Lab 4: ROM-based GameAccess Control on FPGA

ECE 5440

Damon Spencer
UH ID: 1615097

UNIVERSITY of HOUSTON | ECE

Contents

1 Introduction

For this lab assignment, I created an FPGA-based mental binary math game with game access control. First, the player needs to enter the last four digits of my student ID (5097). The game will not accept inputs until then. The logged out LED will also be lit at this time. To enter the password, enter the digits of my student ID one by one using the password switches. Press the password button after each digit is completed. After entering in all four digits, the logged in LED will light up and the logged out LED will switch off. To play the game, player one presses the random number generator button which will generate a random number. After doing this, the player needs to hit the player load button to load in their number. Next, the player tries to guess the binary number that when added to random number generator's number, equals F in hexadecimal. The player inputs this number using the player slide switches and then presses the player load button. The numbers the random number generator inputted, the player inputted, and the sum of the random number and the player's number numbers is outputted. If the sum of the two players numbers equals F in hexadecimal, the matching LED lights up. Otherwise, the non-matching LED lights up. If the LEDs light up, the player scores a point. The score is recorded manually. At the end of the game, the player with the highest score wins.The player can logout by pressing the player 1 load button after a round ends. The payer can also reset their password by pressing the random number generator button after a round ends. To reset their password, press the random number generation button. Then flip the password switches to enter the first digit of the new password in binary. To save the first digit, hit the password button. After this, the second digit of the new password may be set. To enter the second digit, flip the password switches again with the binary version of the digit and hit the password button.

Figure 1 below shows an image of the FPGA board along with the switches and displays used:

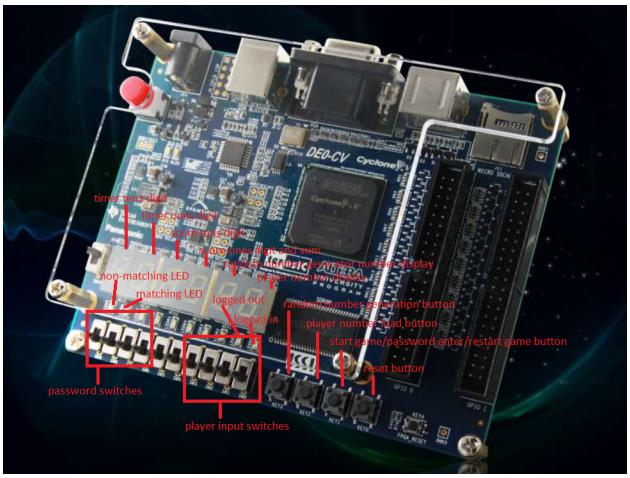

Figure 1: FPGA board picture

2 System architecture design

2.1 Top level system architecture

The top level design of my FPGA based mental binary math game with game access control uses one adder instance along with five seven segment display module instances, two buttonShaper instances, one LoadRegister instance, one rng instance, one score counter instance, one two digit timer instance, an authentication module instance, and a gameController instance. The two buttonShaper instances are connected to the player load button, and the password button respectively. The buttonShaper instances also have clock and reset inputs. The game controller instance takes in a clock input, a reset input, the output of the player button shaper instance, a timeout signal from the two digit timer, a signal that is used for the password check and password reset, and a rng button input signal. It outputs two control signals for the loadRegister

instance and the rng instance along with a signal to log out of the game, timeout enable and timer reconfig signals, and a game is over signal. The authentication module takes in a four bit password digit signal, a signal to enter in a password digit, a clock signal, a reset signal, a rom output signal, a signal that reads data from the ram, a game log out signal, and a password reset signal. The authentication module outputs a logged out signal, a logged in signal, an authentication passed signal, a rom address signal, a ram write enable signal, and a ram data write signal. The LoadRegister instance is connected directly to player one's input. The loadRegister instance also has clock and reset inputs. The output of the load register is connected to the adder instance. For the seven segment decoder, one is connected to the output of the player 1 LoadRegister, one is connected to the output of the player 2 LoadRegister, one is connected to the sum output of the adder module. Each of the player's inputs are received via four input switches. The outputs from the seven segment display module instances are displayed on the seven segment displays in the FPGA diagram above, and the adder's wongame and didnotwingame outputs are displayed on LED_R9 and LED_R0 respectively. The two digit timer module takes the clock, reset, timeout enable, and timer reconfig signals as inputs, and outputs the ones digit of the timer, the tens digit, and a timeout signal. The rng module takes in an rng generator signal, a clock signal, and a reset signal. It outputs a random number between 0x0 and 0xF. The rom module takes in an address and clock and outputs the data at that address to q. The ram module takes in an address, clock, input data, and a write enable switch and outputs the data at the inputted address to its q signal. To further explain this, Figure 2 below shows a diagram of the system architecture:

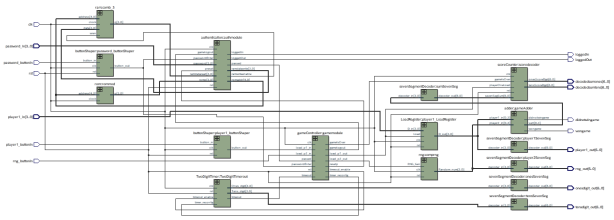

Figure 2: System architecture

2.2 Module symbols

2.2.1 gameController module symbol (logout bonus feature included)

Figure 3 below shows an image of the gameController module:

Figure 3: gameController module symbol

The gameController module uses passed, load_p1_in, load_p2_in, timeout, passwordEnter, resetp, clk, rst as inputs where passed is a signal that is one if the user passed the password check and zero otherwise, load_p1_in is the player load button signal, load_p2_in is the random number generator load signal, timeout is the input timeout signal from the timer, clk is the clock input, passwordEnter is the password entering button signal, resetp is the reset password signal, and rst is the reset input. The gameController module outputs a gamelogout signal, load_p1_out signal, load_p2_out signal, timerconfig signal, timeout_enable signal, and gameIsOver signal where gamelogout sends a pulse if the user wants to log out of teh game, load_p1_out is

the output load signal for the player, load_p2_out is the output load signal for the random number generator, timerconfig configures the timer, timeout_enable enables the timer, and gameIsOver sends a one when the game is over and zero otherwise.

Figure 4 below shows a state machine diagram of the gameController module to help further clarify its function: Figure 5 below shows the conditions for changing state for the above state machine:

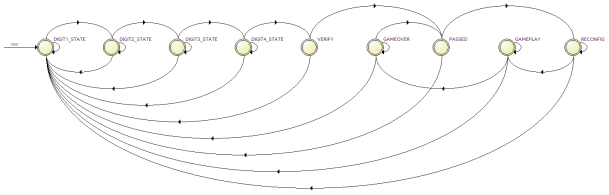

Figure 4: Game controller module state machine diagram

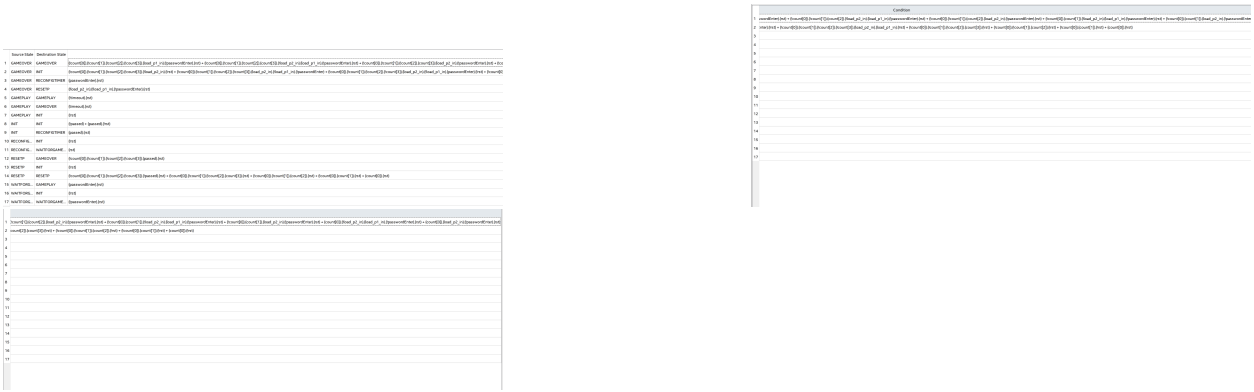

Figure 5: Game controller module state machine transition table

2.2.2 authentication module symbol (logout bonus feature included)

Figure 6 below shows an image of the authentication module:

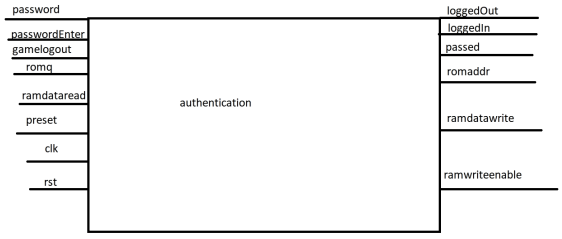

Figure 6: authentication module symbol

The authentication module uses password, passwordEnter, gamelogout, romq, ramdataread, preset, clk, and rst as inputs where password is a signal that gives a digit of the password used to log into the game, passwordEnter is a pulse when the password is entered, gamelogout is a pulse that occurs when the player attempts to log out of the game, romq is the output word of the rom module at the selected address, ramdataread is the data that is read from the ram module, preset is a pulse signal that occurs when the player would like to reset their password, clk is the clock input, and reset is the reset input. It outputs the loggedOut signal, loggedIn signal, passed signal, romaddr, ramdatawrite, and ramwriteenabled where the loggedOut signal is a signal that shows if the user is logged out, the logged in signal shows if the user

is logged in, the passed signal outputs weather the user passed the password check or not, the romaddr outputs the selected address of the rom, the ramdatawrite signal outputs the data to write to the ram, and the ramwriteenable outputs if writing is enabled for the ram or not.

Figure 7 below shows a state machine diagram of the authentication module to help further clarify its function: Figure 8 below shows the conditions for changing state for the above state machine:

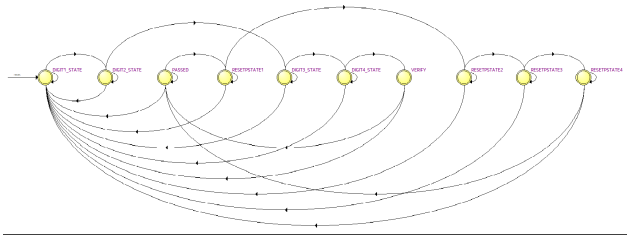

Figure 7: Authentication module state machine diagram

	Source State	Destination State	Condition
1	DIGIT1_STATE	DIGIT1_STATE	(!count[0]).(!count[1]).(!count[2]).(!count[3]).(!rst) + (!count[0]).(!count[1]).(!count[2]).(count[3]) + (!count[0]).(!count[1]).(count[2]) + (!count[0]).(count[1])
2	DIGIT1_STATE	DIGIT2_STATE	(!count[0]).(!count[1]).(!count[2]).(!count[3]).(rst)
3	DIGIT2_STATE	DIGIT1_STATE	(!rst)
4	DIGIT2_STATE	DIGIT3_STATE	(!count[0]).(!count[1]).(!count[2]).(!count[3]).(rst)
5	DIGIT2_STATE	DIGIT2_STATE	(!count[0]).(!count[1]).(!count[2]).(count[3]).(!rst) + (!count[0]).(!count[1]).(count[2]).(!rst) + (!count[0]).(count[1]).(!rst) + (count[0]).(rst)
6	DIGIT3_STATE	DIGIT1_STATE	(!rst)
7	DIGIT3_STATE	DIGIT4_STATE	(!count[0]).(!count[1]).(!count[2]).(!count[3]).(rst)
8	DIGIT3_STATE	DIGIT3_STATE	(!count[0]).(!count[1]).(!count[2]).(count[3]).(!rst) + (!count[0]).(!count[1]).(count[2]).(!rst) + (!count[0]).(count[1]).(!rst) + (count[0]).(rst)
9	DIGIT4_STATE	VERIFY	(!count[0]).(!count[1]).(!count[2]).(!count[3]).(rst)
10	DIGIT4_STATE	DIGIT1_STATE	(!rst)
11	DIGIT4_STATE	DIGIT4_STATE	(!count[0]).(!count[1]).(!count[2]).(count[3]).(!rst) + (!count[0]).(!count[1]).(count[2]).(rst) + (!count[0]).(count[1]).(!rst) + (count[0]).(rst)
12	PASSED	DIGIT1_STATE	(!gamelogout).(!rst) + (gamelogout)
13	PASSED	RESETPSTATE1	(preset).(!gamelogout).(rst)
14	PASSED	PASSED	(!preset).(!gamelogout).(rst)
15	RESETPSTATE1	DIGIT1_STATE	(!rst)
16	RESETPSTATE1	RESETPSTATE2	(!count[0]).(!count[1]).(!count[2]).(!count[3]).(rst)
17	RESETPSTATE1	RESETPSTATE1	(!count[0]).(!count[1]).(!count[2]).(count[3]).(!rst) + (!count[0]).(!count[1]).(count[2]).(rst) + (!count[0]).(count[1]).(!rst) + (count[0]).(rst)
18	RESETPSTATE2	RESETPSTATE3	(!count[0]).(!count[1]).(!count[2]).(count[3]).(rst)
19	RESETPSTATE2	DIGIT1_STATE	(!rst)
20	RESETPSTATE2	RESETPSTATE2	(!count[0]).(!count[1]).(!count[2]).(count[3]).(!rst) + (!count[0]).(!count[1]).(count[2]).(rst) + (!count[0]).(count[1]).(!rst) + (count[0]).(rst)
21	RESETPSTATE3	RESETPSTATE4	(!count[0]).(!count[1]).(!count[2]).(count[3]).(rst)
22	RESETPSTATE3	RESETPSTATE3	(!count[0]).(!count[1]).(!count[2]).(count[3]).(!rst) + (!count[0]).(!count[1]).(count[2]).(rst) + (!count[0]).(count[1]).(!rst) + (count[0]).(rst)
23	RESETPSTATE3	DIGIT1_STATE	(!rst)
24	RESETPSTATE4	RESETPSTATE4	(!count[0]).(!count[1]).(!count[2]).(count[3]).(!rst) + (!count[0]).(!count[1]).(count[2]).(rst) + (!count[0]).(count[1]).(!rst) + (count[0]).(rst)
25	RESETPSTATE4	DIGIT1_STATE	(!rst)
26	RESETPSTATE4	PASSED	(!count[0]).(!count[1]).(!count[2]).(count[3]).(rst)
27	VERIFY	DIGIT1_STATE	(!Equal5) + (!Equal5).(!Equal4) + (!Equal5).(Equal4).(!Equal2) + (!Equal5).(Equal4).(Equal2).(!Equal3) + (!Equal5).(Equal4).(Equal2).(Equal3).(!rst)
28	VERIFY	PASSED	(Equal5).(Equal4).(Equal2).(Equal3).(rst)

Figure 8: Authentication module state machine transition table

2.2.3 ram module symbol (part of password reset bonus feature)

Figure 9 below shows an image of the ram module:

Figure 9: ram module symbol

The ram module uses clk, address, data, and wren as inputs where clk is the clock input address is the address of the rom word that the program wants to read, data is the data the program wants to write to the ram, and write enable enables writing to the ram if it is 1, and reading otherwise. It outputs the word at the address that is inputted to the ram module to q.

2.2.4 rom module symbol

Figure 10 below shows an image of the rom module:

Figure 10: rom module symbol

The rom module uses clk, and address as inputs where clk is the clock input and address is the address of the rom word that the program wants to read. It outputs the word at the address that is inputted to the rom module to q.

