LAB 4: ROM-BASED GAME ACCESS CONTROL ON FPGA

ECE 6370

Zhang, David J
2213233

Introduction

This lab introduces three new mechanisms on top of the previous iterations of the FPGA based mental binary game. The log-in feature using the second player's switches still functions, and now the **log-out feature can be done by pressing the player's LOAD button whenever the timer is not running**. The load feature, which only loads a player's number when their pushbutton is pressed, is retained however since there is no longer a second player their load button functionality has been removed. This iteration also introduces a 99 second timer that will count down at the start of the game and will lock out the player once it reaches zero. In addition, a scoring feature is added that will display at the end of the game (timer reaches zero) indicating how many correct matches the player had (combinations of RNG and player input that equate to F). Finally, if a player wishes to change their password to some other 4-digit combination, at any point when the game timer is not running they can press the RNG button to log out of the game and begin setting a new 4-digit password using the password entry system.

System Architecture

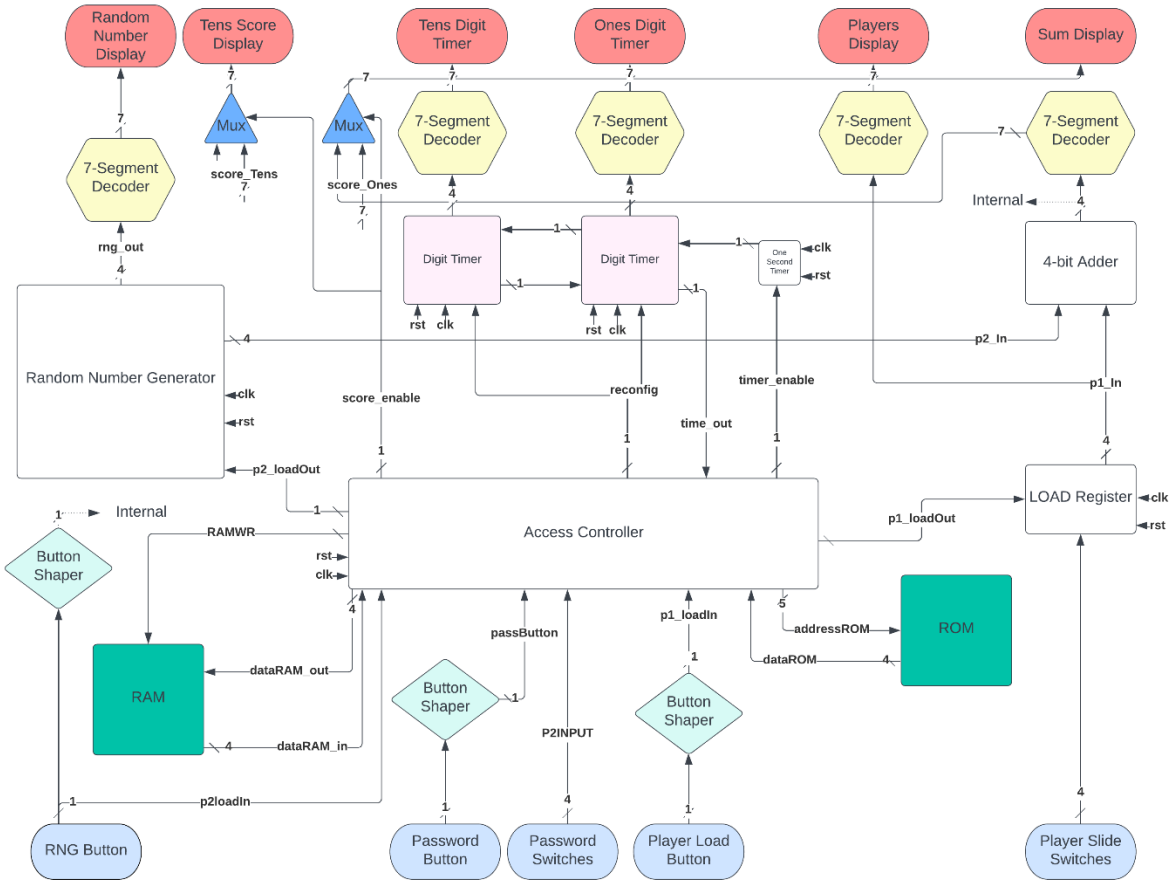
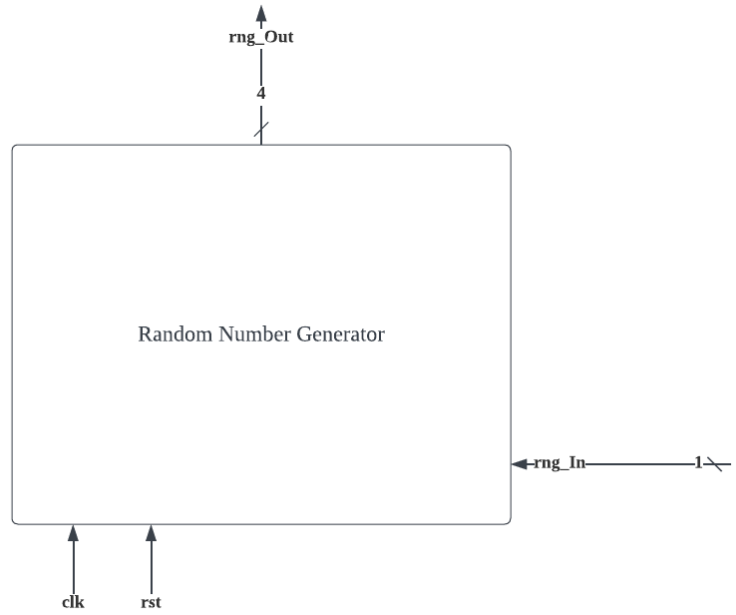


Figure 1: Overall System Architecture

The above figure depicts the top-level system architecture for the game. Along with all the modules of previous games, there are 4 new modules: Mux, RNG, 1secTimer, and digitTimer. In addition, password management is now handled by the ROM/RAM instead of the Access Controller.

Mux is a simple module that acts as a two-to-one multiplexer. It takes in two separate 7-bit inputs and outputs a 7-bit output to one of the 7-segment LED's. The select signal comes from the Access Controller and is a single bit that is 0 for the left input and 1 for the right input. This module is needed to share resources—there are not enough 7-segment LED's to display everything needed, so the SUM display is shared with the ones digit of the final score.



RNG is a module that takes in an unshaped 1-bit signal from a push button and outputs a random 4-bit number from 0 to F. As with every sequential logic module, it also takes in a global clock and reset signal. This module functions by **using a Linear Feedback Shift Register**. It inputs and shifts values every clock cycle it detects that rng_In is high, so for this an unshaped button signal is needed to ensure that the number of clock cycles read is random, as humans cannot press buttons fast enough to consistently get certain outputs.

Figure 2: Module Depiction of RNG

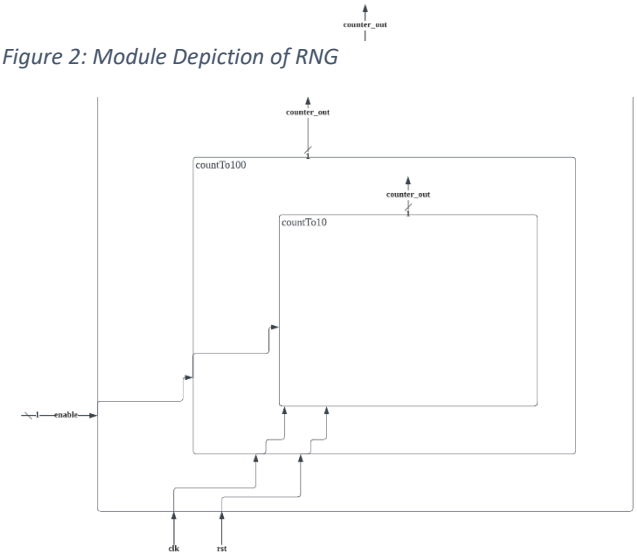


Figure 3: Module Depiction of OneSecondTimer

1secTimer is a module that consists of 2 nested modules: countTo100 and countTo10. However, all of these modules function similarly and are only nested for speed of operation. 1secTimer is functionally the same as countTo100 and countTo10, except that it counts to 50,000 instead. This is done to ensure that in total, a count of 50,000,000 is reached (50,000 x 100 x 10). At a clock speed of 50 MHz, counting to 50,000,000 clock cycles effectively counts for 1 real second. Like the RNG module, the highest module that counts to 50,000 is **implemented using an LFSR to save resources.**

1secTimer (and by extension, countTo10 and countTo100) takes in a long

active high signal called enable that puts the module in operational mode. In this mode, it takes in a 1-bit input that will increment an internal counter until it reaches the specified maximum (10, 100 or 50,000) at which point it will output a pulsed 1-bit signal high. Importantly, the most nested module takes in an input directly from the global clock. Its output is then fed as an input to its upper-nested version. As an example, since countTo10 is nested inside countTo100, then countTo100 will only increment when countTo10 has reached its stated maximum (10). This way, when countTo100 outputs, it has actually counted 100x10 clock cycles, not 100.

