drives "matched" signal high
//and "unmatched" signal low when the "sum" is 4'hF
//else "matched" is driven low and "umatched" is high
module verification (matched, unmatched, sum);
    input [3:0]sum;
    output matched, unmatched;
    reg matched, unmatched;
    always @(sum)
      begin
        if(sum == 4'hF)
            begin matched =1'b1; unmatched = 1'b0; end
        else

            begin matched = 1'b0; unmatched = 1'b1; end
       end
endmodule
```

Figure 7.0.53: Verification