2.2.5 onemslfsr module symbol

Figure 11 below shows an image of the onemslfsr module:

Figure 11: One millisecond LFSR timer symbol

The onemslfsr module uses clk, rst, and enable as inputs where clk is the clock input, reset is the reset input, enable is the input that turns the timer on, and onems_Timerout is the output of the module and is a one cycle pulse every one millisecond. The timer uses an internal linear feedback shift register to operate the timer.

2.2.6 Score counter module symbol (bonus feature from Lab 3)

Figure 12 below shows the module symbol for my score counter, which is used for the bonus feature:

Figure 12: Score Counter module symbol

This module implements automatic scoring for the FPGA based mental binary math game. The scoreCounter module receives a clock input as clk, a reset input as rst, the sum of the players number and the random number generator's number decoded into a format suitable for input into a seven segment display as sevenSegSum, the player one button signal outputted from the access controller as playerOneLoad, and a game over signal from the access controller as gameIsOver. Note gameIsOver is one if the game is over and zero otherwise. onesScoreDigit outputs the ones digit of the player's score decoded into a format suitable for display on a seven segment display if the game is over, and the sum of the player's number and the random number generator's number otherwise. tensScoreDigit displays outputs the tens digit of the player's score decoded into a format suitable for display on a seven segment display if the game is over,

and 0b1111111 so that the seven segment display appears off otherwise. The module also uses two internal sevenSegmentDecoder modules.

2.2.7 Two digit timer module symbol

Figure 13 below shows the module symbol for my two digit timer:

Figure 13: Two digit timer module symbol

This module creates a two digit timer for the FPGA. It counts down from 99 seconds to zero and sends a timeout signal when time is up. The module takes a clock signal labled clk, a reset signal labeled rst, a timer enable signal labeled timeout_enable, and a timer reconfiguration signal called timer_reconfig as inputs. The module outputs the ones digit of the current time remaining as Ones_digit, the tens digit of the current time remaining as Tens_digit, and the timeout signal that is one when time is up and zero otherwise as timeout. Internally, the timer module uses two one digit timer modules connected together which are driven by a one second timer that goes off every one second. When 99 seconds elapse, the timer sends a timeout signal.

2.2.8 Digit timer module symbol

Figure 14 below shows the module symbol for my two digit timer:

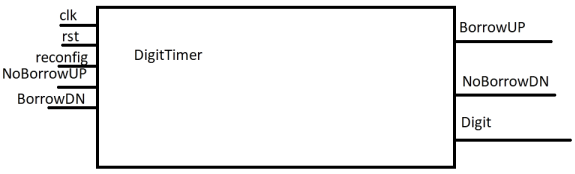

Figure 14: Digit timer module symbol

This module creates a one digit timer for the FPGA. It counts down from 9 to zero and sets NoBorrowDN to one when time runs out. The module takes a clock signal labled clk, a reset signal labeled rst, a reconfiguration signal labeled reconfig, a NoBorrowUP signal that is one when the digit immediately greater than the current digit timer instance cannot be borrowed from and zero otherwise, and a BorrowDN signal that is a one clock cycle pulse when the digit immediately smaller than the current timer would like to borrow a one as inputs. The module outputs the BorrowUP signal when it would like to borrow a one from the digit immediately larger than the current digit, a NoBorrowDN signal when the digit timer that represents a digit immediately to the left of the current digit timer instance cannot borrow, and a Digit signal which outputs the current digit of the digit timer instance.

2.2.9 One second timer module symbol

Figure 15 below shows the module symbol for my one second timer:

This

This module generates a one second timer for the FPGA. The one second timer takes a clock signal called clk, a reset signal called rst, and a timer enable signal called enable as inputs. The enable signal is one when the timer is enabled, and zero otherwise. It outputs a timeout pulse when one second elapses. Internally, the one second timer module passes the enable, clock, and reset signals to an internal one hundred millisecond

Figure 15: One second timer module symbol

timer module. The output of the internal one hundred millisecond module is passed to an internal module called count to 10 that sends an output pulse when 10 input pulses have been received. When the internal count to 10 module sends out an output pulse, one second has elapsed.

2.2.10 One hundred millisecond timer module symbol

Figure 16 below shows the module symbol for my one hundred millisecond timer:

Figure 16: One hundred millisecond timer module symbol

This

This module generates a one hundred millisecond timer for the FPGA. The one hundred millisecond timer takes a clock signal called clk, a reset signal called rst, and a timer enable signal called enable as inputs. The enable signal is one when the timer is enabled, and zero otherwise. It outputs a timeout pulse when one second elapses. Internally, the one hundred millisecond timer module passes the enable, clock, and reset signals to an internal one millisecond timer module. The output of the internal one millisecond module is passed to an internal module called count to 100 that sends an output pulse when 100 input pulses have been received. When the internal count to 100 module sends out an output pulse, one hundred milliseconds have elapsed.

2.2.11 One millisecond timer module symbol

Figure 17 below shows the module symbol for my one millisecond timer:

Figure 17: One millisecond timer module symbol

This

This module generates a one millisecond timer for the FPGA. The one millisecond timer takes a clock signal called clk, a reset signal called rst, and a timer enable signal called enable as inputs. The enable signal is one when the timer is enabled, and zero otherwise. It outputs a timeout pulse when one second elapses. Internally, the one millisecond timer module produces a timeout signal every 50,000 clock cycles and by then one millisecond elapses.

2.2.12 Count to ten module symbol

Figure 18 below shows the module symbol for my count to ten module:

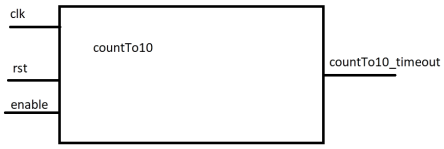

Figure 18: Count to ten module symbol

This module generates a one clock cycle pulse every time ten input pulses are received. The countTo10 module takes a clock signal called clk, a reset signal called rst, and a timer enable signal called enable as inputs. The enable signal is one when the timer is enabled, and zero otherwise.

2.2.13 Count to one hundred module symbol

Figure 19 below shows the module symbol for my count to one hundred module:

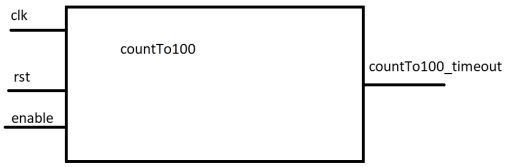

Figure 19: Count to one hundred module symbol

This module generates a one clock cycle pulse every time one hundred input pulses are received. The countTo100 module takes a clock signal called clk, a reset signal called rst, and a timer enable signal called enable as inputs. The enable signal is one when the timer is enabled, and zero otherwise.

2.2.14 Random number generator module symbol (LFSR bonus feature included)

Figure 20 below shows the module symbol for my random number generator which uses a LFSR:

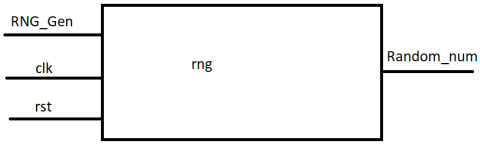

Figure 20: Digit timer module symbol

This module generates a random number for the FPGA. The random number generator module takes a signal that tells it to generate a random number, a clock signal as clk, and a reset signal as rst as inputs. The RNG_Gen signal is zero when the random number generator should generate a random number, and one otherwise. It is a button press from a physical button. The Random_num signal is the only output, and is a 4 bit random number. The random number generator works by incrementing a LFSR for each clock

cycle that the RNG_Gen signal is zero. During this time the Random_num signal will change between 0x0 and 0xF as the LFSR is changed. Random_num is sourced from several bits of the LFSR. When Rng_Gen signal is one, the current Random_num value will be outputted without being changed. Since there is no way to know exactly how many clock cycles a button press will be for, the output will be a random number.

2.2.15 buttonShaper module symbol

Figure 21 below shows an image of the buttonShaper module:

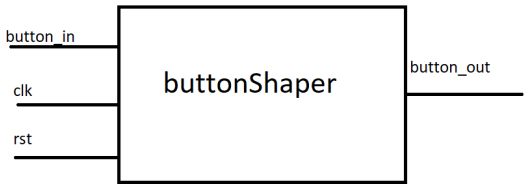

Figure 21: Button shaper module symobol

The purpose of this module is to turn an input button press into a single clock cycle pulse. button_in is the input signal for the button press, rst is the reset button input, clk is the FPGA clock input, and button_out is the button pulse output. Figure 22 below shows a state machine diagram of the buttonShaper module to help further clarify it's function:

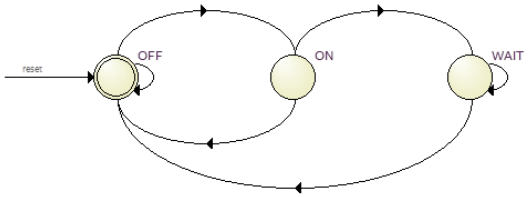

Figure 22: Button shaper module state machine diagram

Figure 23 below shows the conditions for changing state for the above state machine:

	Source State	Destination State	Condition
1	OFF	OFF	(!button_in).(!rst) + (button_in)
2	OFF	ON	(!button_in).(rst)
3	ON	OFF	(!rst)
4	ON	WAIT	(rst)
5	WAIT	OFF	(!button_in).(!rst) + (button_in)
6	WAIT	WAIT	(!button_in).(rst)

Figure 23: Button shaper module state machine transition table

In addition, Figure 24 below shows the expected waveform for the button shaper module:

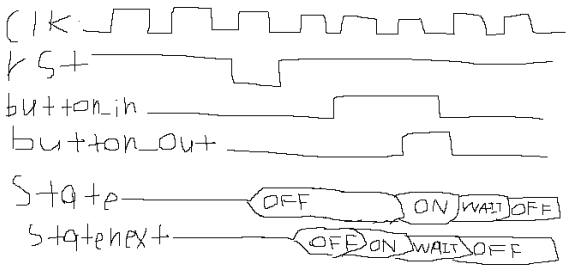

Figure 24: Button shaper module waveform

2.2.16 LoadRegister module symbol

Figure 25 below shows an image of the LoadRegister module:

Figure 25: Load register module symobol

The purpose of this module is to store a four bit number into a register for later use. It is controlled via the load, clk, and rst signals. D_in is the number LoadRegister stores in its register, D_out is the number outputted by the register, Load is the load signal, rst is the reset button input, and clk is the FPGA clock input.

2.2.17 Seven segment decoder module symbol

Figure 26 below shows the module symbol for my seven segment decoder:

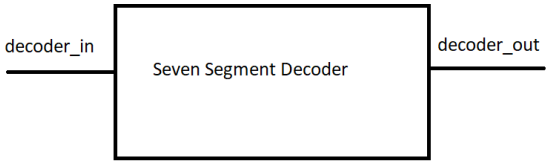

Figure 26: Seven segment decoder module symbol

The purpose of the module is to allow a four bit input to be displayed on a seven segment display. There are two signals for the module: decoder_in, and decoder_out. decoder_in is a four bit input, and decoder out is a seven bit output that maps the input bits to a seven segment display digit.

2.2.18 Adder module symbol

Figure 27 below shows the module symbol for my adder:

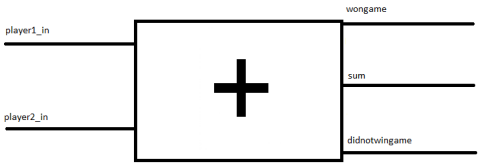

Figure 27: Adder module symbol

The purpose of the module is to allow two four bit numbers to be added together. The module receives two four bit numbers player1_in and player2_in as inputs. It outputs two one bit numbers wongame and didnotwingame along with one seven bit number sum. wongame is one if the sum of player1_in and player2_in is 0xF, and zero otherwise. didnotwingame is zero if the sum of player1_in and player2_in is 0xF, and one otherwise. sum is the sum of the binary numbers player1 and player2 inputted.

3 Simulation screen captures

3.1 rom delay simulation screen capture

Figure 28 below shows a screen capture of my rom module simulation that finds the propagation delay of the rom: To find the propagation delay for the rom, I change the rom read address every few clock cycles and

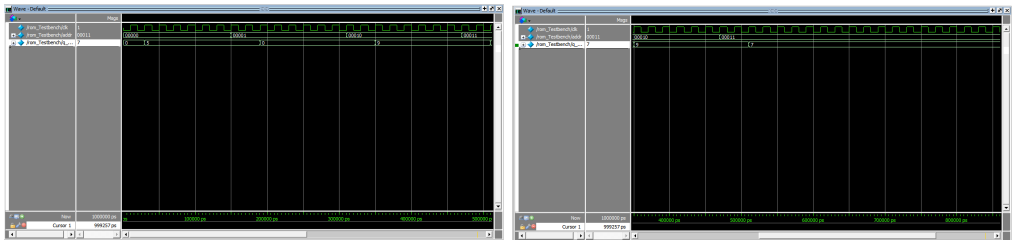

Figure 28: rom delay simulation screen capture

visually determine how many cycles later the q output of the rom changes. I determined the propagation delay to be 2 clock cycles.

3.2 authentication module with rom inside simulation screen capture

Figure 29 below shows a screen capture of my rom module simulation that tests the rom while connected to the authentication module: To test the rom, I tested the login process of the authentication module with the rom connected. I entered the rom password 5097. After entering each digit in binary using the password switches, I hit the password button. This was repeated four times to enter in all digits. After doing this, I was able to successfully log into the device with the rom and the rom q bus displayed every output correctly during the right time.

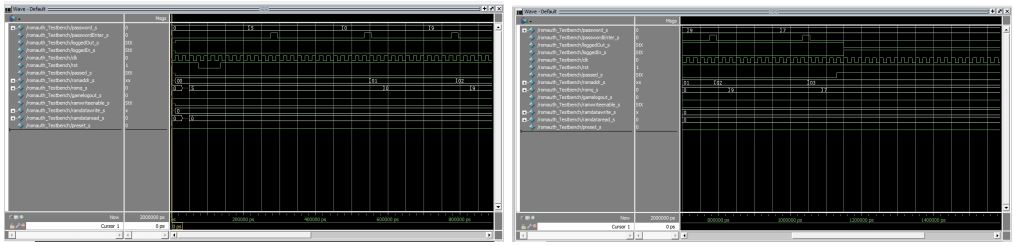

Figure 29: authentication module with rom inside simulation screen capture

3.3 one millisecond LFSR find last number simulation screen capture

Figure 30 below shows a screen capture of my onemslfsr module simulation that finds the last value of the lfsr used: To find the value for the lfsr 1 millisecond timer that occurs when the counter based one millisecond

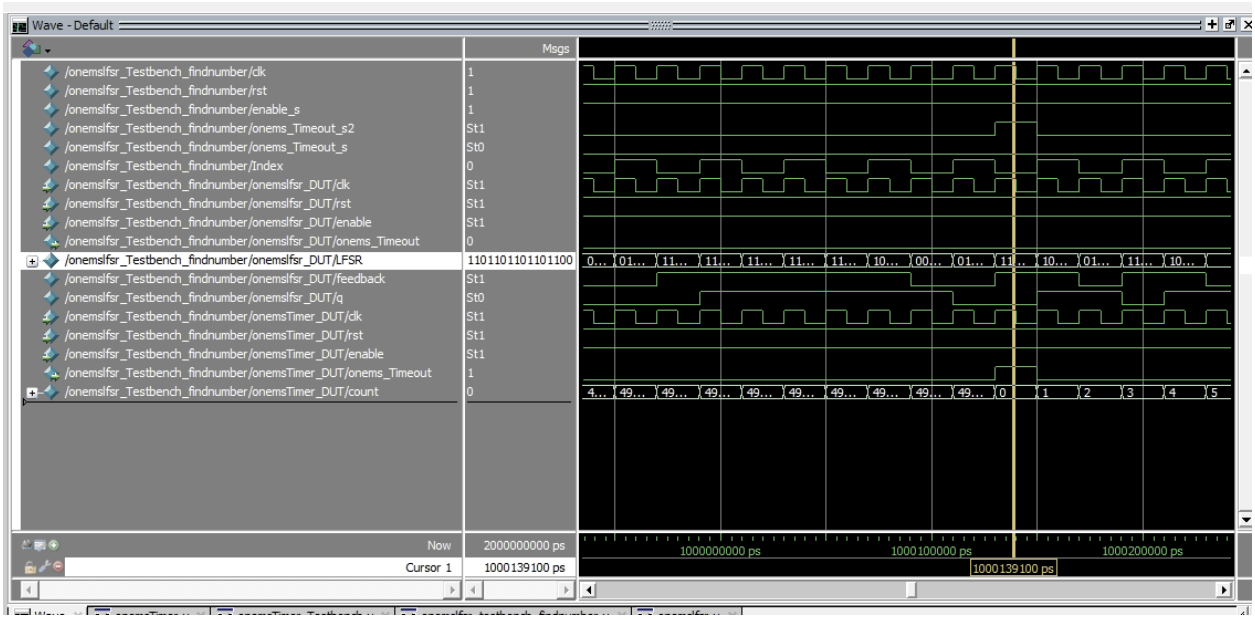

Figure 30: one millisecond lfsr find last number simulation screen capture

timer has a timeout pulse, I created a testbench that runs the clock for 50,000 cycles after a reset. I then search for the clock pulse that the one millisecond timer produces when one millisecond occurs and take the value that the lfsr has at that time as the last value that the lfsr should have after one millisecond. This value ends up being 0b1101101101101100. Note that due to the way my timer is designed, I actually need to use the value of the LFSR one clock cycle before this which is 0b0110110110110110.