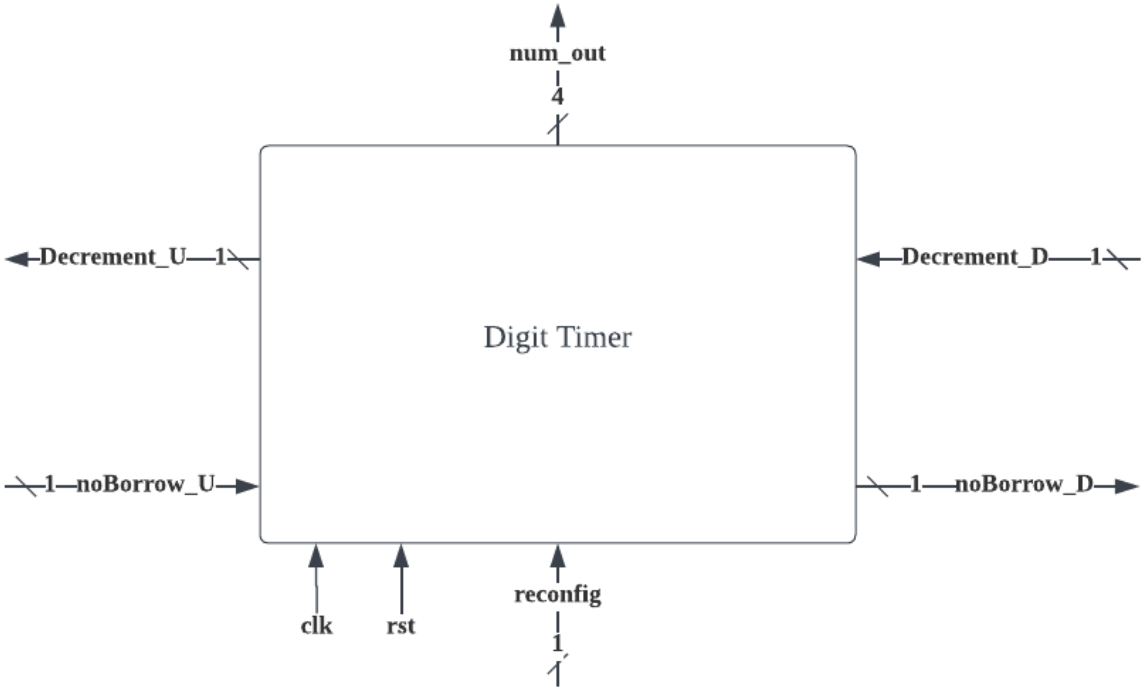


Figure 4: Module Depiction of Digit Timer

digitTimer is a module that has 2 signals that come and leave externally and 4 signals that it uses to interface with other iterations of itself. This module is non-digits place specific, meaning that it can be used for a tens digit, a ones digit, a hundreds digit and so on non-specifically. The 2 “external” signals are a 1-bit active high signal called reconfig and a 4-bit binary number output, num_out. The reconfig signal comes from the access controller and is only sent upon Game Start. The 4-bit num_out represents the number that the digitTimer is holding. Importantly, since these modules are for a timer, the 4-bit maximum is defined in the module to be decimal 9, not the true 4-bit maximum of F as that would be unwieldy and awkward for a timer.

The 4 “interface” signals are 1-bit active high signals that are used to interface with other iterations of itself. DecrementD and DecrementU are pulsed signals that come from Downstream or go Upstream to tell the receiver to decrement its internal register. noBorrowD and noBorrowU are long signals that similarly come from Downstream or go Upstream to tell the receiver that it cannot “lend” a 1 to its messenger. The logic of this module is modeled after how we as humans perceive subtraction—for example, for 22 – 8, a human will look at the ones place and see that 8 cannot be subtracted from 2,

so we look to the tens digit, see that we can “lend” a digit to the ones place and do so to create a mathematical operation that makes sense: 12-8.

By default, the noBorrowD signals (outputs) are set long high when reconfig is pressed—in other words, when a 9 is loaded to the internal register the module tells its downstream neighbor that it has numbers to lend. When the internal register counts down to 0 as a result of DecrementD signals, it checks if noBorrowU (inputs) is low—the signal from its upstream neighbor indicating that it CAN borrow—and if it is, it sends a DecrementU (outputs) signal and resets its own internal counter to 9. If it is not, then it holds its register at 0 and outputs noBorrowD high—indicating to its downstream neighbor that there is no higher order digits that it can borrow from.

Importantly, in Figure 1 the leftmost Digit Timer does not have its DecrementU or noBorrowD signal drawn. As this is the most significant digit of the timer, the tens spot, noBorrowD is set to always 1 to ensure that it does not try to lend from a fictional hundreds spot that doesn’t exist. DecrementU is not used anywhere then, so where it is mapped is not important.

For the Lab 4 bonus features, RAM and ROM represent the onboard memory that stores the values of the password for log in entry. Initially, the 4-digits are stored in ROM but if desired a new password can be written into RAM which the system will use as the new password in subsequent log ins. Each memories respective dataBus signal is a 4-bit signal that represents the value of the digit from 0 to F, and each address signal represents the value of the address for each number. For a 4-digit signal, only 2 bits are needed for addressing but since the smallest size of ROM/RAM to instantiate is 32 addresses the top 3 bits are zero-padded. RAMWR is a necessary signal for the RAM only to tell it whether the databus contains data to be written or read into RAM, respectively.

Other modules not listed are legacy code from the previous iteration, but in short:

Button Shaper outputs a single clock cycle active high signal whenever it detects that its input has gone low. It will only send out another pulsed signal after input has gone back to high, then low again.

Load Register passes its 4-bit input to 4-bit output whenever it receives a 1-bit load signal from the Access Controller. This load signal is shaped from Button Shaper and is in reality the push of the Player’s LOAD button.

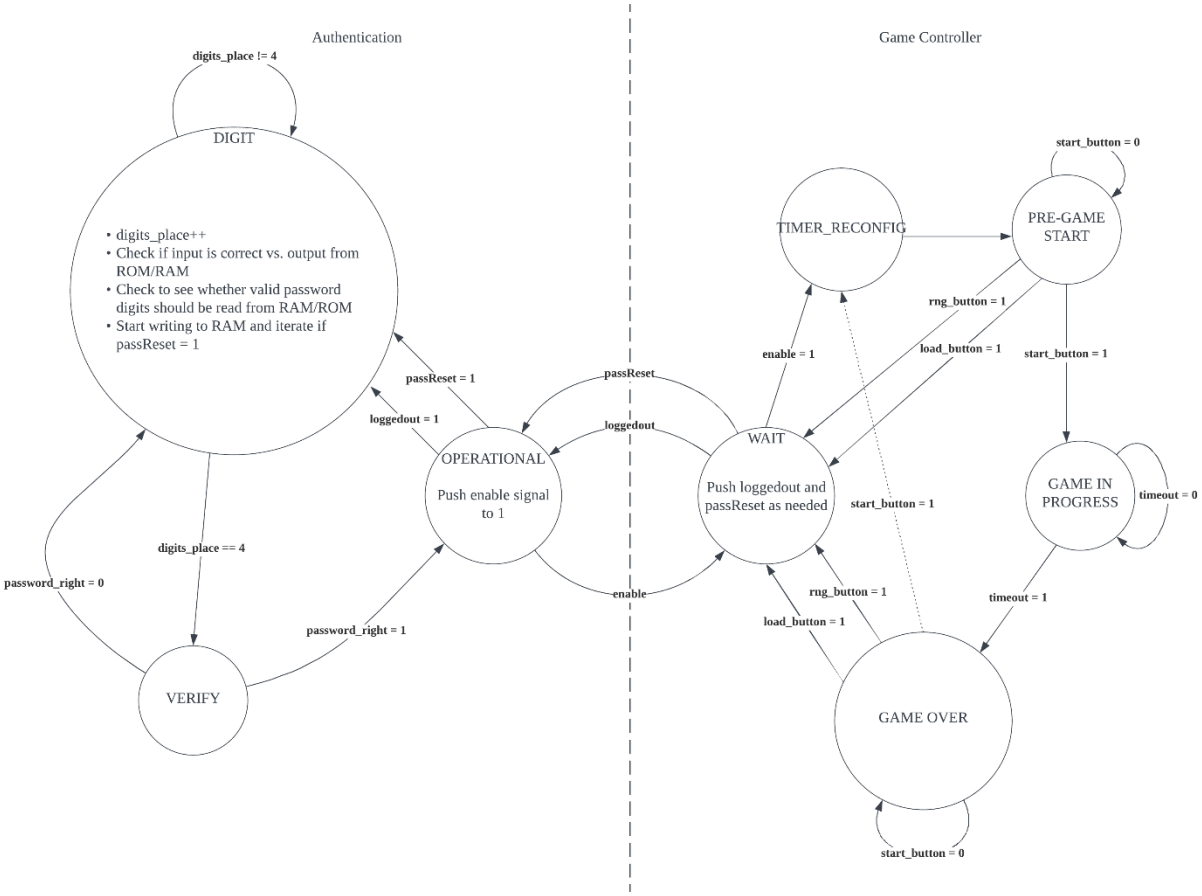


Figure 5: Finite State Machine for Split Access Controller

Depicted in Figure 5 is the modified FSM for the Split Access Controller. **The responsibilities of the Access Controller have been split into Authentication and Game Controller.** Authentication handles the log in and password change features, while Game Controller handles all game functions. **Signals that cross the dotted line do not represent state change conditions, but signals that are passed between modules**. The addition of a new WAIT state for the Game Controller is necessary to handle initialization, since now both modules FSM's are technically independent from each other.

In TIMER RECONFIG, all player inputs are blocked, the reconfig signal is set high and immediately moves to PRE-GAME START. For implementation of the bonus feature, the internal signal score_enable is set to 0 during TIMER RECONFIG for correct looping behavior. During PRE-GAME START, reconfig is set back low to ensure that it was only high for one clock cycle. In addition, it waits until Button Push is high—the shaped signal coming from the Game Start button—before setting timer_enable to high and moving to GAME IN PROGRESS.