3.4 one millisecond LFSR comparison simulation screen capture

Figure 31 below shows a screen capture of my onemslfsr module simulation that compares the LFSR based timer to the counter based timer for accuracy: To test the LFSR based timer, I verified that it produced a timeout pulse at the same time as the counter based timer. The timeout pulses for both modules occured at the same time, which is exactly what I would like.

3.5 Digit timer module simulation screen capture

Figure 32 below shows a screen capture of my digit timer module simulation:

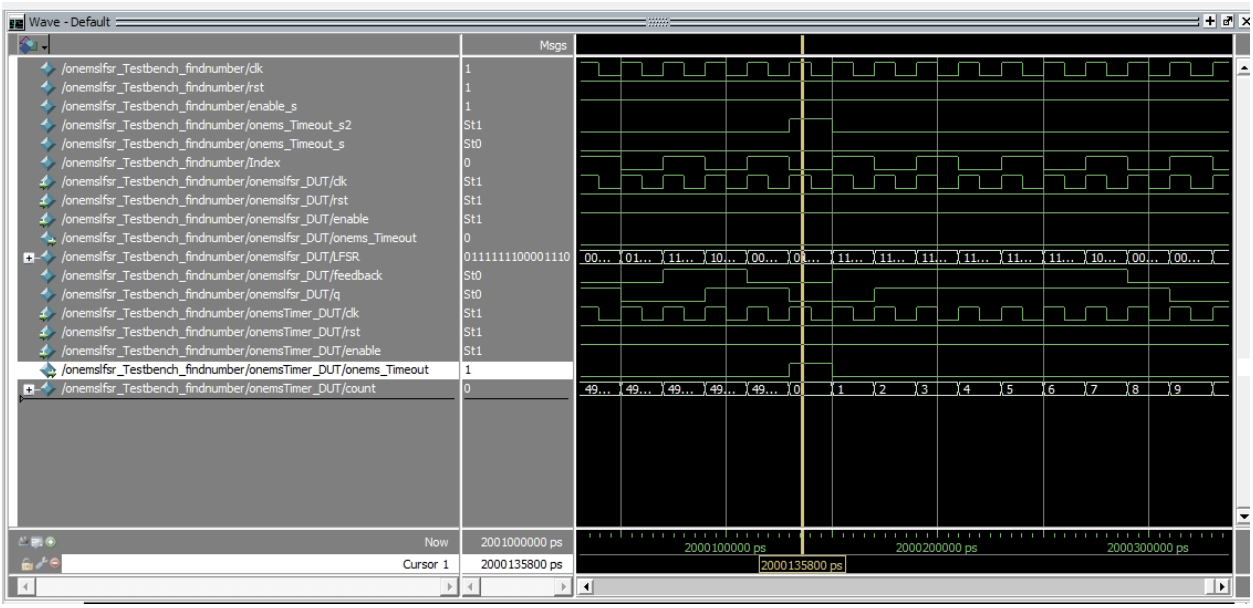

Figure 31: one millisecond lfsr comparison simulation screen capture

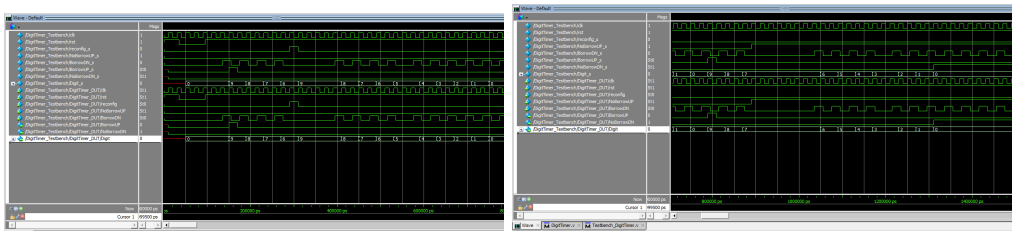

Figure 32: Digit timer module simulation screen capture

The simulation thoroughly tests the digit timer module. The digit timer module is tested with multiple borrowDn pulses and a reconfig signal pulse. The NoBorrowUP signal is also changed in the second half of the simulation from 0 to 1. Initially when running the simulation, the digit is deincremented as the borrowDN signal inputs pulse signals. Next, the reconfig signal is used, and the digit is reset to 9. After this, the digit is deincremented to 0 and borrows a 1 from the next largest digit. After doing this, it continues to deincrement as more borrowDN pulses are inputted. The NoBorrowUP signal is then set to 1 and borrowDN pulses continue to be inputted. Eventually, the digit in the digit timer reaches zero, and the NoBorrowDN flag is set to 1. Overall, the digit timer works correctly.

4 oneSecondTimer module simulation screen capture

Figure 33 below shows a screen capture of my oneSecondTimer module simulation: To test the one second

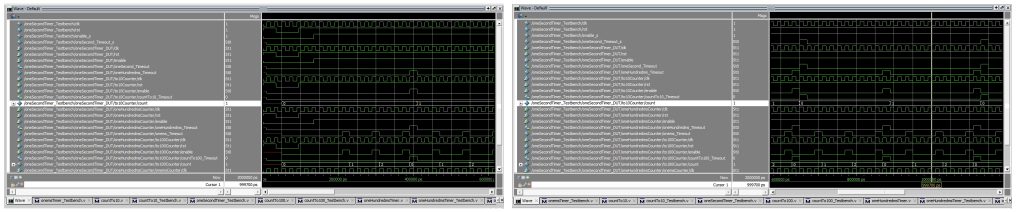

Figure 33: 1 second timer module simulation screen capture

timer, I created a test bench that temporarily lowers reset to zero, sets reset back to one, then sets enable to one for enough clock cycles to see two iterations of the one second timer. The critical case occurs when the one second timer produces a timeout signal. This causes the timeout signal to go high for one clock cycle. In this simulation, the durations of the 1msTimer, countTo10 timer, and countTo100 timer were reduced to 4, 2, and 3 clock cycles each respectively to make the modules easier to test. Since the 1 ms timer was an input to the countTo100 timer and the countTo100 timer was an input to the countTo10 timer, the 1 second timer should take 24 clock cycles to go off. In the simulation, the one second timer goes off as expected, which is great.

5 Random number generator module simulation screen capture

Figure 34 below shows a screen capture of my rng module simulation: To test the random number geneator,

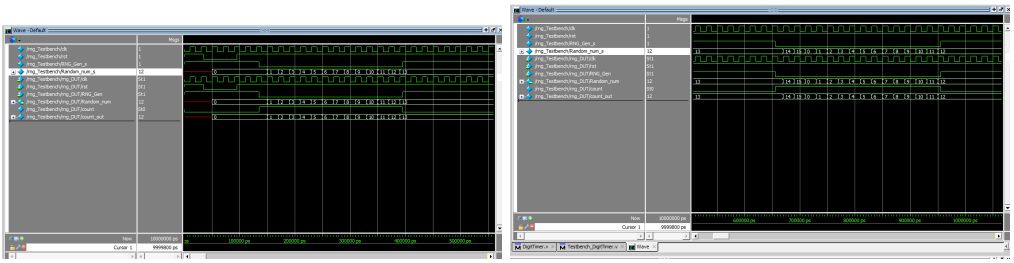

Figure 34: random number generator module simulation screen capture

I created a test bench that sets RNG_Gen equal to zero for thirteen clock cycles, then sets RNG_Gen to one for thirteen more cycles, sets RNG_Gen to zero for fifteen cycles, and finally sets RNG_Gen to one for four clock cycles. The Random number generator first increments its internal counter and its output settles on 13 when the random number generator input is one. Next the random nuumber genrator input is set low again. After finishing its second low signal, the output of the random number generator ends up at 12 during the second round. Overall, the random number generator worked correctly.

5.1 Load register module simulation screen capture

Figure 35 below shows a screen capture of my load register module simulation:

The simulation tests the LoadRegister module quite well. The testbench tests two different instances where a load signal is inputted, with false inputs created during the process. One of the most important tests is what happens when the first load signal is inputted. When the first load signal is inputted, D_in = 4'b0001. One clock cycle later, D_out becomes 4'b0001. For several cycles, D_in is changed without a load signal and D_out does not change. This demonstrates my module works very well.

5.2 Button shaper module simulation screen capture

Figure 36 below shows a screen capture of my button shaper module simulation:

To test the buttonShaper module, I first simulated a reset button press. From the above screenshot, the state variable only has an unknown value before the reset button is pressed, which is good. Next, I simulated two button presses of differing lengths. In both cases the button shaper module produced a one clock cycle pulse. Overall, I think my button shaper module works correctly.

5.3 Seven segment decoder module simulation screen capture

Figure 37 below shows screen captures of my seven segment decoder module simulation:

The simulation tests all 16 cases of the seven segment decoder module. Note for each case, a segment of the seven segment decoder is lit up when the bit corresponding to that segment is zero and is off when the bit corresponding to that segment is one. To test the module code, I compared the input of the module to

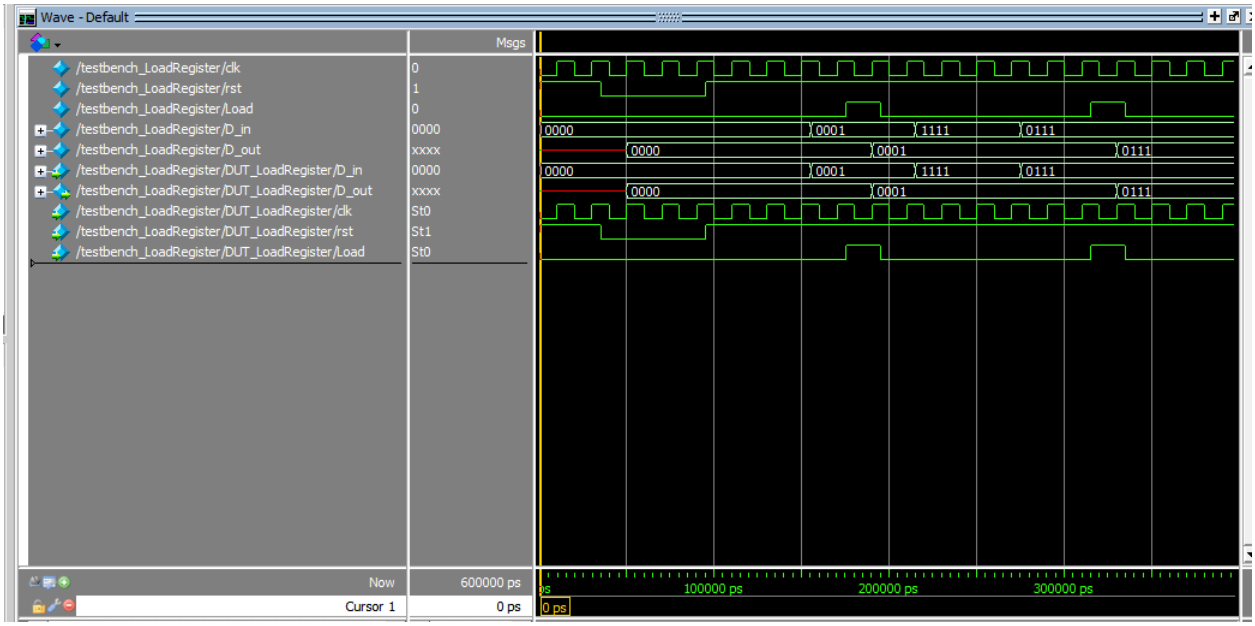

Figure 35: Load register module simulation screen capture

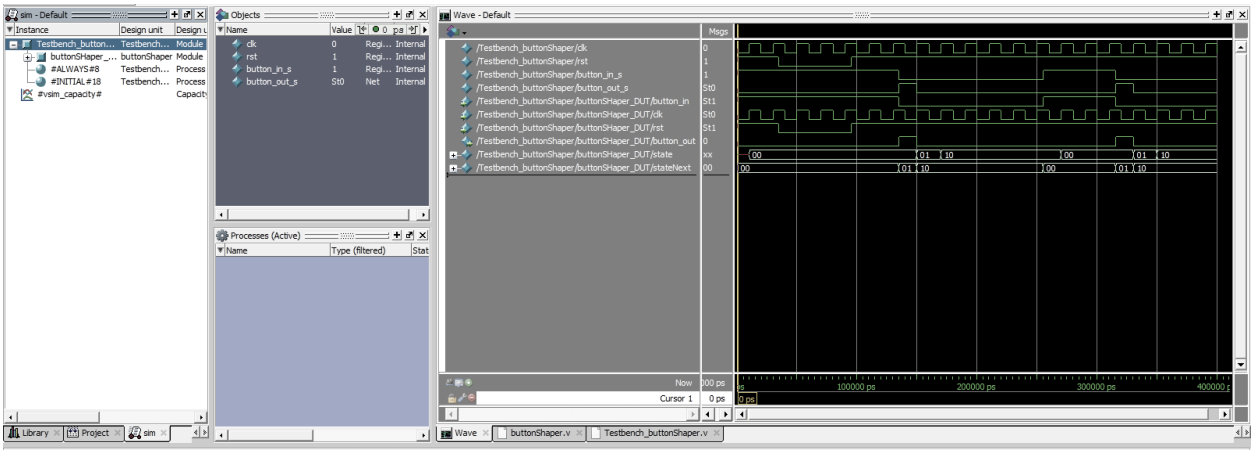

Figure 36: Button Shaper module simulation screen capture

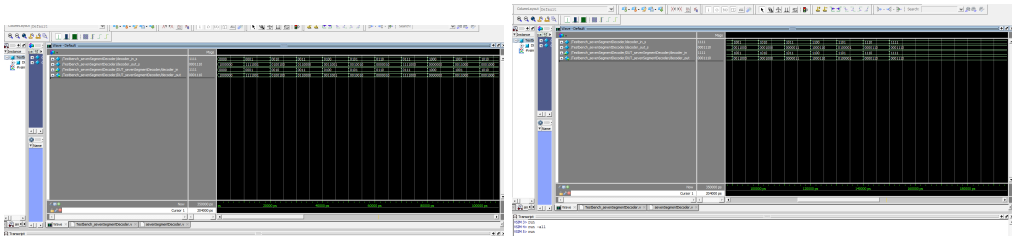

Figure 37: Seven segment decoder module simulation screen capture

the expected output of the module. Every single case outputted exactly what I expected, so I was able to believe my module code was correct. One good benchmark for the module code is inputting 0b1000 should output the binary number 0b0000000. 0b0000000 corresponds to an eight on the seven segment display. My module correctly outputs the number 0b0000000 when the input is 0b1000, so this benchmark case is correct.

5.4 Adder module simulation screen capture

Figure 38 below shows a screen capture of my adder module simulation:

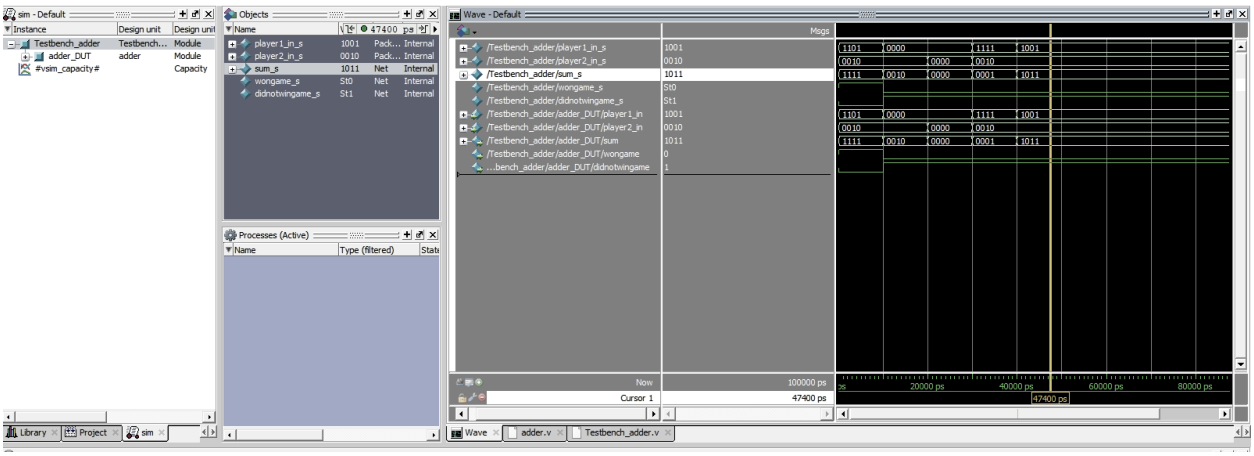

Figure 38: Adder module simulation screen capture

The simulation tests five different cases for the adder module. Below I will describe two critical test cases: a case when the sum of the player's two numbers adds up to 0xF, and another case where the player's two numbers do not add up to 0xF.

To test what happens when the player's numbers add up to 0xF, I created a test case where player 1 inputted 0b1101, and player 2 inputted 0b0010. In that case, the seven segment display with player 1's number outputted player 1's number, the seven segment display with player 2's number outputted player 2's number, and the seven segment display with the sum of both player's numbers outputted 0xF, the won game LED lit up, and the did not win game led did not light up.

To test what happens when the player's numbers do not add up to 0xF, I created a test case where player 1 inputted 0b0000, and player 2 inputted 0b0010. In that case, the seven segment display with player 1's number outputted player 1's number, the seven segment display with player 2's number outputted player 2's number, and the seven segment display with the sum of both player's numbers outputted 0x1, the won game LED did not light up, and the did not win game led lit up.

Overall, the simulation went exactly as I expected.

6 FPGA board testing results

Below are eight images displaying my FPGA board testing results. Figure 39 below shows an image of the FPGA right after pressing the reset button:

Figure 40 below shows an image of the FPGA after logging in with the password 5097. To log in, I entered each digit with the slide switches and then pressed the passwordEnter button after each digit. This happened four times until I was logged in. The timer is then set for 99 seconds:

Note the loggedIn LED turns on and the loggedOut LED turns off.

Next, I hit the game start button to start the game. I then hit the random number generation button to generate a random number. 0xd was generated. I then entered 0x1 and the sum display was set to 0xE. The timer had 48 seconds remaining in this photo.

Figure 41 below shows an image of the FPGA after hitting the random number generator button, entering in 0x1 for the player, and hitting the player load button:

Next, I changed the player input to 0x2 and the sum was set to 0xF. The wongame led lit up while the didnotwingame led turned off. The timer had 42 seconds remaining in this photo. Figure 42 below shows an image of the FPGA after entering in 0x2 for the player and hitting the player load button:

After changing the player input, I waited for the timer to expire. The time ran out and the timer displayed 0 seconds remaining. The score of 1 was then displayed on the FPGA score seven segment displays

where the sum was displayed. Figure 43 below demonstrates this functionality: I then played another round and allowed the timer to expire again. After that, I hit the random number generator button to change the password for the player. I changed the password to 1234. Figure 44 below demonstrates the password change functionality: Next, I proceeded to logout with the password changed. Figure 45 below demonstrates the logout functionality: Finally, I log back in with the new password 1234. The login functionality works as expected with the new password. Figure 46 below demonstrates the login functionality:

7 FPGA demonstration video

7.1 Login feature demonstration

I broke the demonstration videos of my FPGA into several sections. Below is a link to the login feature of the FPGA:

https://drive.google.com/file/d/15xfrzqKUT-PPZI7NQez0g_5lVkUhNRMT/view?usp=sharing

7.2 Basic gameplay demonstration

Below is a video demonstration of the basic gameplay and automatic scoring feature of my FPGA based mental binary math game: https://drive.google.com/file/d/162o6GW0tHPDkY-n2398Zc2Oa6vipB22e/view?usp=sharing

7.3 Password reset, logout, and login demonstration (bonus features)

Below is a video demonstration of the password reset and logout functionality along with an additional login for testing: https://drive.google.com/file/d/164yCRNxPfxuzE_Fu2MQgd6oxnd_em7Xc/view?usp=sharing

8 Conclusion (all bonus features included for lab 4)

Overall my lab 4 works perfectly. There are no known issues, and all three bonus features for the lab are implemented.

9 Appendix

9.1 Verilog module code screen captures

9.1.1 LFSR Random number generator module Verilog code (bonus)

Figure 47 below shows a screen capture of my rng module code with an internal LFSR: The random number generator with a LFSR works quite well.

9.1.2 Game controller Verilog code (bonus incuded)

Figure 48 below shows a screen capture of my game controller module code: My game controller module works as expected.

9.1.3 authentication module Verilog code (bonus included)

Figure 49 below shows a screen capture of my screen capture of my romauth_Testbench module code: The test bench shows the new authentication module works well.

9.1.4 ram module Verilog code (part of bonus)

Figure 50 below shows a screen capture of my ram module code: My ram module works as expected.

9.1.5 rom module Verilog code

Figure 51 below shows a screen capture of my rom module code: My rom module works as expected.

9.1.6 rom module test bench Verilog code

Figure 52 below shows a screen capture of my screen capture of my rom_Testbench module code: The test bench does a great job at finding the propagation delay of the rom module.

9.1.7 rom and authentication module integration test bench Verilog code

Figure 53 below shows a screen capture of my screen capture of my romauth_Testbench module code: The test bench shows the rom works well with the authentication module.

9.1.8 1 millisecond LFSR timer module Verilog code

Figure 54 below shows a screen capture of my onemslfsr module code: My linear feedback shift register is significantly more efficient than the counter based timer.

9.1.9 1 millisecond LFSR timer module test bench Verilog code

Figure 55 below shows a screen capture of my screen capture of my onemslfsr_testbench_findnumber module code: The test bench does a great job at finding the last value needed in the LFSR for the 1 millisecond timer.

9.1.10 Score counter module Verilog code

Figure 56 below shows a screen capture of my scoreCounter module code: The score counter efficiently counts the player's score without errors.

9.1.11 Random number generator module test bench Verilog code

Figure 57 below shows a screen capture of my OneSecondTimer testbench module code: The test bench for the random number generator demonstrates the random number generator's capabilities well.

9.1.12 Digit timer module Verilog code

Figure 58 below shows a screen capture of my DigitTimer module code: The digit timer accurately keeps track of a single digit number.

9.1.13 Digit timer module testbench Verilog code

Figure 59 below shows a screen capture of my DigitTimer testbench module code: The digit timer testbench does a good job of testing teh digit timer's capabilities.

9.1.14 Two Digit timer module Verilog code

Figure 60 below shows a screen capture of my TwoDigitTimer module code: The two digit timer works well for a 99 second timer.

9.1.15 Two Digit timer module testbench Verilog code

Figure 61 below shows a screen capture of my DigitTimer testbench module code: This module thoroughly tests the two digit timer module.

9.1.16 1 second timer module Verilog code

Figure 62 below shows a screen capture of my oneSecondTimer module code: The one second timer module counts seconds extremely well.

9.1.17 1 second timer module testbench Verilog code

Figure 63 below shows a screen capture of my OneSecondTimer testbench module code: The one second timer module testbench uses a shortened one second timer to test out the capabilities of the one second timer module.

9.1.18 100 milliseconds timer module Verilog code

Figure 64 below shows a screen capture of my oneHundredmsTimer module code: The 100 milliseconds timer module goes off every 100 millisecond and works very well.

9.1.19 Count to 10 timer module Verilog code

Figure 65 below shows a screen capture of my countTo10 module code: The count to 10 timer module uses a four bit counter to count to 10.

9.1.20 Count to 100 timer module Verilog code

Figure 66 below shows a screen capture of my countTo100 module code: The count to 100 module accurately counts to 100.

9.1.21 Load register module Verilog code screen capture

Figure 67 below shows screenshots of my load register module Verilog code:
The load register module does a good job at storing number for later use.

9.1.22 Load register test bench module Verilog code screen capture

Figure 68 below shows screenshots of my load register test bench module Verilog code:
The load register module test bench tests two different cases for the load register module and both work as expected.

9.1.23 Button shaper module Verilog code screen capture

Figure 69 below shows screenshots of my button shaper module Verilog code:
The button shaper is quite accurate at turning an arbitrary button press into a single clock cycle pulse.

9.1.24 Button shaper test bench module Verilog code screen capture

Figure 70 below shows screenshots of my button shaper test bench module Verilog code:
The button shaper test bench tests multiple cases with arbitrary length button inputs. Overall, the button shaper module works well.

9.1.25 Seven segment decoder module Verilog code screen capture

Figure 71 below shows a screenshot of my seven segment decoder module Verilog code:
Since the input is four bits, I needed sixteen input combinations to test all cases.

9.1.26 Seven segment decoder test bench module Verilog code screen capture

Figure 72 below shows a screenshot of my seven segment decoder test bench module Verilog code:
The seven segment decoder module test bench includes all sixteen cases.

9.1.27 Adder module Verilog code screen capture

Figure 73 below shows a screenshot of my adder module Verilog code:
The adder module includes all bonus features.

9.1.28 Adder test bench module Verilog code screen capture

Figure 74 below shows a screenshot of my adder test bench module Verilog module code:

The test bench has five different test cases for the adder.