During GAME IN PROGRESS, player inputs are finally unblocked and all inputs are passed to outputs. Once the timeOut signal has gone high—from the timer reaching 0 seconds—the system moves to the last state, GAME OVER. In GAME OVER, timer_enable is set low to turn off the function and all player inputs are blocked again. Importantly, in this stage the bonus feature must be implemented: an internal signal named score_enable is set to 1, changing the output of the muxes to the score counter

instead of the sum display. The system remains in GAME OVER until it detects that the Game Start button has been pressed again, after which it moves back to TIMER RECONFIG and the cycle begins anew.

For the implementation of the previous bonus feature, the score counting mechanism, an internal counter records how many correct matches there were during GAME IN PROGRESS and divides that number by 10 to get the tens digit and computes the modulo of the number to obtain the ones digit. These two numbers are then passed into separate 7-segment decoders to obtain a 7-segment display version of them. These decoded scores are sent to the mux, which only shifts values when select is switched high—only possible in the GAME OVER state. When GAME OVER is exited, the mux switches back to the normal sum display, as depicted in Figure 2.

For the new bonus features, logout and password reset, 2 new 1-bit control signals (load_button and rng_button) are needed to exit out of any state where the timer is not currently running. These control signals directly come from the push button they are named after and move the current state back to wait while also setting loggedout to 1 or passReset to 1, respectively. Loggedout is an internal trigger for Authentication that simply moves the system back to DIGIT and waits for the correct password. Passreset is a signal that triggers a special state of DIGIT where new password values can be entered to RAM through the password switches. After 4 numbers have been input, a special flag is raised that reads subsequent password values from the RAM instead of the ROM.

Simulation Results

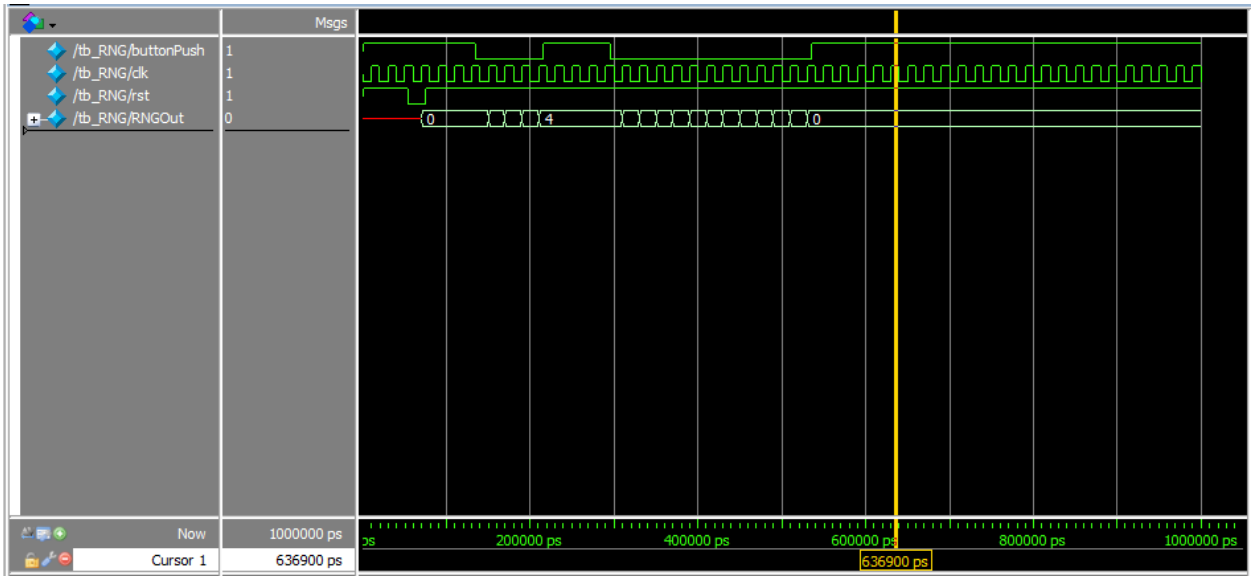


Figure 6: Simulation Results of RNG Module

Pictured above in Figure 6 is the output waveform of the RNG module. When buttonPush is low (normal operation for a pushbutton) for 4 clock cycles, the output is incremented to 4 one at a time. After an arbitrary amount of time, buttonPush is set low again for 12 clock cycles and the output

increases again one at a time until it reaches F, at which point it will overflow and roll back to 0. This behavior is desired, as this will be how we ensure that the output is always a number between 0 and F.

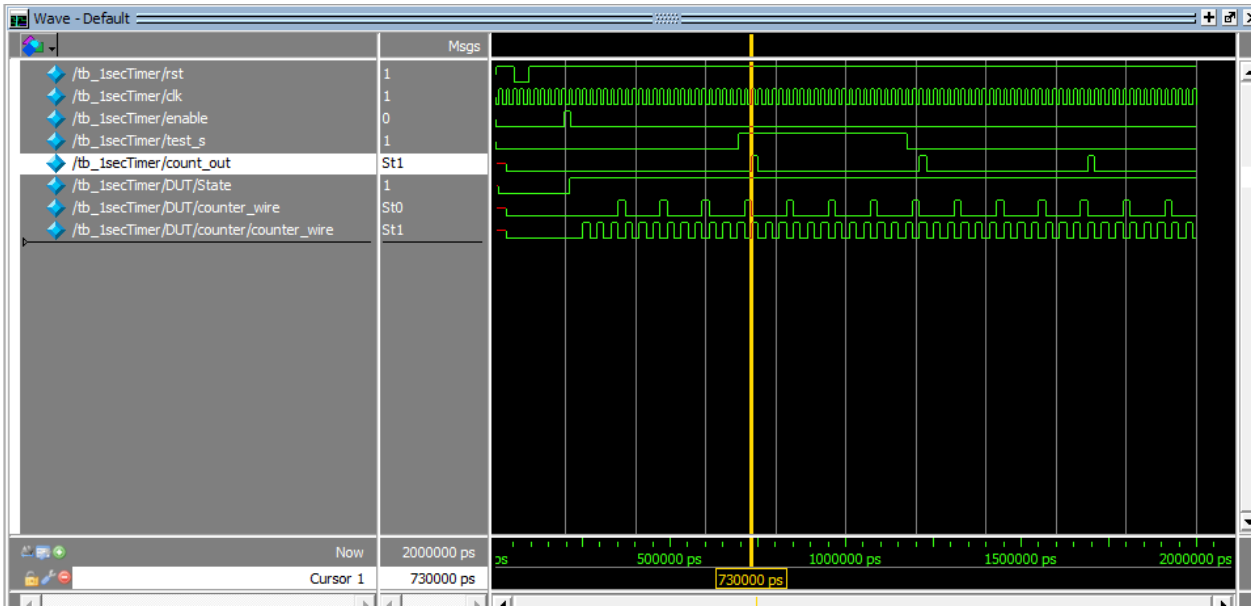


Figure 7: Output Waveform of Modified 1secTimer, First Output

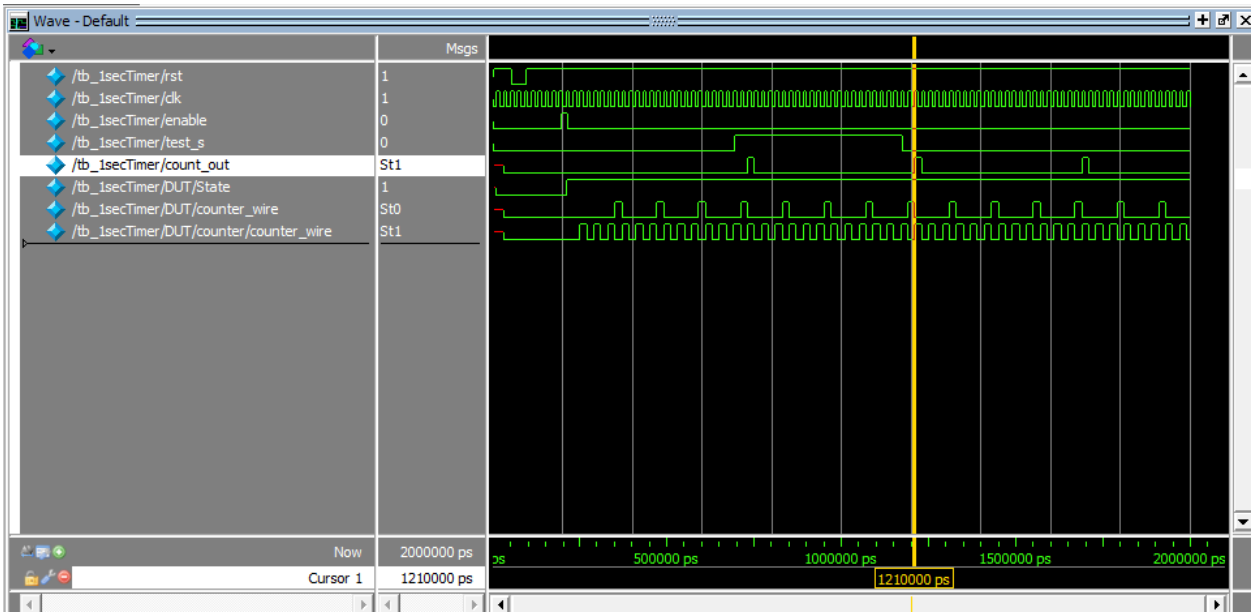


Figure 8: Output Waveform of Modified 1secTimer, Second Output

Depicted above in Figures 7 and 8 are the waveforms of the modified 1secTimer. While the real 1secTimer needs to count to 50,000,000, this would be unfeasible for testing purposes so the final count was modified to be 24 (4x3x2) as a proof of concept. This means that all outputs should be separated by 24 clock cycles. The signal highlighted in the above figures, count_out, is the output of the highest level module, i.e. the one that needs to be separated by 24 clock cycles. Counter_wire and counter/counter_wire represent the nested signals from one module to another.

As can be seen, counter/counter_wire is high every 2 clock cycles, and counter_wire is high for every 3 counter/counter_wire highs. Counter_out is then only high for every 4 counter_wire highs. As a quick proof of function, we can look at the marker in yellow—the first total output occurs at 730 ns and the second one at 1210 ns, a difference of 480 ns. In our testbench (Appendix C) the clock cycle is defined to be 20 ns long, indicating that there are 24 clock cycles between outputs: a perfect match.

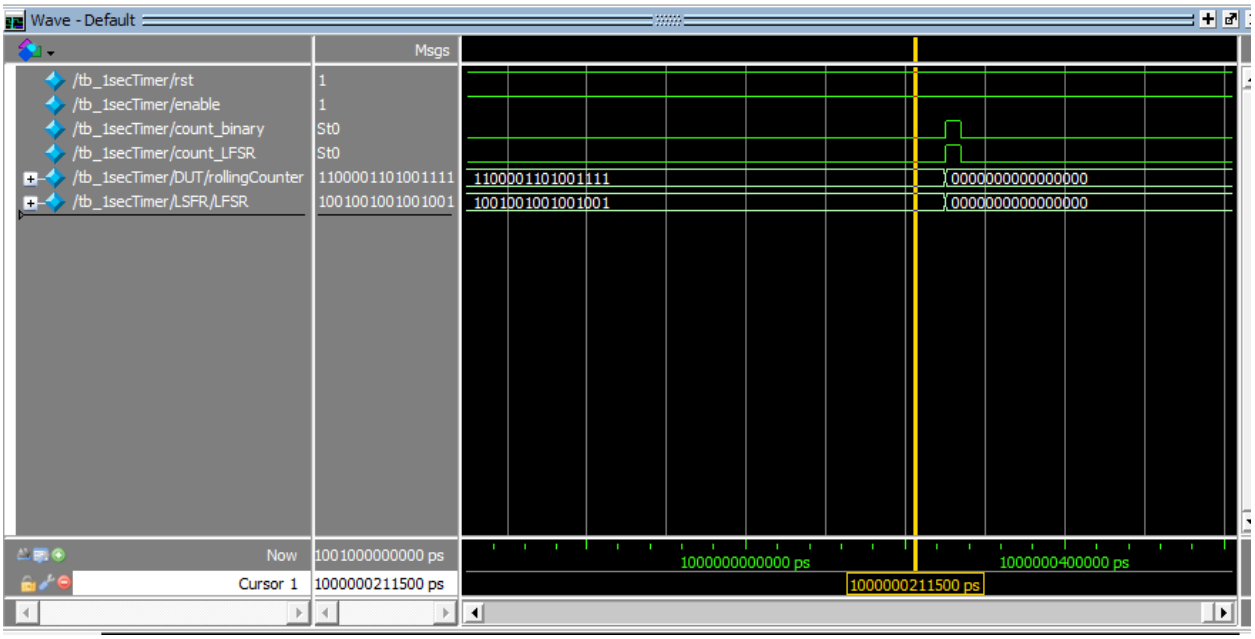


Figure 9: Waveform of OneSecondTimer Done with LFSR

Due to the way that LFSR work, a termination value of 50,000 must still be set in a 16-bit LFSR to get correct looping behavior. However, since LFSR do not count linearly from 0 to 50,000 but input and shift values, one easy way to decide the new termination value is simply to compare the output of the LFSR timer with that of the binary timer. By seeing which number corresponds to 50,000 in the binary timer, the termination value can be found quite easily. Figure 9 shows the behavior of the LFSR timer when this correct termination value is used and as can be seen, it perfectly replicates the old binary timer.

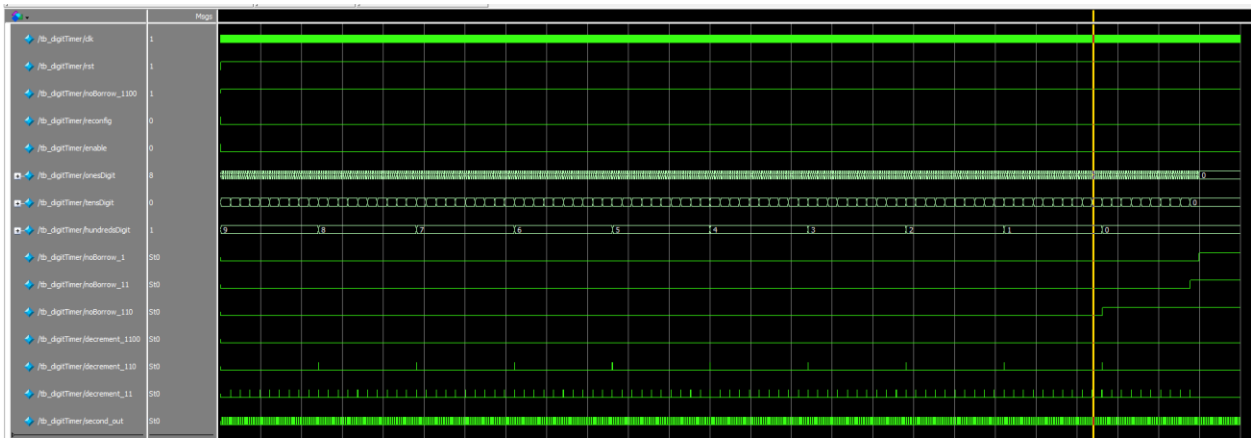


Figure 10: Output Waveform of 3 digitTimer Modules

Depicted above in Figure 10 is the output waveform of 3 instances of digitTimer counting down from 999 to 0. While it is impossible to read what is happening in onesDigit, what is important is to notice the operation of hundredsDigit and tensDigit. From these, the operation of tensDigit to onesDigit can be inferred since these modules are all instances of each other. It takes 10 changes of tensDigit to decrement hundredsDigit by one, indicating that tensDigit is correctly counting down from 9 to 0, and then looping back to 9. At the yellow marker, when the hundredsDigit is just about to turn to 0, we can see a little further down that the instant that hundredsDigit turns to 0 it detects that noBorrow_1100 (the borrow signal from a theoretical thousands digit) is 1, changing its output noBorrow_110 from zero to one indicating to tensDigit that it cannot borrow any more from it. Ten digit changes later in tensDigit it sees that noBorrow_110 is high and changes its own noBorrow_11 from zero to high indicating to onesDigit that it cannot borrow more. onesDigit replicates this behavior and after 10 of its own digit changes, it changes its output noBorrow_1.

FPGA Board Testing Results

Depicted below are pictures of the FPGA in various, key stages of operation. The captions for each picture explain what step is depicted.