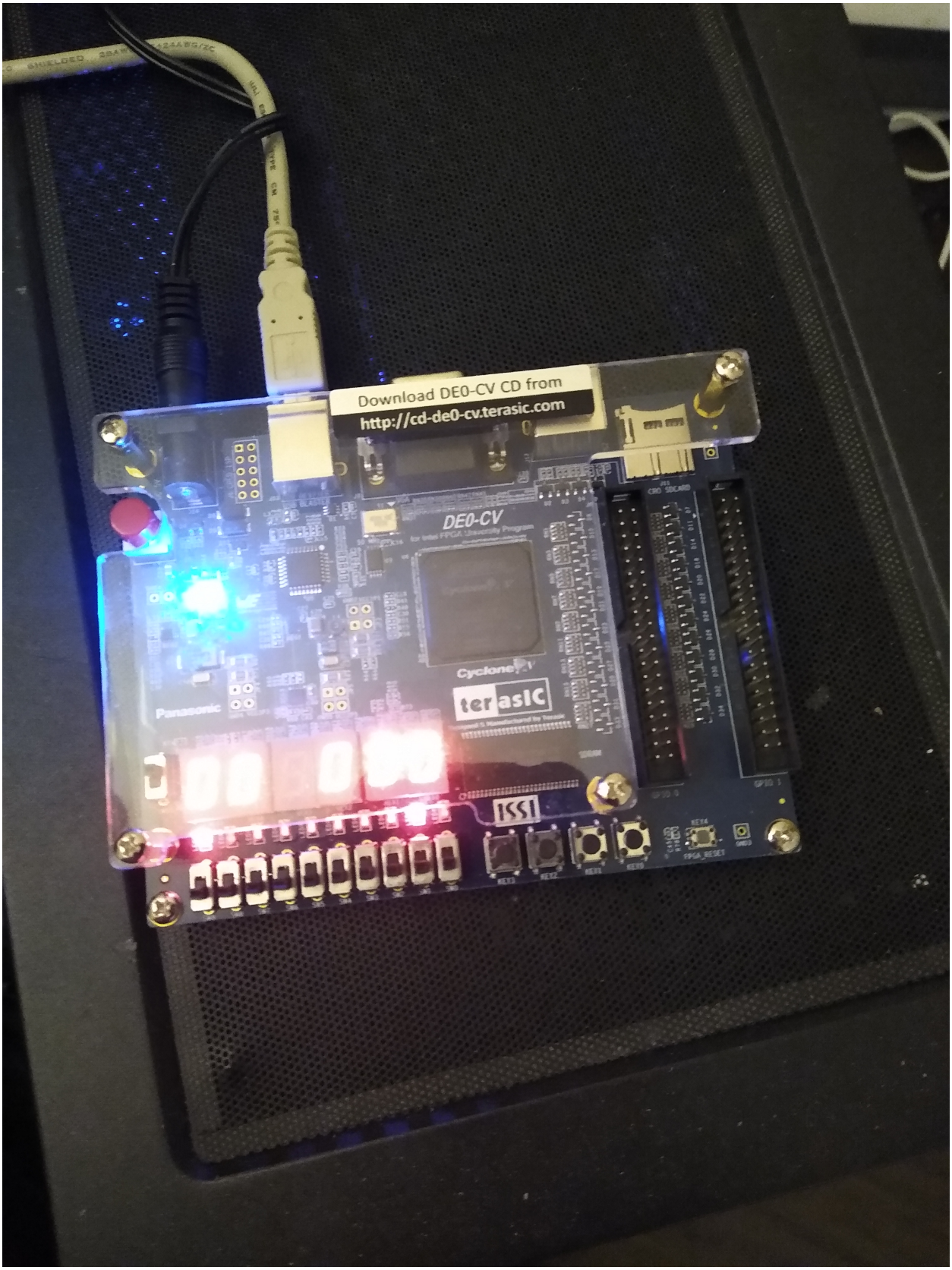

Figure 39: FPGA picture right after reset

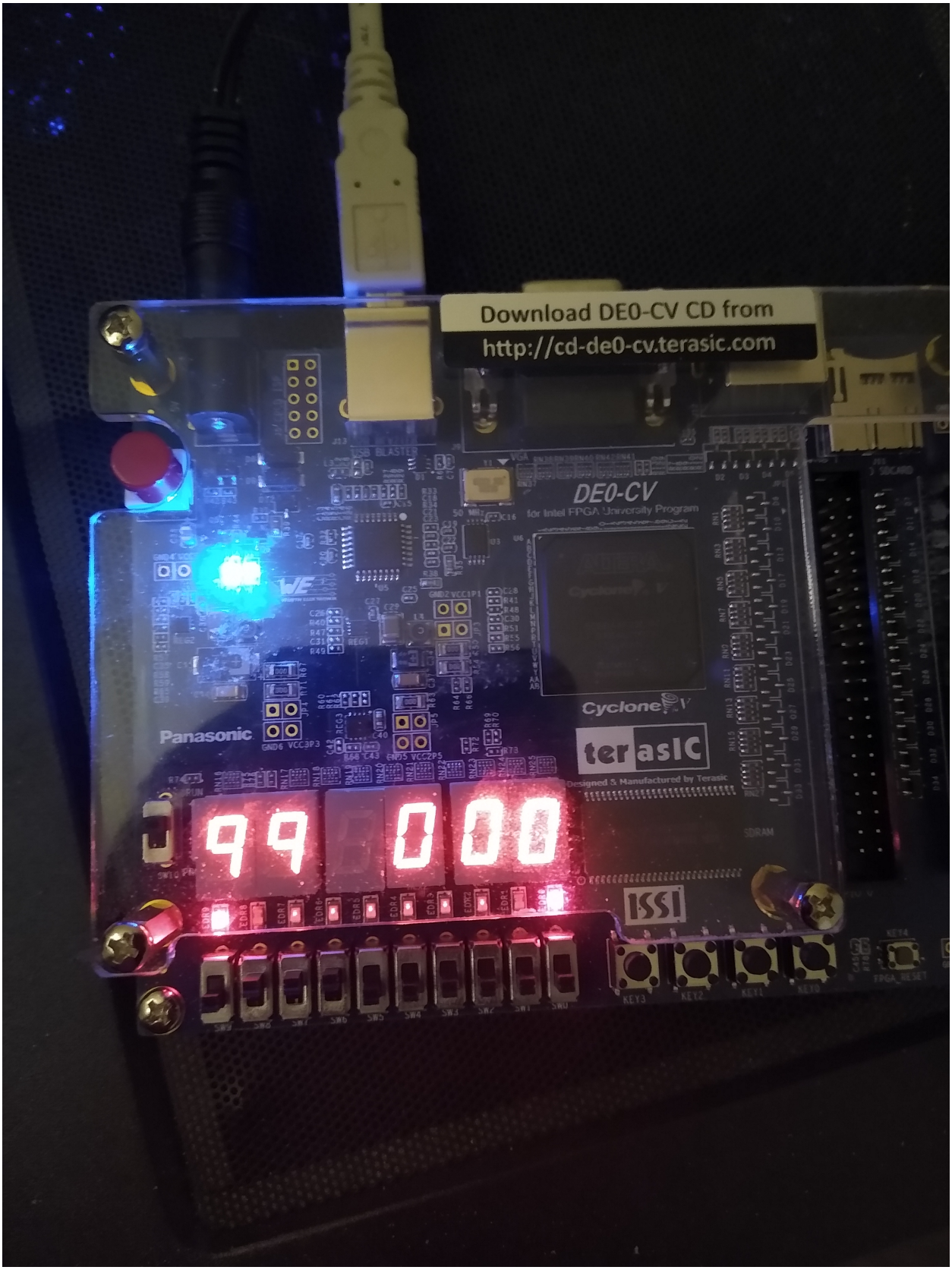

Figure 40: FPGA picture after logging in

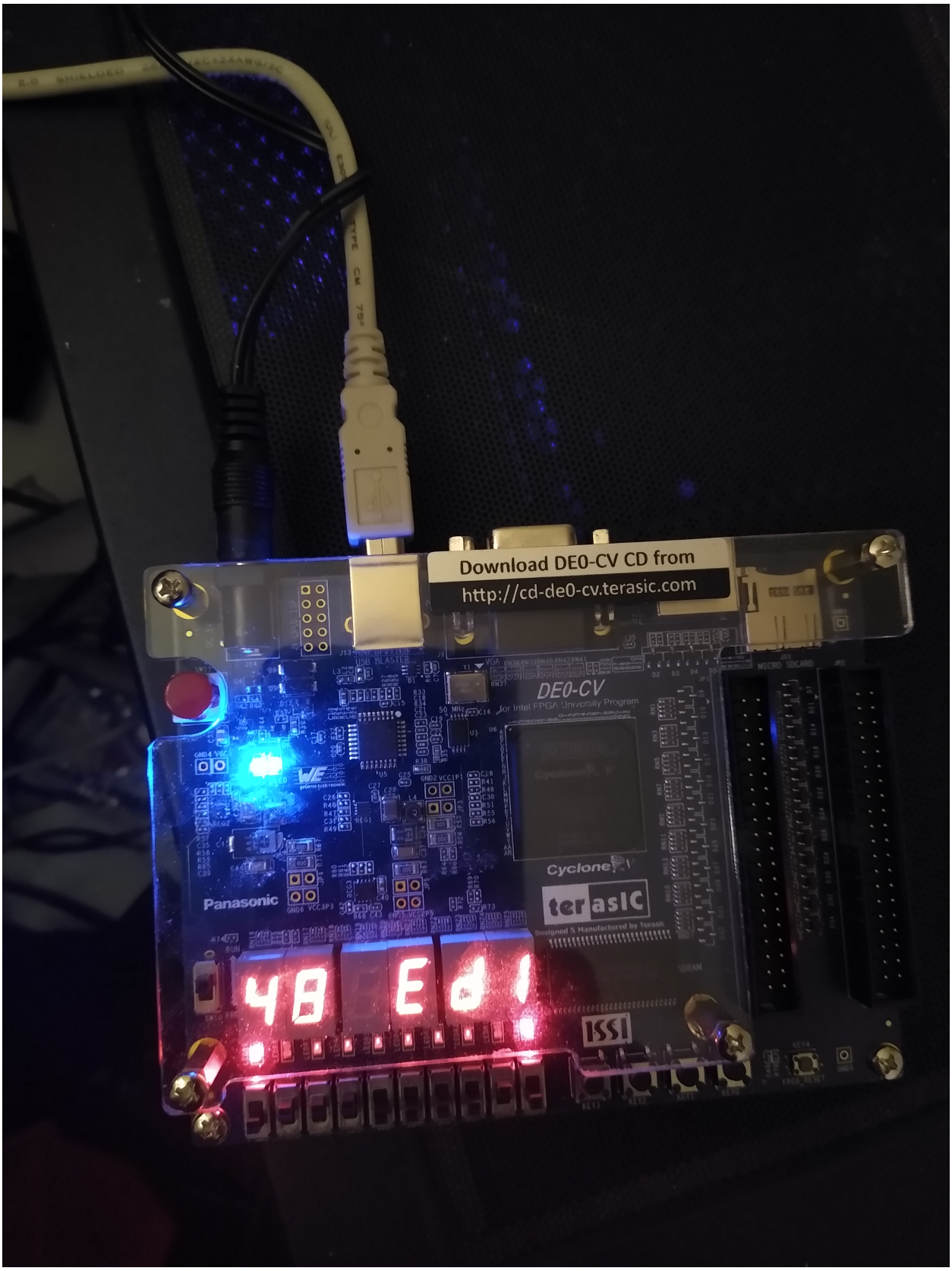

Figure 41: FPGA picture after hitting the random number generator button, entering in 0x1 for the player, and hitting the player load button

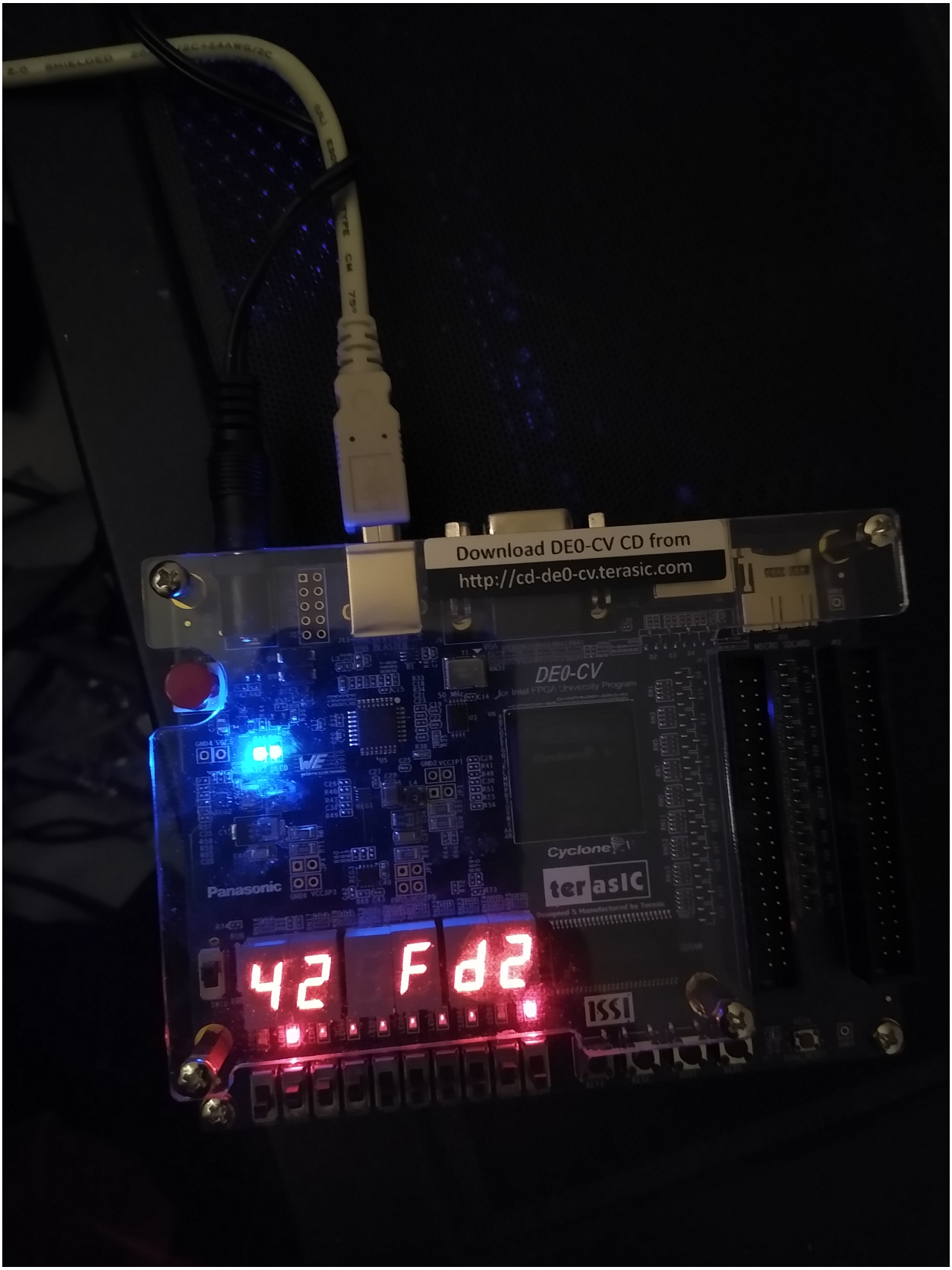

Figure 42: FPGA picture after entering in 0x2 for the player and hitting the player load button

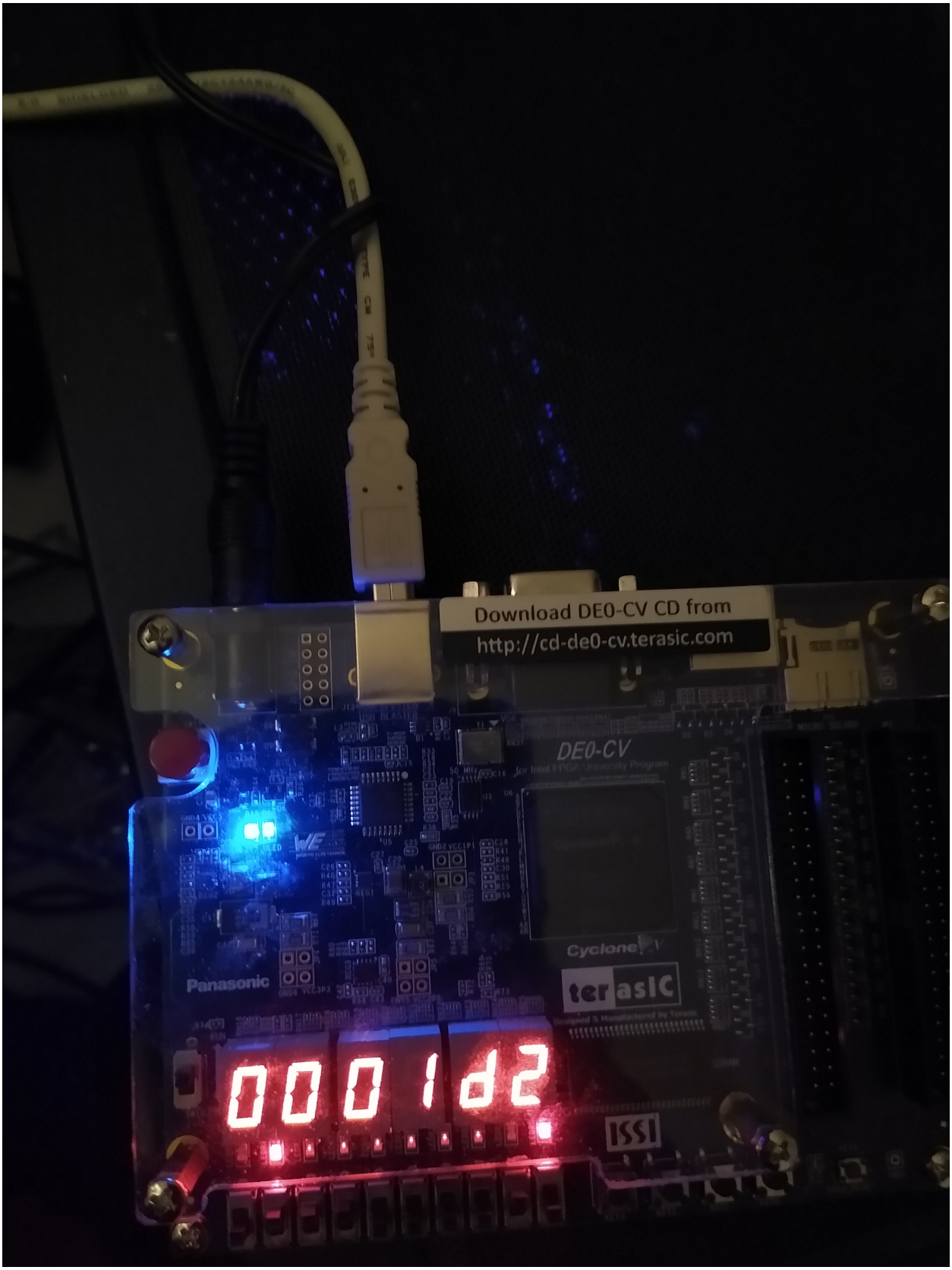

Figure 43: FPGA automatic scoring demo (bonus)