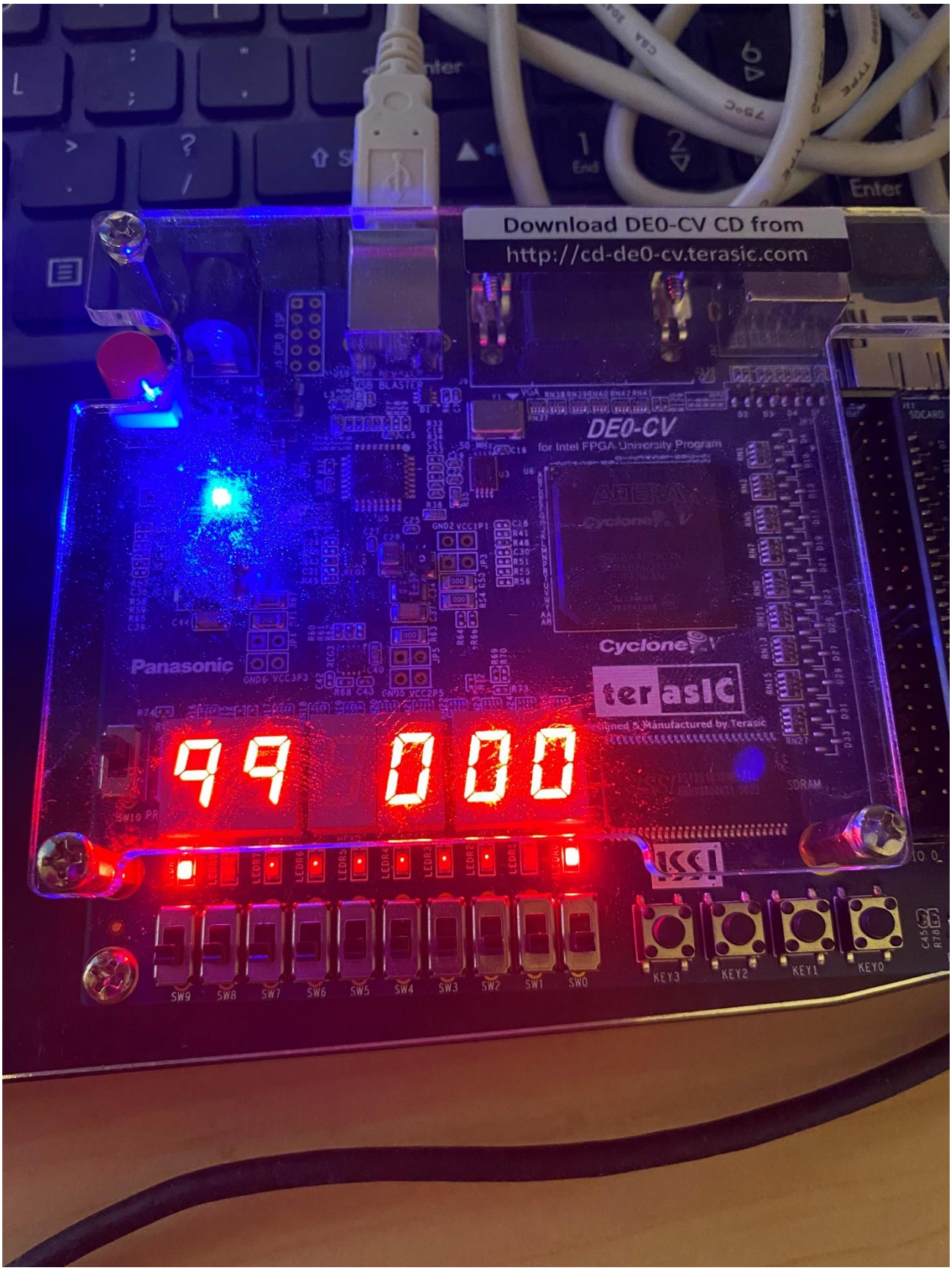


Figure 11: FPGA After Logging In: Pre-Game Start

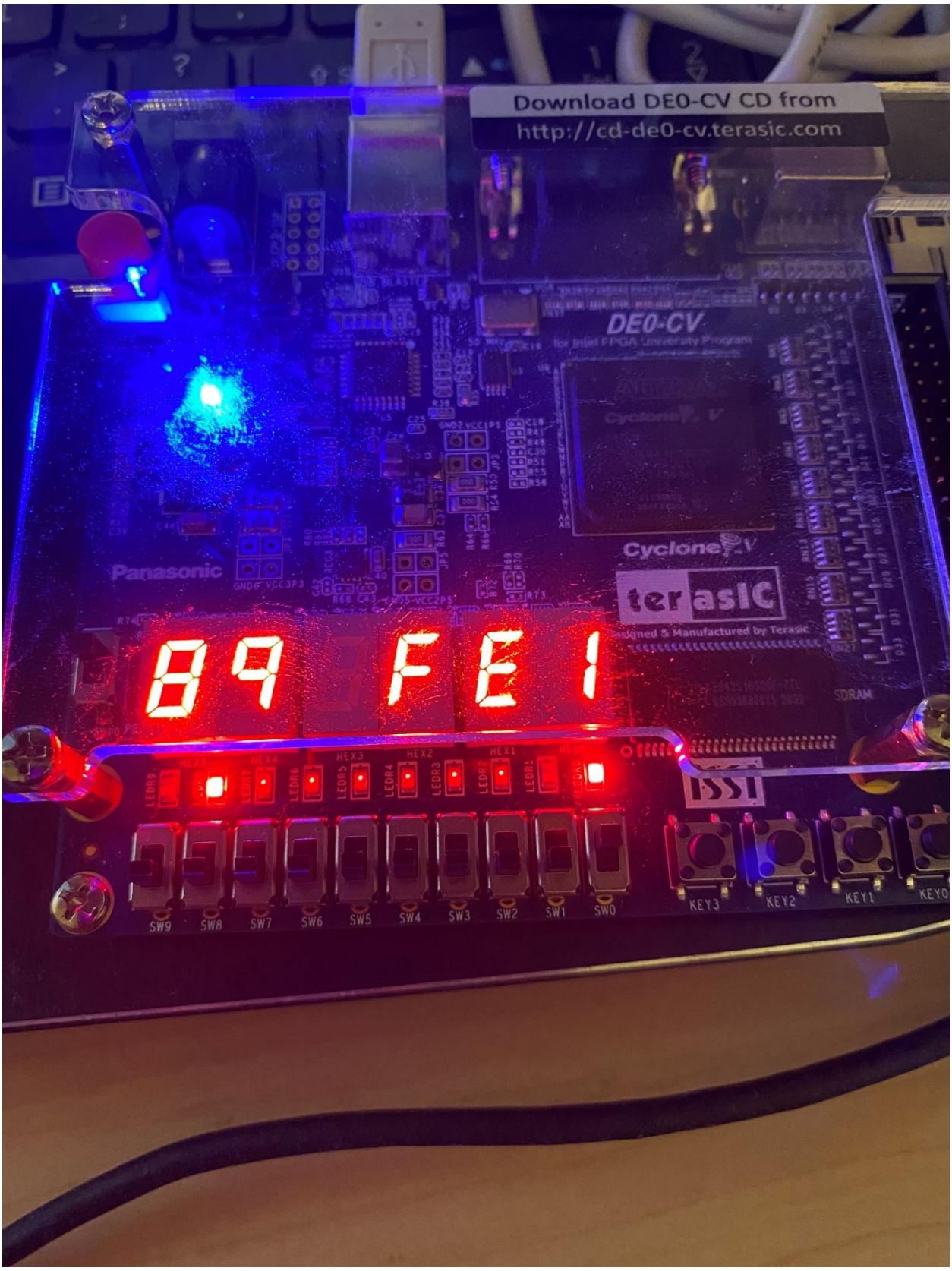


Figure 12: Game in Progress

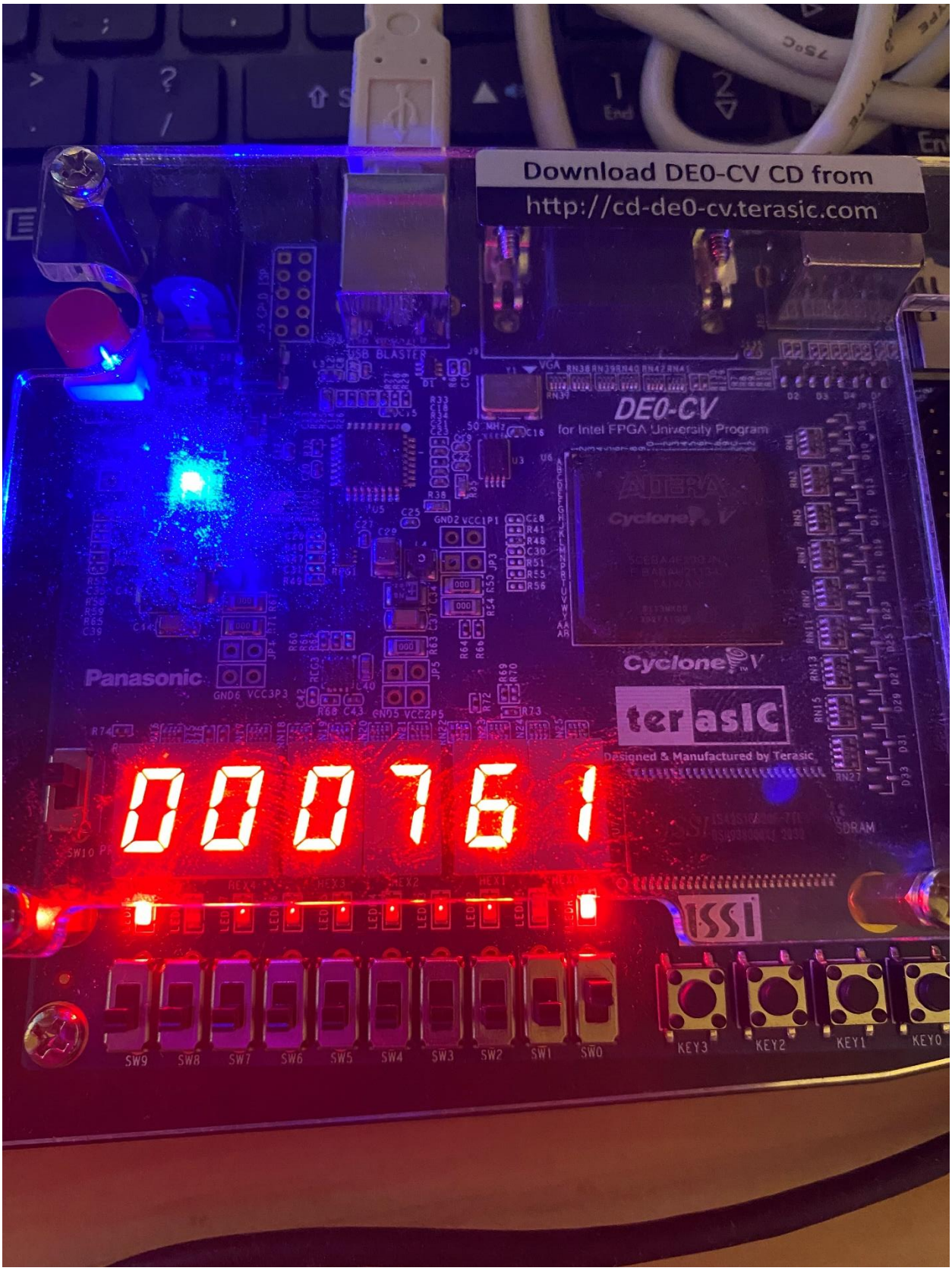


Figure 13: Game Over with Score Depicted

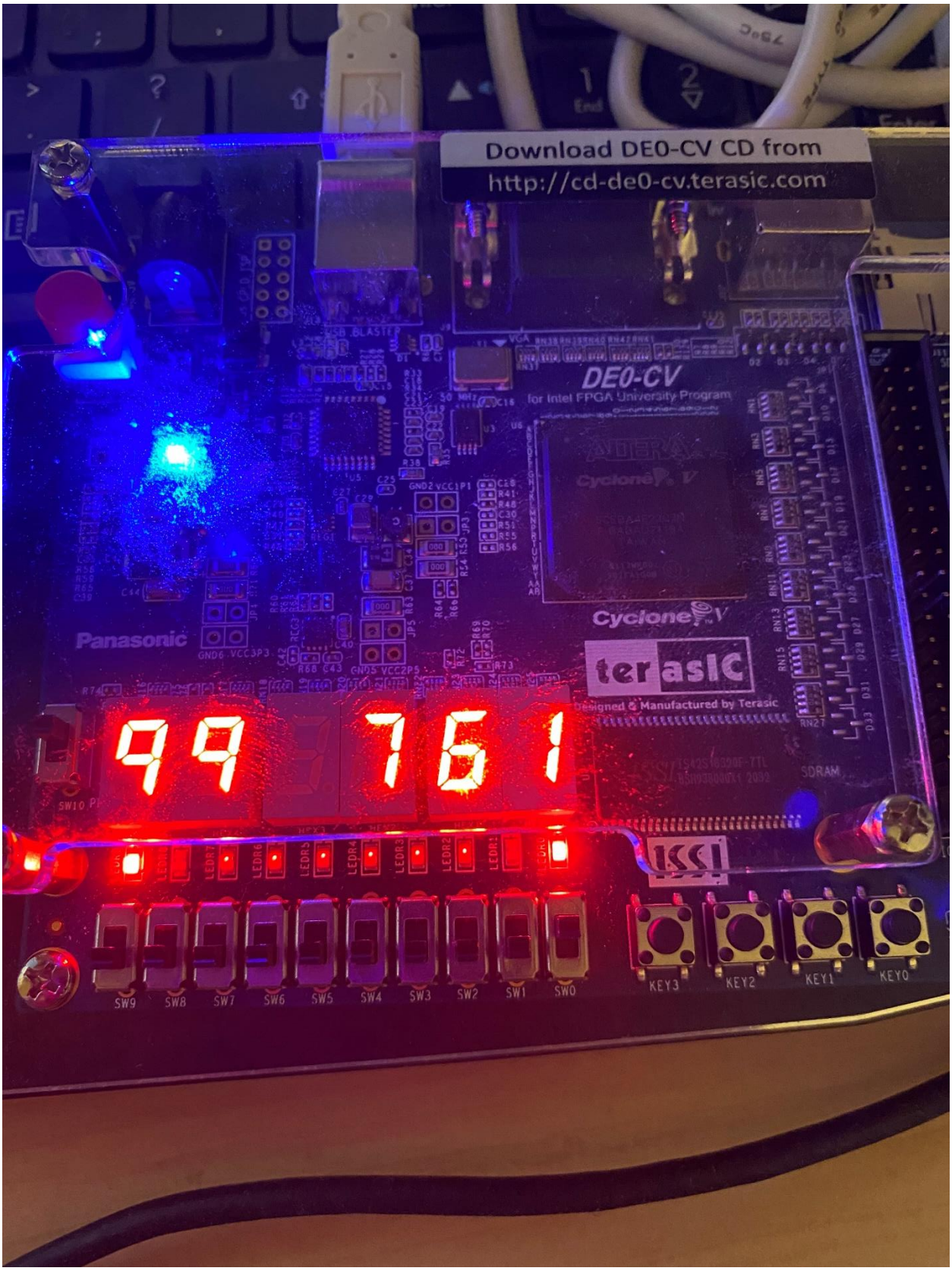


Figure 14: Pre-Game Start of Game 2

It is important to note that in Figure 13 while it may look like the score depicted is actually the sum of RNG and Player 1, it is not. The correct score LED is off, indicating that the game is over as well as the 3rd display showing 0 instead of being off.

Several video demos are shown below illustrating key functions of the new game.

https://drive.google.com/file/d/1EGNSd8g9HJ9wQEqASp-rZ16MK3skOBR-/view?usp=share_link

The video demo above depicts what the board should look like after correctly logging in. A 99 should be displayed on the timer portion and all player inputs should be locked out. When the Game Start button is pressed, the timer begins counting down from 99 and the game has begun.

https://drive.google.com/file/d/10DozI-mu0b0nq8crwDgMqS1due3Z9xwN/view?usp=share_link

The video above demonstrates the functionality of the RNG button and an example round being played. The player will hold the RNG button for as long as they wish to generate a number, and then attempt to match that number to 15 using their slide switches. If the sum is correct, the matching LED will turn on.

https://drive.google.com/file/d/1yrBwHdfnqTjIFGQVVAFpznf_sbw16nY6/view?usp=share_link

The video above shows the functionality of the score counter. After the timer has reached 0 seconds, the middle two displays will change to the score counter display instead of just sum. In the scenario depicted in the demo, 4 correct matches have been done.

https://drive.google.com/file/d/1Mcj7-ZlvStBLn5D8iwJH94jFhoO-n1UE/view?usp=share_link

The video demo above depicts the Game Over state and the beginning of a second iteration of gameplay. In the Game Over state, all the player inputs will be locked out with the only way to progress being pushing the Game Start button. Once this is done, 99 is loaded back into the timer and the middle two displays switch from displaying score to displaying sum again. After this, a second push of Game Start counts the timer down again—signifying a new round of the game.

https://drive.google.com/file/d/1nlhqI6b15eV04f9KkGcBECbihv353-9_/view?usp=share_link

The video demo above shows the new logout feature. After logging in, a player's LOAD button can be pressed to log out. Entering the correct password will subsequently log the player back in.

Conclusion

This new iteration of the game shifts the mental-binary game from a 2-player game to a single player game, increasing replayability and skill level required. The implementation of the bonus feature was actually the hardest feature to create—far more difficult than the timer and digit displays. To do so, a shaped signal from the RNG pushbutton needed to be created for timing reasons.

Detection of a single correct answer is done whenever RNG is pressed, not when player 1's Load button is pressed, as there is no way to check the next output of the Adder module at the initial rising edge. To remedy this, the Adder output is checked at every RNG push. Logically, a player will either keep trying to match an RNG number or move on to a different one. In both cases, at every RNG push the

Adder output will be incorrect and the internal register doesn't increment. A player will only be correct if they match the RNG number and then press the RNG button to move on to a new number. In this situation, at the rising edge of RNG pushbutton, the adder output is checked and found to be correct, and the internal register is incremented.

The unintended side effect of coding the score counter like this is that it relies on players to play as fast as they can, since correct answers are only counted when moving on to a new number. If a player correctly matches a number right as the timer hits 0, unfortunately that isn't counted towards the score total as RNG has not yet been pressed.

The new bonus features logout and an LFSR RNG module have also been implemented. The updated LFSR RNG code can be found in Appendix B. Since password values are no longer hard coded into the FSM of the Access Controller but read off the ROM, the various DIGIT states can all be condensed down into one DIGIT state that iterates a set number of times. This change can be seen in Appendix F. Password reset proved difficult to implement, as in the end there was something wrong with how values are being written into the RAM—the correct password is not the entered values but 0000. The address is being iterated through correctly and it works in simulation, but something about the RAM is messing with the write timing. Since there is no way to see this, this feature was left broken.