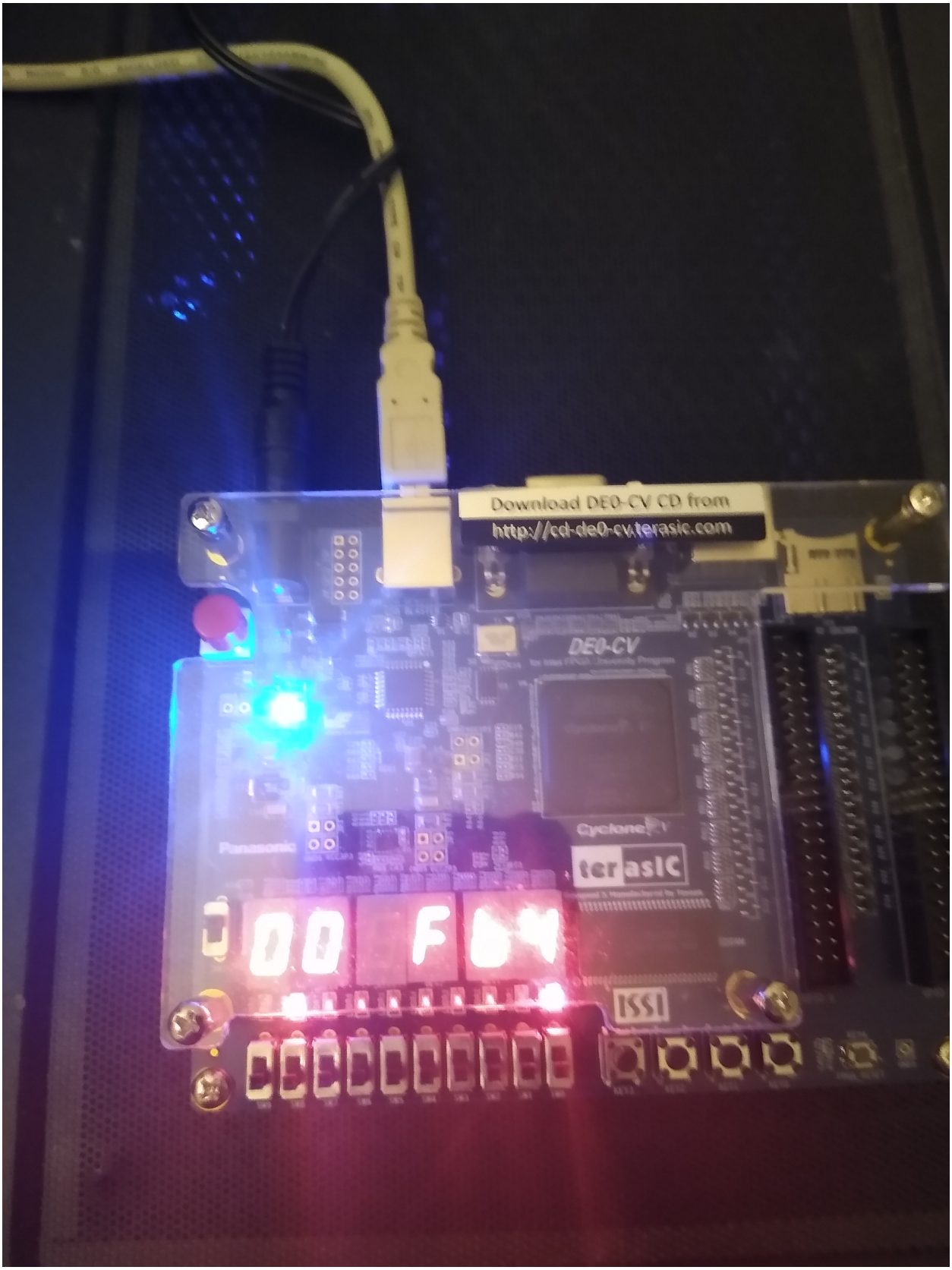

Figure 44: Password changing feature (bonus)

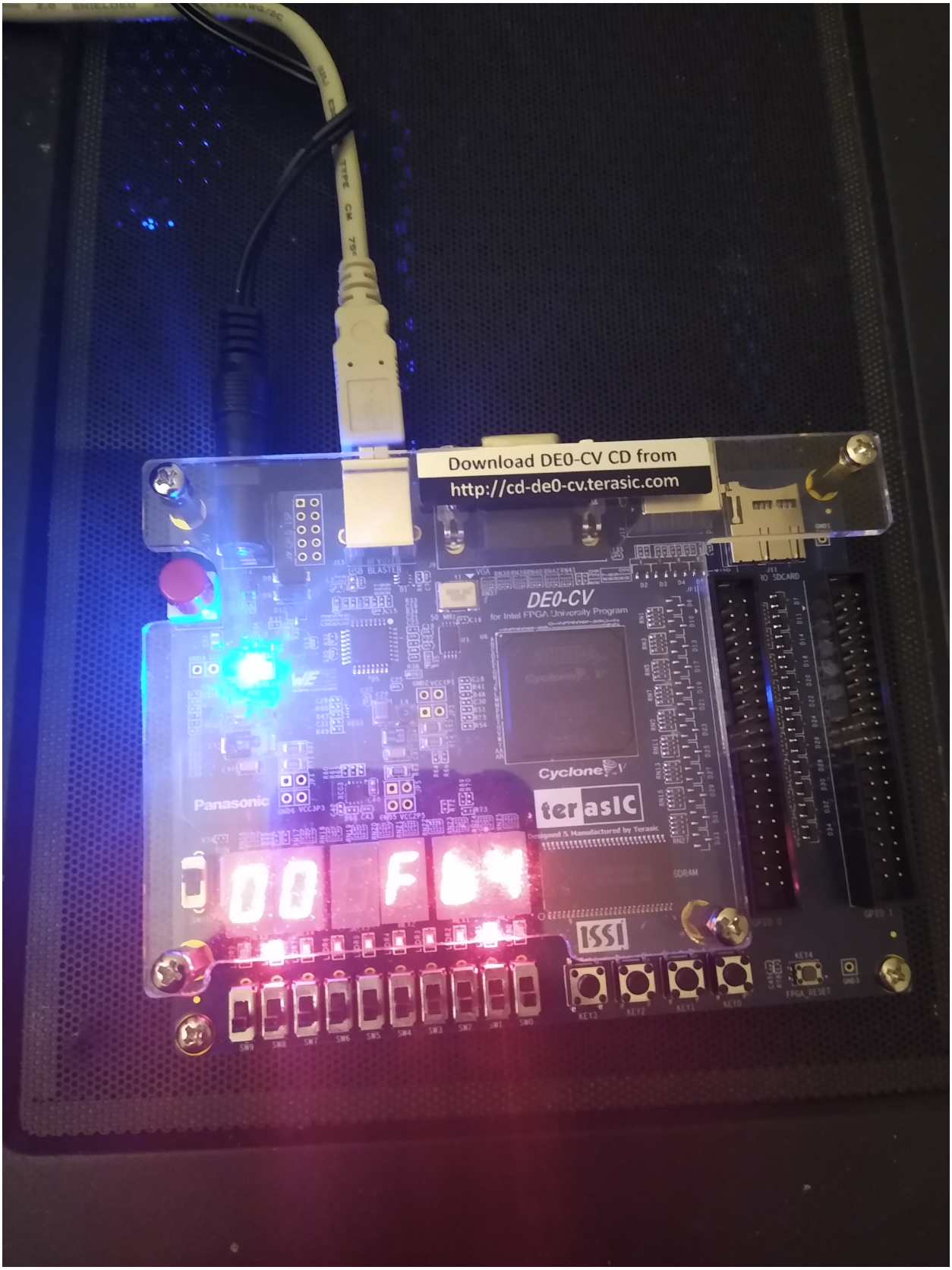

Figure 45: logout feature (bonus)