Appendix A: Verilog Code for Lab 4

```
//ECE 6370
//David Zhang, 2213233
//Lab 4 Module

//This module defines all the modules and connections that go inside of Lab 4.
//It is functionally identical to Lab 3 except for the instantiations of
//the ROM and RAM that are needed for some of the bonus features.
module Lab4_ZHANG_David(P1INPUT, P2INPUT, P1DISPLAY, P2DISPLAY, TENSDISPLAY, ONESDISPLAY, SUMDISPLAY, LED_C, LED_W, LED_LI, LED_LO,
          P1BUTTON, P2BUTTON, PASSBUTTON, RST, CLK, SCORETENS);

    input [3:0] P1INPUT, P2INPUT;
    output [6:0] P1DISPLAY, P2DISPLAY, TENSDISPLAY, ONESDISPLAY, SUMDISPLAY, SCORETENS;
    output LED_C, LED_W, LED_LI, LED_LO;
    input P1BUTTON, P2BUTTON, PASSBUTTON, RST;
    input CLK;
    reg LED_C, LED_W;
    reg correctAnswer;
    reg hundreds_tensB;
    reg [6:0] dummySignal;
    reg [3:0] scoreCounterTens, scoreCounterOnes;
    reg [6:0] scoreCounterTotal;                        //Minimum size for 99 correct rounds.

    wire p1load_in, passButton_s, p1load_out, p2load_out;
    wire timer_enable, timer_reconfig;                  //New signals from Access Controller to Digits.
    wire ones_Timer, tens_ones, hundreds_tens;          //Signals to connect digit Timers together.
    wire ones_TimerB, tens_onesB;                       //Borrow Signals
    wire score_enable;
    wire correctAnswerShaped;                           //Need a pulsed signal for scoring mechanic.
    wire [6:0] debugging;

    wire [3:0] p1load_N, p2load_N;
    wire [3:0] AdderOutput, tens_display, ones_display;
    wire [6:0] tensScore, onesScore, sumdisplay;

    wire [4:0] addressROM, addressRAM;
    wire [3:0] dataBusROM, dataRAM_out, dataRAM_in;
    wire RAMWR;

    ButtonShaper BS1(P1BUTTON, p1load_in, RST, CLK);
    //ButtonShaper BS2(P2BUTTON, p2load_in, RST, CLK);
    ButtonShaper BSP(PASSBUTTON, passButton_s, RST, CLK);
    ButtonShaper BSC(P2BUTTON, correctAnswerShaped, RST, CLK);  //BS for correct scoring mechanic.

    LoadRegister LR1(P1INPUT, p1load_N, CLK, RST, p1load_out);
    //LoadRegister LR2(P2INPUT, p2load_N, CLK, RST, p2load_out);   //No 2nd Load register needed, things go directly into RNG.
    RNG_LFSR RNGesus(p2load_out, CLK, RST, p2load_N);       //Even though its not technically p2 anymore, to prevent things from breaking
                                                        //we keep the naming convention as it already is.

    Decoder7Seg P1D(p1load_N, P1DISPLAY);
    Decoder7Seg P2D(p2load_N, P2DISPLAY);
    Decoder7Seg PSD(AdderOutput, sumdisplay);
    Decoder7Seg scoreTensDisplay(scoreCounterTens,tensScore);   //Extra 7SD's for keeping score.
    Decoder7Seg scoreOnesDisplay(scoreCounterOnes,onesScore);

    Decoder7Seg Tens(tens_display, TENSDISPLAY);        //New Decoders for the timer digits.
    Decoder7Seg Ones(ones_display, ONESDISPLAY);

    Adder4Bit SUM(p1load_N, p2load_N, AdderOutput);
    AccessControllerSplit MASTER(P2INPUT, passButton_s, LED_LI, LED_LO, p1load_in, P2BUTTON, p1load_out,
             p2load_out, RST, CLK, timer_reconfig, timer_enable, ones_TimerB, score_enable, addressROM,
             dataBusROM, RAMWR, addressRAM, dataRAM_out, dataRAM_in); //Use borrow signal of ones digit as time_out.

    oneSecondLFSR internalTimer(CLK, RST, timer_enable, ones_Timer);  //Instantiation of timing portion
    digitTimer OnesDigit(ones_Timer, tens_ones, tens_onesB, ones_TimerB, timer_reconfig, ones_display, CLK, RST);
    digitTimer TensDigit(tens_ones, hundreds_tens, hundreds_tensB, tens_onesB, timer_reconfig, tens_display, CLK, RST);

    mux_2to1 scoreTens(debugging, tensScore, score_enable, SCORETENS);
    mux_2to1 scoreOnes(sumdisplay, onesScore, score_enable, SUMDISPLAY); //Two muxes needed for scoring feature.

    ROM_PSWD Romulus(addressROM, CLK, dataBusROM);
    Decoder7Seg ARAMSeg(addressRAM, debugging);                     //Debugging]
    RAM_PSWD ARAM(addressRAM, CLK, dataRAM_out, RAMWR, dataRAM_in);

    always @(p1load_N) begin
       if (AdderOutput == 4'b1111)
          begin
             LED_C = 1'b1; LED_W = 1'b0;
          end
       else
          begin
             LED_C = 1'b0; LED_W = 1'b1;
          end
    end
    always @(posedge CLK) begin
       if (RST == 1'b0) begin
          hundreds_tensB <= 1'b1;       //Hundreds-Tens borrow high so tensDigit cannot borrow.
          dummySignal <= 7'b1111111;    //Whatever number to make all segments DARK.
          scoreCounterTotal <= 0;
          scoreCounterTens <= 0;
          scoreCounterOnes <= 0;
       end
       else begin
          scoreCounterTens <= scoreCounterTotal / 4'b1010;
          scoreCounterOnes <= scoreCounterTotal % 4'b1010;   //Math to make sure it counts the 10's and 1's digit.
          if (correctAnswerShaped == 1'b1) begin
             if (AdderOutput == 4'b1111) begin              //Increments on the rng button after a correct answer.
                scoreCounterTotal <= scoreCounterTotal + 1;
             end
          end
          else if (timer_reconfig == 1'b1) begin            //This signal to wipe score as it indicates a new game starting.
             scoreCounterTotal <= 0;
          end
       end
    end
endmodule
```

Figure 15: Module Definition for Lab 4 in Quartus

Appendix B: Verilog Code for LFSR RNG

```
1  //ECE 6370
2  //David Zhang, 2213233
3  //Random Number Generator Module with Linear Feedback Shift Register
4
5  //This module defines the Random Number Generator, LFSR. It takes in a 1-bit active low signal from a push button
6  //and increments an internal register every clock cycle that it detects this active low signal. Increment
7  //in this case refers to how the LFSR works, not by exactly incrementing like a binary counter. This input
8  //is not shaped, and thus how long a player pushes down on the button can be used for RNG. The output is
9  //continuously changing as long as the input is held low, and stops after input is high.
10
11 module RNG_LFSR(rngIn, clk, rst, LFSR);
12         input rngIn, clk, rst;
13         output [3:0] LFSR;
14         reg [3:0] LFSR;
15         wire feedback = LFSR[3] ^ (LFSR[2:0]==3'b111);
16         wire proxy;
17
18
19         assign proxy = ~rngIn;
20
21         always @(posedge clk) begin
22                 if (rst == 1'b0) begin
23                         LFSR <= 0;
24                 end
25                 if (proxy == 1'b1) begin        //Unshaped inverted signal, so active high.
26                         LFSR[0] <= feedback;
27                         LFSR[1] <= LFSR[0] ~^ feedback;
28                         LFSR[2] <= LFSR[1];
29                         LFSR[3] <= LFSR[2];
30                 end
31         end
32 endmodule
33
34
```

Figure 16: Module Definition for RNG

```
//ECE 6370
//David Zhang, 2213233
//Testbench for RNG_LFSR Module

//This module defines the testbench used for testing the RNG_LFSR module. Since control of the input signal is
//done through the access controller, there is no enable for this module. It tests whether it can steadily
//increment through its internal register for every clock cycle that input is held active low.
`timescale 1 ns/100 ps
module tb_RNG();
        reg buttonPush, clk, rst;
        wire [3:0] RNGOut;
        RNG_LFSR DUT(buttonPush, clk, rst, RNGOut);

        always
                begin
                        clk = 1'b0;
                        #10;
                        clk = 1'b1;
                        #10;
                end
        initial
        begin
                rst = 1'b1;
                buttonPush = 1'b1;
                @(posedge clk);
                @(posedge clk);
                @(posedge clk);
                #5 rst = 1'b0;
                @(posedge clk);
                #5 rst = 1'b1;
                @(posedge clk);
                @(posedge clk);
                @(posedge clk);

                #5 buttonPush = 1'b0;
                @(posedge clk);
                @(posedge clk);
                @(posedge clk);
                @(posedge clk);
                @(posedge clk);
                #5 buttonPush = 1'b1;   //4 clk cycles active, should expect 3.

                @(posedge clk);@(posedge clk);@(posedge clk);@(posedge clk);
                #5 buttonPush = 1'b0;   //Should not advance while signal is high.
                @(posedge clk);@(posedge clk);@(posedge clk);@(posedge clk);
                @(posedge clk);@(posedge clk);@(posedge clk);@(posedge clk);
                @(posedge clk);@(posedge clk);@(posedge clk);@(posedge clk);

                @(posedge clk);@(posedge clk);@(posedge clk);@(posedge clk);
                #5 buttonPush = 1'b1;   //Should roll back to 0.
        end

endmodule

```

Figure 17: Testbench Definition for RNG

Appendix C: Verilog Code for 1 second Timer

```
//ECE 6370
//David Zhang, 2213233

//This module counts to 10 based on the clk input. It is part of a nested loop that counts to a total
//of 50,000,000, at which point it will determine that 1 second has passed.

//For simulation purposes, this module will be set to a count of 2. After simulation, the number
//should be set back to 4'b1010, minus 1.
module countTo10(clk, rst, enable, counter_out);
	input clk, enable, rst;
	output counter_out;
	reg counter_out;

	reg [3:0] rollingCounter;
	parameter OFF = 0, ON = 1;
	reg State;

	always @(posedge clk) begin
		if(rst == 1'b0) begin
			rollingCounter <= 4'b0000;
			counter_out <= 1'b0;
			State <= OFF;
		end
		else
			case (State)
				OFF: begin
					rollingCounter <= 0;
					counter_out <= 0;
					if (enable == 1'b1) begin
						State <= ON;
					end
				end

				ON: begin
					if (enable == 1'b0) begin
						if (rollingCounter == 4'b0001) begin    //Set back after simulation
							counter_out <= 1'b1;
							rollingCounter <= 4'b0000;
						end
						else begin
							counter_out <= 1'b0;
							rollingCounter <= rollingCounter + 1'b1;
						end
					end
					else begin
						State <= OFF;
					end
				end

				default: begin
					State <= OFF;
				end
			endcase
	end
endmodule
```

Figure 18: Module Definition for countTo10, the 3rd nested module.

```
//This module counts to 100 based on the clk input. It is part of a nested loop that counts to a total
//of 50,000,000, at which point it will determine that 1 second has passed.

//For simulation purposes, this module will be set to a count of 3. The real number should be set
//back after simulation to be 7'b1100100, minus 1.
module countTo100(clk, rst, enable, counter_out);
	input clk, enable, rst;
	output counter_out;
	reg counter_out;

	reg [6:0] rollingCounter;
	wire counter_wire;
	parameter OFF = 0, ON = 1;
	reg State;
	countTo10 counter(clk, rst, enable, counter_wire);

	always @(posedge clk) begin
		if(rst == 1'b0) begin
			rollingCounter <= 7'b0000000;
			counter_out <= 1'b0;
			State <= OFF;
		end
		else
			case (State)
				OFF: begin
					rollingCounter <= 0;
					counter_out <= 0;
					if (enable == 1'b1) begin
						State <= ON;
					end
				end

				ON: begin
					if (enable == 1'b0) begin
						if (counter_wire == 1'b1) begin
							if (rollingCounter == 7'b0000010) begin //Set back after simulation
								counter_out <= 1'b1;
								rollingCounter <= 0;
							end
							else begin
								counter_out <= 1'b0;
								rollingCounter <= rollingCounter + 1'b1;
							end
						end
						else begin
							counter_out <= 1'b0;
						end
					end
					else begin
						State <= OFF;
					end
				end

				default: begin
					State <= OFF;
				end
			endcase
	end
endmodule
```