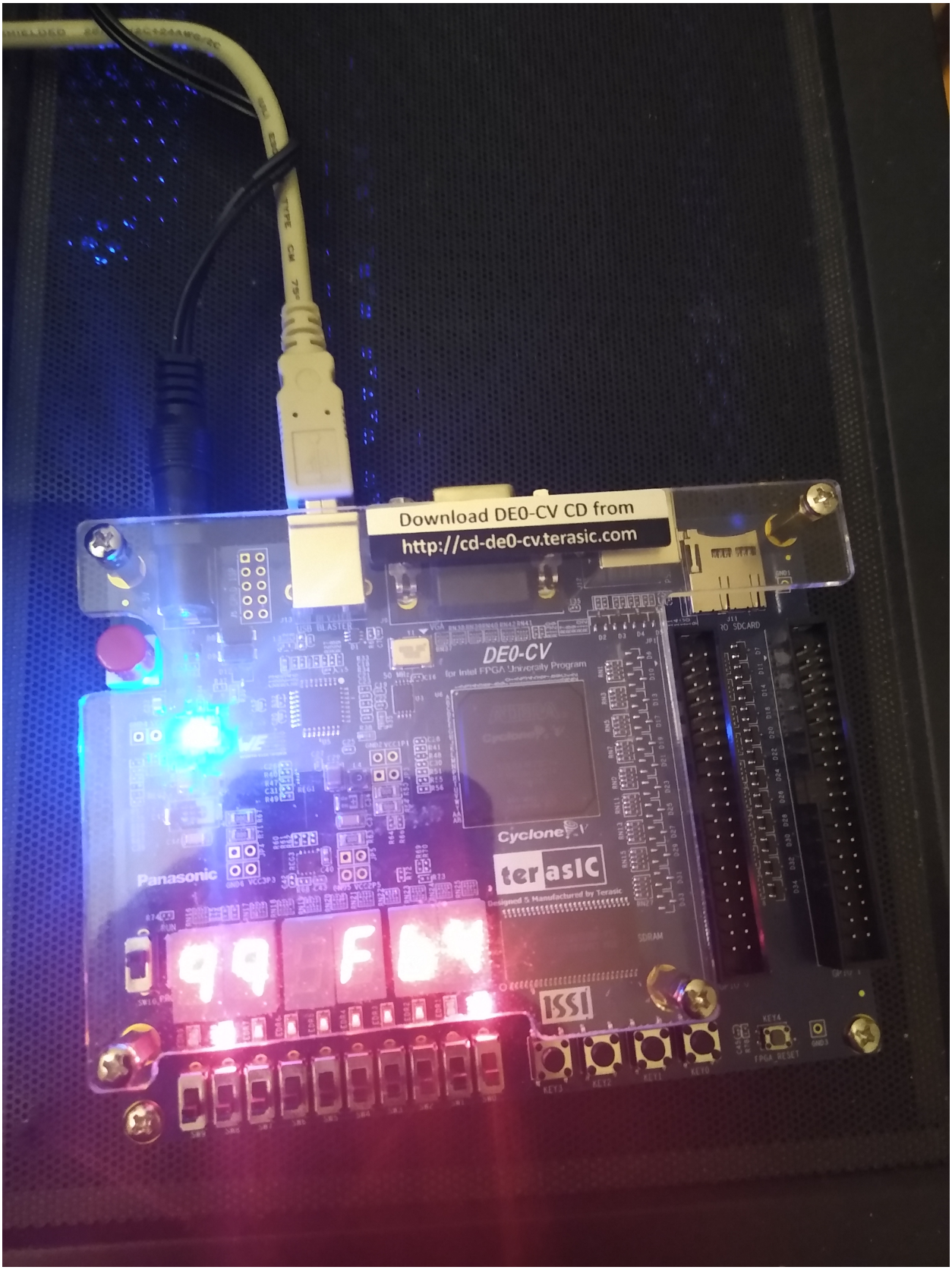

Figure 46: login after password reset

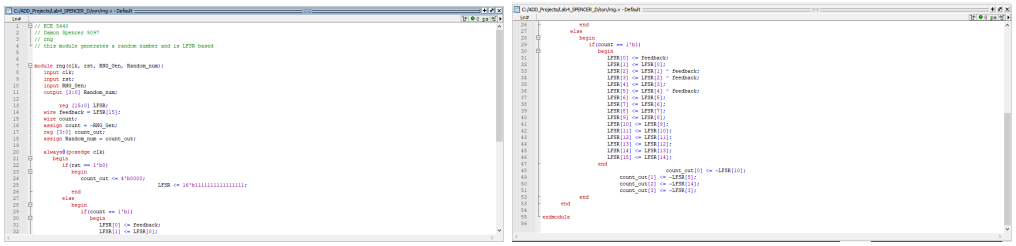

Figure 47: LFSR Random number generator module Verilog code screen capture

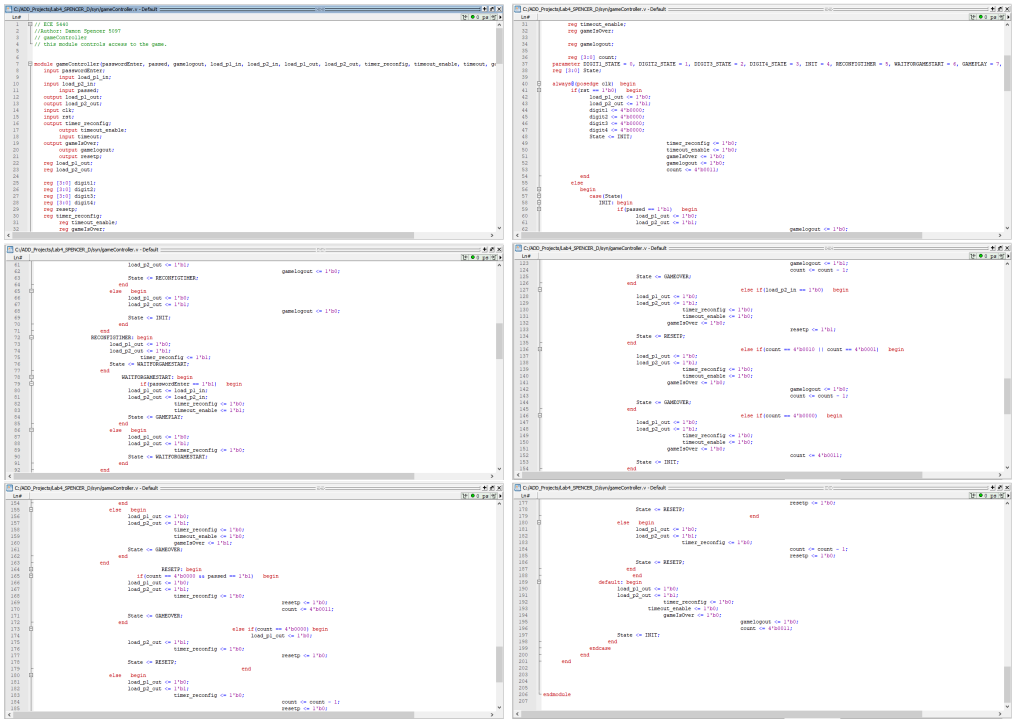

Figure 48: game controller module screen capture

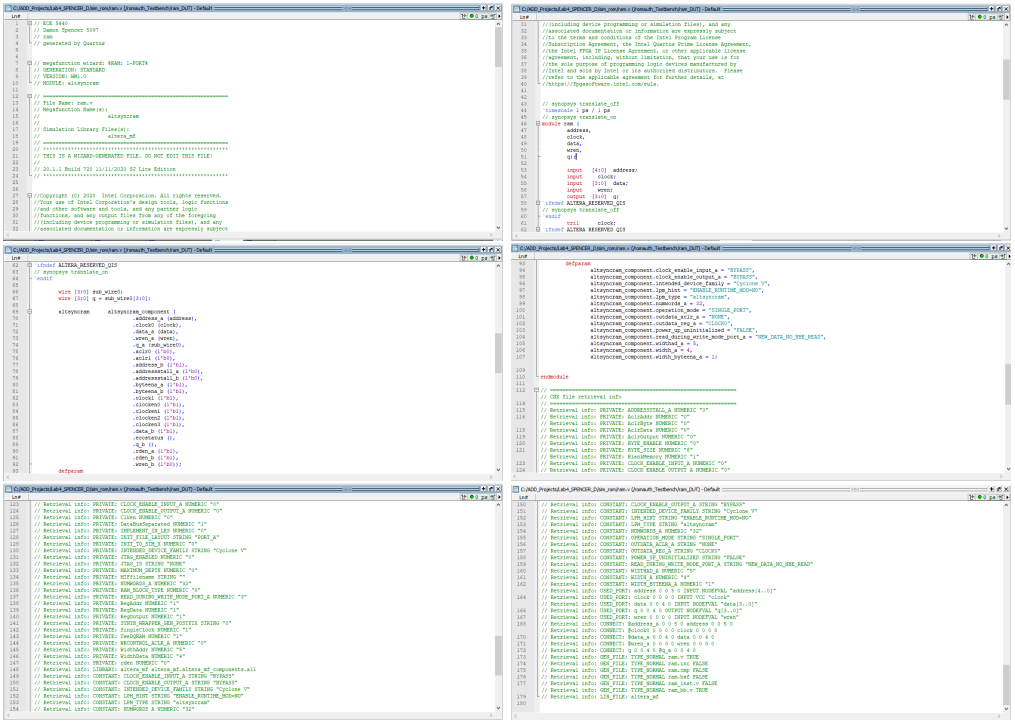

Figure 50: ram module code screen capture

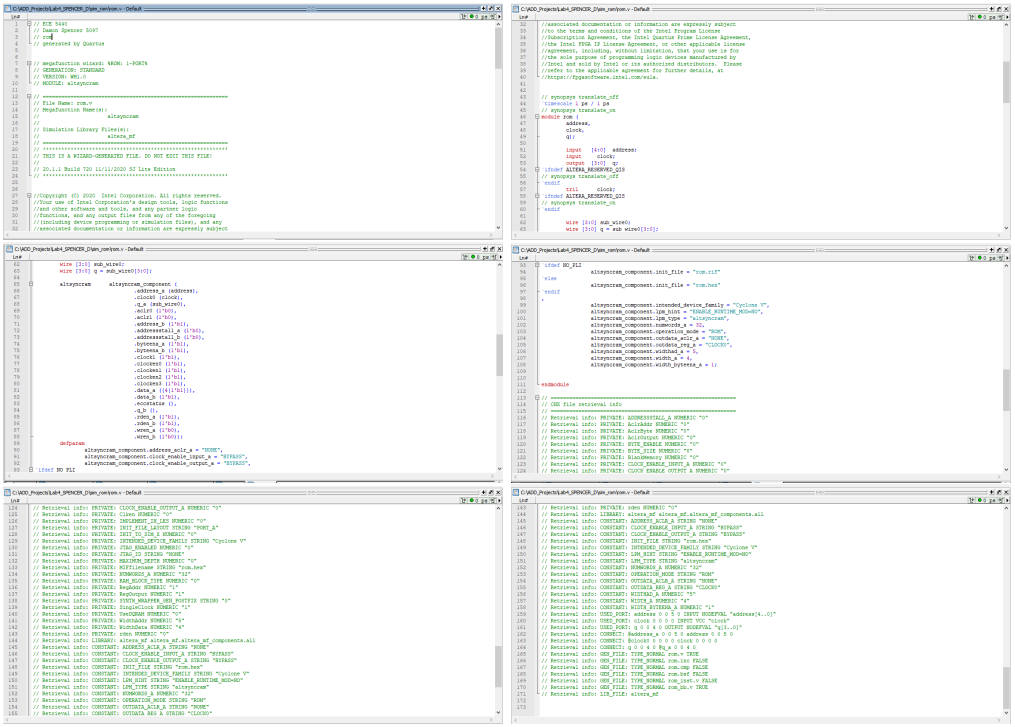

Figure 51: rom module code screen capture

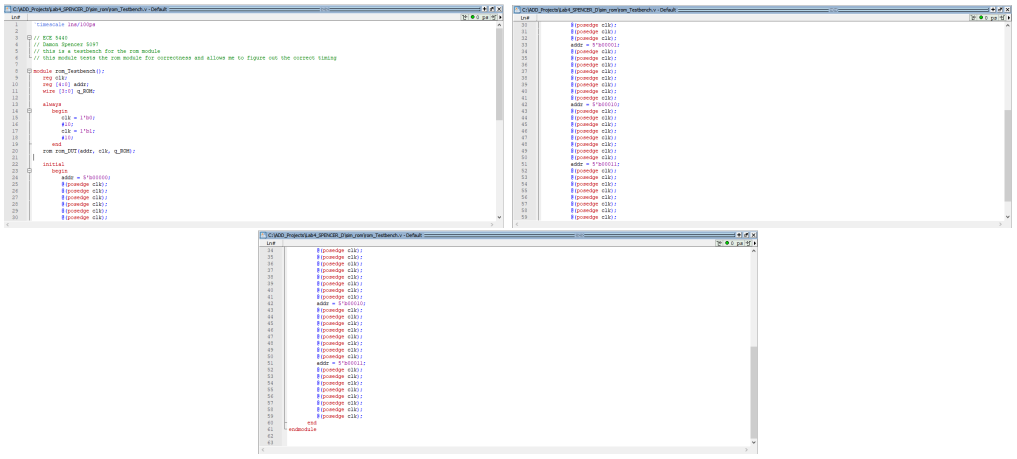

Figure 52: rom module test bench Verilog code

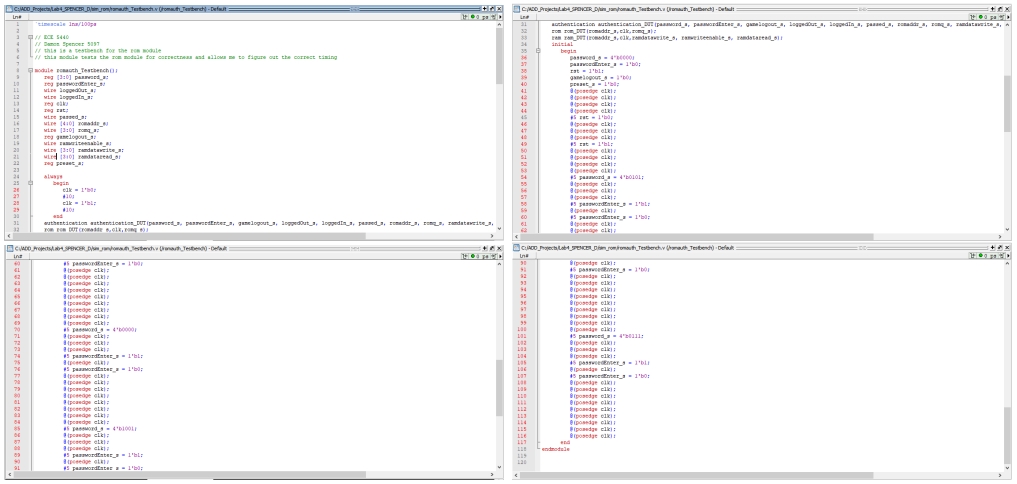

Figure 53: rom and authentication module integration test bench Verilog code

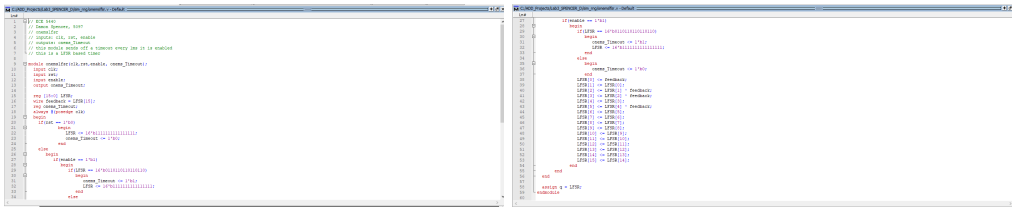

Figure 54: onemslfsr module code screen capture

Figure 55: 1 millisecond LFSR timer module Verilog code

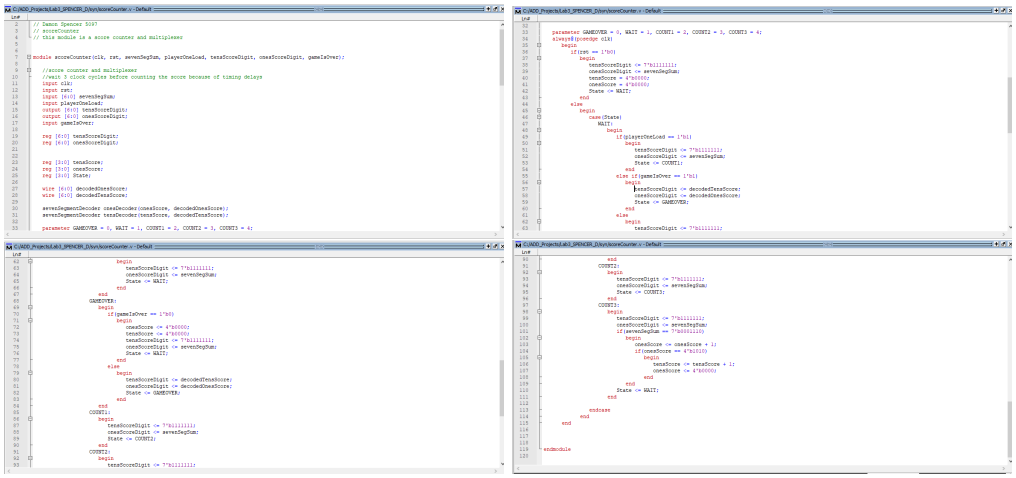

Figure 56: Score counter module Verilog code screen capture

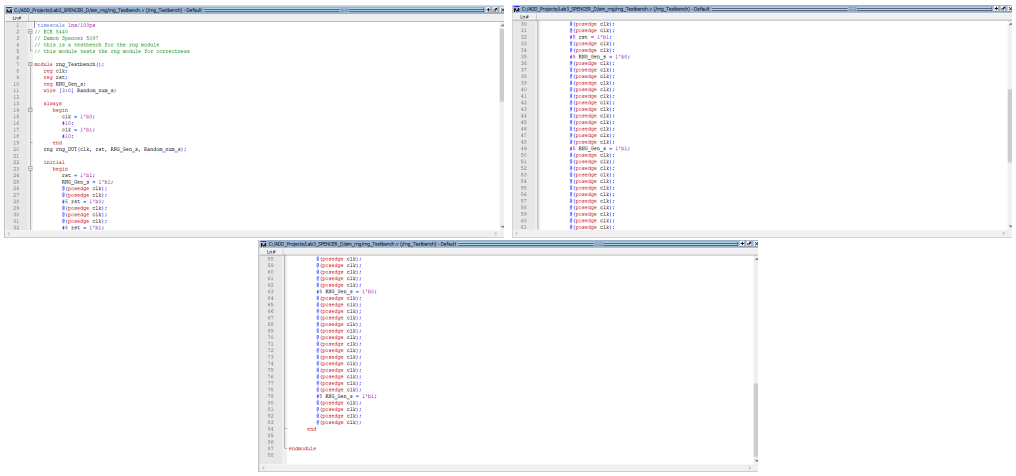

Figure 57: Random number generator test bench module Verilog code screen capture

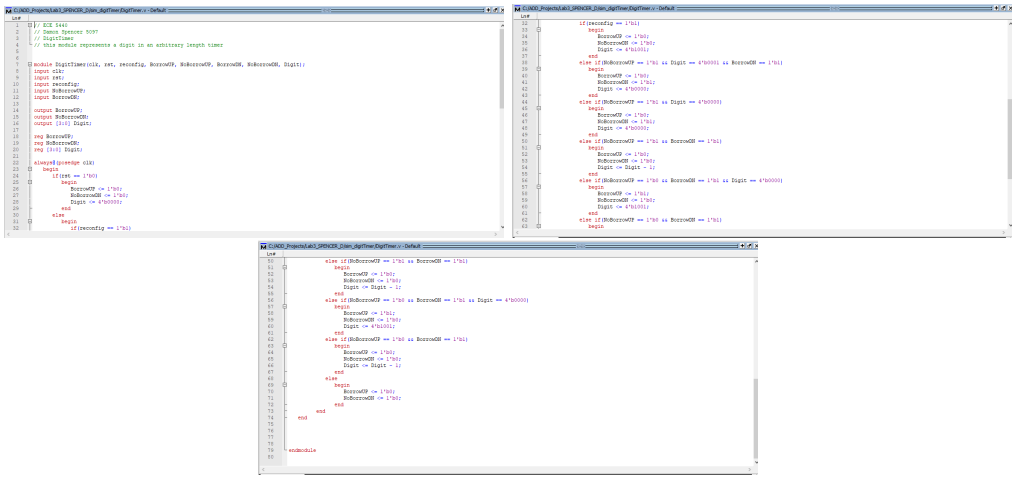

Figure 58: Digit timer module Verilog code screen capture

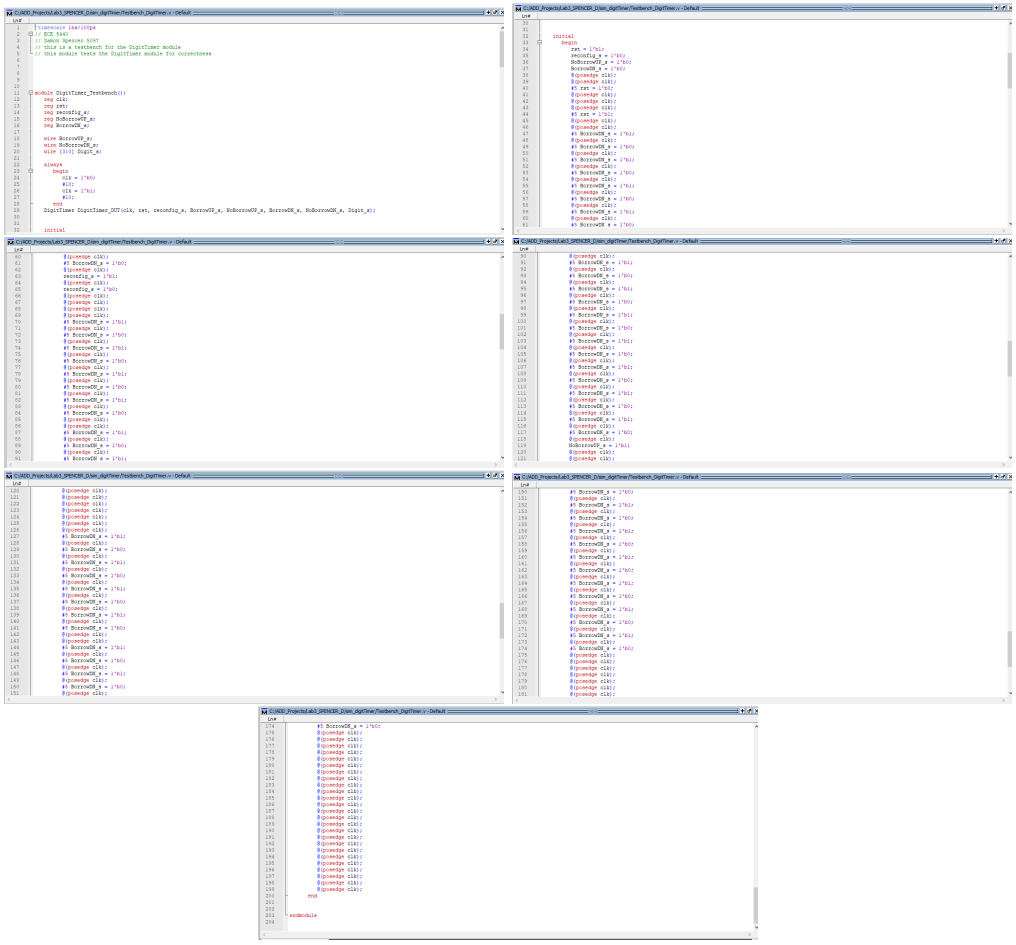

Figure 59: Digit timer module testbench Verilog code screen capture

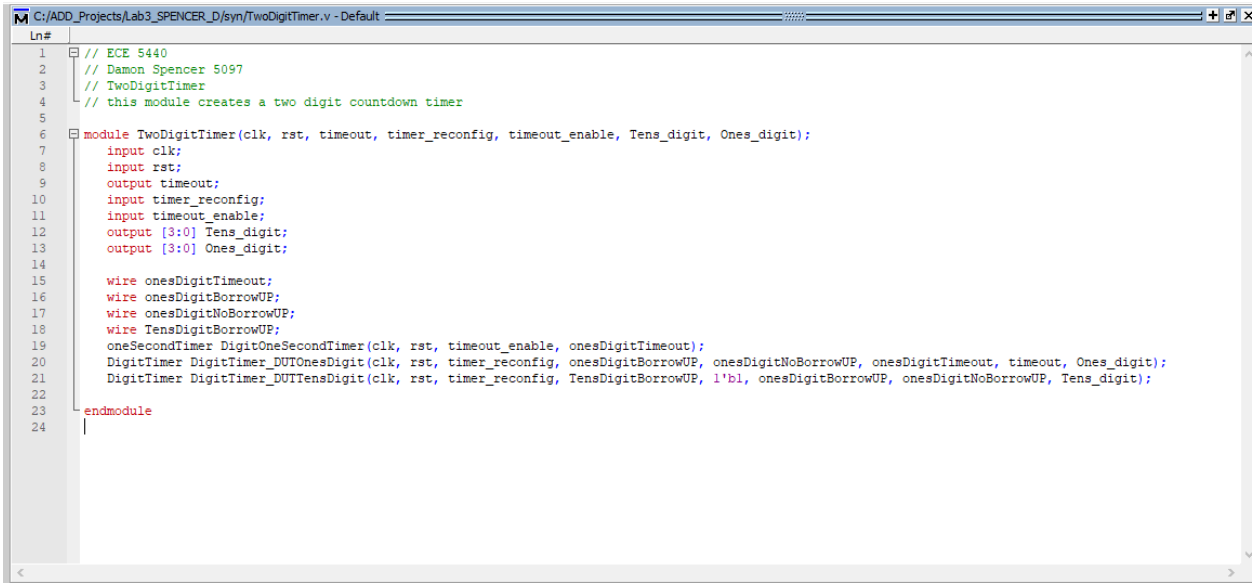

Figure 60: Two Digit timer module Verilog code screen capture

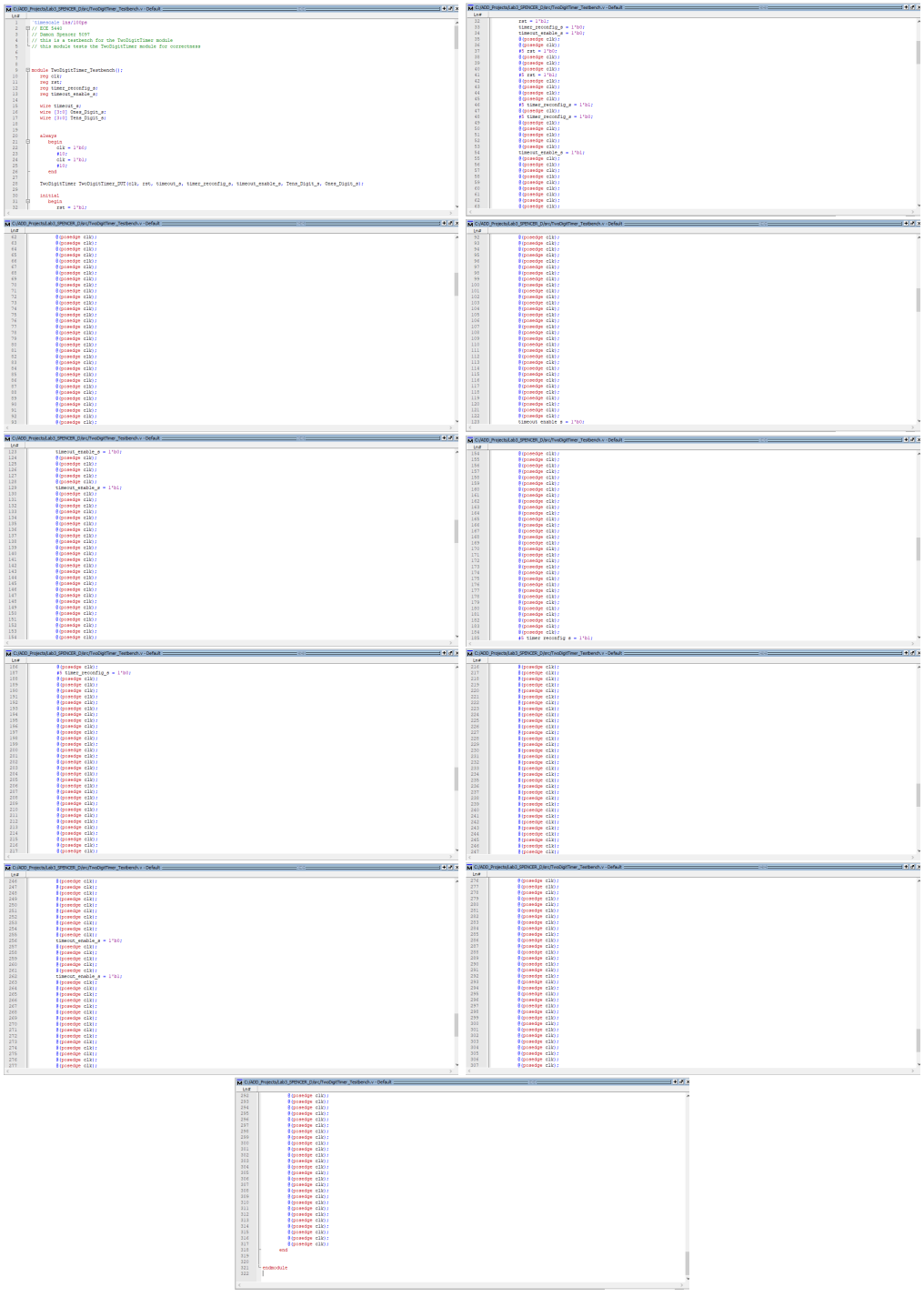

Figure 61: Two Digit timer module testbench Verilog code screen capture

```
1  module oneSecondTimer(clk, rst, enable, oneSecond_Timeout);
2      input clk;
3      input rst;
4      input enable;
5      output oneSecond_Timeout;
6      wire oneHundredms_Timeout;
7      countTo10 to10Counter(clk, rst, oneHundredms_Timeout, oneSecond_Timeout);
8      oneHundredmsTimer oneHundredmsCounter(clk, rst, enable, oneHundredms_Timeout);
9  endmodule
10
```

Figure 62: oneSecondTimer module Verilog code screen capture

Figure 63: oneSecondTimer test bench module code screen capture

```
1  module oneSecondTimer(clk, rst, enable, oneSecond_Timeout);
2      input clk;
3      input rst;
4      input enable;
5      output oneSecond_Timeout;
6      wire oneHundredms_Timeout;
7      countTo10 to10Counter(clk, rst, oneHundredms_Timeout, oneSecond_Timeout);
8      oneHundredmsTimer oneHundredmsCounter(clk, rst, enable, oneHundredms_Timeout);
9  endmodule
10
```

Figure 64: oneHundredmsTimer module code screen capture

```
1  module countTo10(clk,rst,enable, countTo10_Timeout);
2     input clk;
3     input rst;
4     input enable;
5     output countTo10_Timeout;
6
7     reg [3:0] count;
8     reg countTo10_Timeout;
9     always @(posedge clk)
10       begin
11          if(rst == 1'b0)
12             begin
13                count <= 16'b0000;
14                countTo10_Timeout <= 1'b0;
15             end
16          else
17             begin
18                if(enable == 1'b1)
19                   begin
20                      if(count == 1) //need to only count to 1 to have the timer go off every other cycle
21                         begin
22                            count <= 16'b0000;
23                            countTo10_Timeout <= 1'b1;
24                         end
25                      else
26                         begin
27                            count <= count + 1;
28                            countTo10_Timeout <= 1'b0;
29                         end
30                   end
31                else
32                   begin
33                      countTo10_Timeout <= 1'b0;
34                   end
35             end
36       end
37
38
39 endmodule
40
```

Figure 65: countTo10 module code screen capture

```
1  module countTo100(clk,rst,enable, countTo100_Timeout);
2     input clk;
3     input rst;
4     input enable;
5     output countTo100_Timeout;
6
7     reg [6:0] count;
8     reg countTo100_Timeout;
9     always @(posedge clk)
10       begin
11          if(rst == 1'b0)
12             begin
13                count <= 16'b0000;
14                countTo100_Timeout <= 1'b0;
15             end
16          else
17             begin
18                if(enable == 1'b1)
19                   begin
20                      if(count == 2) //need to only count to 2 to have the timer go off every three cycles
21                         begin
22                            count <= 16'b0000;
23                            countTo100_Timeout <= 1'b1;
24                         end
25                      else
26                         begin
27                            count <= count + 1;
28                            countTo100_Timeout <= 1'b0;
29                         end
30                   end
31                else
32                   begin
33                      countTo100_Timeout <= 1'b0;
34                   end
35             end
36       end
37
38
39 endmodule
40
```

Figure 66: countTo100 module code screen capture

```
Ln#
 1  module LoadRegister(D_in, D_out, clk, rst, Load);
 2      input [3:0] D_in;
 3      output [3:0] D_out;
 4      input clk, rst;
 5      input Load;
 6      reg [3:0] D_out;
 7
 8      always@(posedge clk)
 9          begin
10              if(rst == 1'b0)
11                  begin
12                      D_out <= 4'b0000;
13                  end
14              else
15              //normal operation after reset
16                  begin
17                      if(Load == 1'b1)
18                          begin
19                              D_out <= D_in;
20                          end
21                  end
22          end
23
24
25  endmodule
26
```

Figure 67: Load register module Verilog code screen capture

Figure 68: Load register test bench module Verilog code screen capture

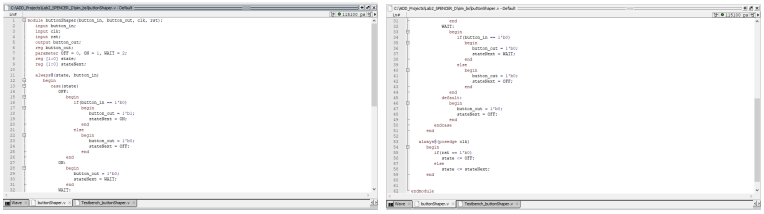

Figure 69: Button shaper module Verilog code screen capture

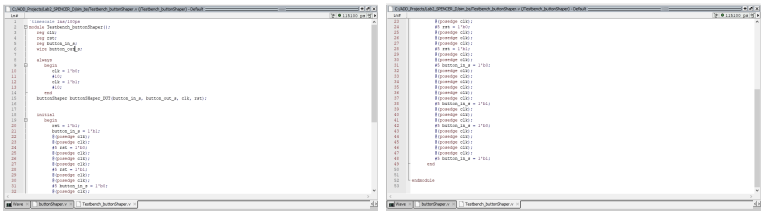

Figure 70: Button shaper test bench module Verilog code screen capture

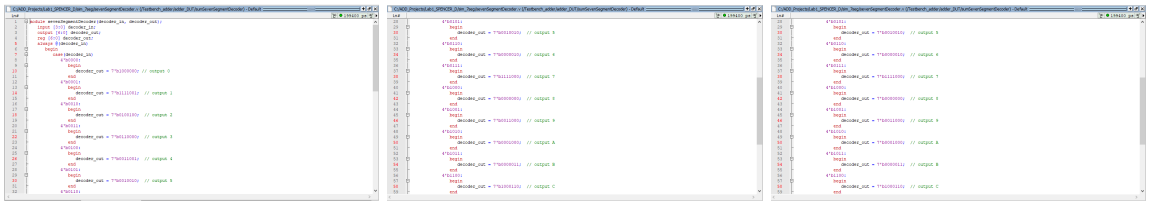

Figure 71: Seven segment decoder module Verilog code screen capture

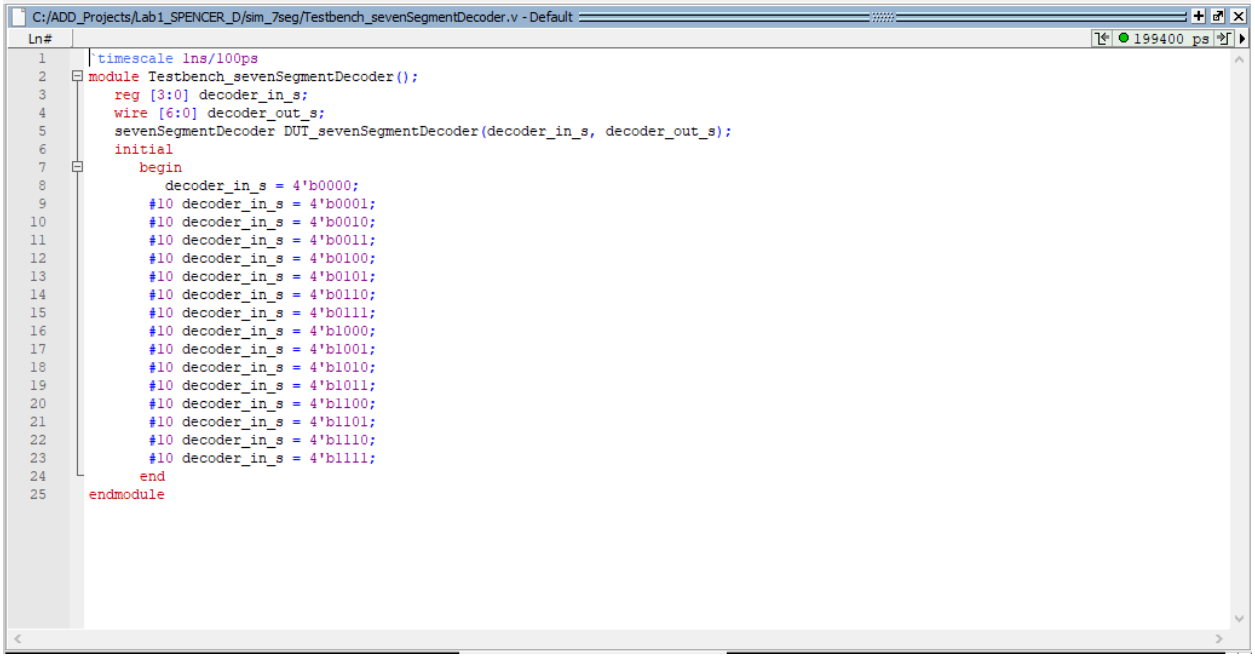

Figure 72: Seven segment decoder test bench module Verilog code screen capture

```
Ln#                                                                    47400 ps
1  module adder(player1_in, player2_in, sum, wongame, didnotwingame);
2     input [0:3] player1_in;
3     input [0:3] player2_in;
4     output [0:3] sum;
5     output wongame;
6     output didnotwingame;
7
8     reg [0:3] sum;
9     reg wongame;
10    reg didnotwingame;
11
12    always@(player1_in, player2_in)
13       begin
14          sum = player1_in + player2_in;
15          case(sum)
16             4'b1111:
17                begin
18                   wongame = 1;
19                   didnotwingame = 0;
20                end
21             default:
22                begin
23                   wongame = 0;
24                   didnotwingame = 1;
25                end
26          endcase
27       end
28 endmodule
29
```

Figure 73: Adder module Verilog code screen capture

```
Ln#                                                                    47400 ps
1  `timescale 1ns/100ps
2  module Testbench_adder();
3     reg [0:3] player1_in_s;
4     reg [0:3] player2_in_s;
5     wire [0:3] sum_s;
6     wire wongame_s;
7     wire didnotwingame_s;
8
9
10    adder adder_DUT(player1_in_s, player2_in_s, sum_s, wongame_s, didnotwingame_s);
11    initial
12      begin
13         player1_in_s = 4'b1101; player2_in_s = 4'b0010;
14         #10 player1_in_s = 4'b0000; player2_in_s = 4'b0010;
15         #10 player1_in_s = 4'b0000; player2_in_s = 4'b0000;
16         #10 player1_in_s = 4'b1111; player2_in_s = 4'b0010;
17         #10 player1_in_s = 4'b1001; player2_in_s = 4'b0010;
18      end
19 endmodule
20
```

Figure 74: Adder test bench module Verilog code screen capture