Figure 19: Module Definition for countTo100, the 2nd nested module

```
//This module counts to 50000 based on the clk input. It is part of a nested loop that counts to a total
//of 50,000,000, at which point it will determine that 1 second has passed.

//For simulation purposes, this module will be set to a count of 4. The real number should be set
//back after simulation to be 16'b1100001101010000, minus 1.
module oneSecondTimer(clk, rst, enable, counter_out);
	input clk, enable, rst;
	output counter_out;
	reg counter_out;

	reg [15:0] rollingCounter;
	wire counter_wire;
	parameter OFF = 0, ON = 1;
	reg State;
	countTo100 counter(clk, rst, enable, counter_wire);

	always @(posedge clk) begin
		if(rst == 1'b0) begin
			rollingCounter <= 0;
			counter_out <= 1'b0;
			State <= OFF;
		end
		else
			case (State)
				OFF: begin
					rollingCounter <= 0;
					counter_out <= 0;
					if (enable == 1'b1) begin
						State <= ON;
					end
				end

				ON: begin
					if (enable == 1'b0) begin
						if (counter_wire == 1'b1) begin
							if (rollingCounter == 16'b0000000000000011) begin	//Set back after simulation
								counter_out <= 1'b1;
								rollingCounter <= 0;
							end
							else begin
								counter_out <= 1'b0;
								rollingCounter <= rollingCounter + 1'b1;
							end
						end
						else begin
							counter_out <= 1'b0;
						end
					end
					else begin
						State <= OFF;
					end
				end

				default: begin
					State <= OFF;
				end
			endcase
	end
endmodule
```

Figure 20: Module Definition for 1 second Timer, the highest module

```
Ln#
 2  //David Zhang, 2213233
 3  //Testbench for 1-second Timer
 4
 5  //This module defines the parameters for testing the 1-second timer. Since realistically is needs
 6  //to count to 50,000,000 on a 50 MHz clock, for a proof of concept we simply test to see if it can
 7  //count to 24 first, based on the nesting design of the 1 second timer.
 8  `timescale 1 ns/100 ps
 9  module tb_1secTimer();
10          reg rst, clk, enable;
11          reg test_s;
12          wire count_out;
13          oneSecondTimer DUT(clk, rst, enable, count_out);
14
15          always
16                  begin
17                          clk = 1'b0;
18                          #10;
19                          clk = 1'b1;
20                          #10;
21                  end
22
23          initial
24          begin
25                  rst = 1'b1;
26                  test_s = 1'b0;
27                  enable = 1'b0;
28                  @(posedge clk);
29                  @(posedge clk);
30                  @(posedge clk);
31                  #5 rst = 1'b0;
32                  @(posedge clk);
33                  @(posedge clk);
34                  #5 rst = 1'b1;
35                  @(posedge clk);
36                  @(posedge clk);
37                  @(posedge clk); //Multiple clock cycles later to make sure it doesn't turn on
38                  @(posedge clk); //without enable signal.
39                  @(posedge clk);
40                  #5 enable = 1'b1;
41                  @(posedge clk);
42                  #5 enable = 1'b0; //Need at least 24 cycles to test if "1-second" has passed.
43
44                  @(posedge clk);@(posedge clk);@(posedge clk);@(posedge clk);@(posedge clk);@(posedge clk);
45                  @(posedge clk);@(posedge clk);@(posedge clk);@(posedge clk);@(posedge clk);@(posedge clk);
46                  @(posedge clk);@(posedge clk);@(posedge clk);@(posedge clk);@(posedge clk);@(posedge clk);
47                  @(posedge clk);@(posedge clk);@(posedge clk);@(posedge clk);@(posedge clk);@(posedge clk);
48
49                  #5 test_s = 1'b1; //Should have observed a 1 by now.
50
51                  @(posedge clk);@(posedge clk);@(posedge clk);@(posedge clk);@(posedge clk);@(posedge clk);
52                  @(posedge clk);@(posedge clk);@(posedge clk);@(posedge clk);@(posedge clk);@(posedge clk);
53                  @(posedge clk);@(posedge clk);@(posedge clk);@(posedge clk);@(posedge clk);@(posedge clk);
54                  @(posedge clk);@(posedge clk);@(posedge clk);@(posedge clk);@(posedge clk);@(posedge clk);
55
56                  #5 test_s = 1'b0; //Another 1.
57          end
58
59  endmodule
60
```

Figure 21: Testbench Definition for the Modified 1 second Timer

Appendix D: Verilog Code for Digit Timer

```
Ln#
 1  //ECE 6370
 2  //David Zhang, 2213233
 3  //Digit Timer Module
 4
 5  //This module defines the operation of the Digit Timers, which are responsible for displaying the amount of time
 6  //left as a timer. These modules are non-digit specific and thus can be instantiated in any place and linked
 7  //together to provide correct operation. They take in an active high signal to decrement their internal counter
 8  //until it reaches 0, at which point it will borrow from its left neighbor until it is no longer possible.
 9
10  module digitTimer (DecrementD, DecrementU, noBorrowU, noBorrowD, reconfig, numOut, clk, rst);
11          input DecrementD, noBorrowU, reconfig, clk, rst;
12          output DecrementU, noBorrowD;
13          output [3:0] numOut;
14
15          reg DecrementU, noBorrowD;              //Outgoing signals to decrement Upstream, or tell right neighbor
16          reg [3:0] numOut;                       //that noBorrow possible.
17
18          always @(posedge clk) begin
19                  if (rst == 1'b0) begin          //Standard reset block.
20                          noBorrowD <= 0;
21                          numOut <= 0;
22                          DecrementU <= 0;
23                  end
24                  if (reconfig == 1'b1) begin
25                          numOut <= 4'b1001;      //Load 9 in when reconfig.
26                          noBorrowD <= 0;         //"I can now lend numbers to my rightmost neighbor"
27                          DecrementU <= 0;
28                  end
29                  else if (DecrementD == 1'b1) begin      //If downstream requests decrement...
30                          if (numOut == 1'b0) begin       //Check if internal number is 0.
31                                  if (noBorrowU == 1'b0) begin    //If it is 0, see if I can borrow from upstream
32                                          numOut <= 4'b1001;
33                                          DecrementU <= 1'b1;     //If you can borrow, tell upstream and reset self back to 9.
34                                  end
35                                  else begin                      //Cannot borrow from upstream, and I am at 0. Tell my right neighbor that I cannot lend.
36                                          noBorrowD <= 1'b1;
37                                  end
38
39                          end
40                          else begin
41                                  numOut <= numOut - 1'b1;//If internal is not 0, decrement by 1.
42                                  DecrementU <= 1'b0;
43                          end
44                  end
45                  else if (DecrementD == 1'b0) begin
46                          DecrementU <= 1'b0;             //While just chilling, make sure you are not requesting from upstream neighbor.
47                          if (noBorrowU == 1'b1) begin
48                                  if (numOut == 1'b0) begin
49                                          noBorrowD <= 1'b1;      //Same condition as above (cannot borrow and at zero) just done during downtime.
50                                  end
51                          end
52                  end
53          end
54  endmodule
55
```

Figure 22: Module Definition for Digit Timer

```
Ln#
 1  //ECE 6370
 2  //David Zhang, 2213233
 3  //Digit Timer Testbench.
 4
 5  //This module defines the testbench for the Digit Timer modules. It instantiates an arbitrary, greater than 1
 6  //number of Digit Timers and links them together. It then feeds the a single pulse signal created from the modified 1secTimer into
 7  //the least significant digit after loading all timers with 9.
 8   `timescale 1 ns/100 ps
 9  module tb_digitTimer();
10          reg clk, rst, noBorrow_1100, reconfig, enable;
11          wire [3:0] onesDigit, tensDigit, hundredsDigit;
12          wire decrement_11, decrement_110, decrement_1100, noBorrow_11, noBorrow_110, noBorrow_1, second_out;
13
14          digitTimer ones(second_out, decrement_11, noBorrow_11, noBorrow_1, reconfig, onesDigit, clk, rst);
15          digitTimer tens(decrement_11, decrement_110, noBorrow_110, noBorrow_11, reconfig, tensDigit, clk, rst);
16          digitTimer hundreds(decrement_110, decrement_1100, noBorrow_1100, noBorrow_110, reconfig, hundredsDigit, clk, rst);
17          oneSecondTimer oneSec(clk, rst, enable, second_out);
18
19          //Instantiation of 3 timer modules for each base 10 digit. Each is linked to the other through wires. The naming convention of the
20          //wires shows which digits are linked by 1's, i.e. 110 is the wire connecting hundreds to tens.
21
22          //oneSecondTimer here is the modified version that counts 24 clock cycles per pulsed output.
23
24          always
25                  begin
26                          clk = 1'b0;
27                          #10;
28                          clk = 1'b1;
29                          #10;
30                  end
31          initial
32          begin
33                  enable = 1'b0;
34                  rst = 1'b1;
35                  noBorrow_1100 = 1'b1;           //Signals all start at 0, noBorrow always stays at 1 to ensure that hundreds does not borrow.
36                  reconfig = 1'b0;
37                  @(posedge clk); @(posedge clk); @(posedge clk);
38                  #5 rst = 1'b0;
39                  @(posedge clk);
40                  #5 rst = 1'b1;
41                  reconfig = 1'b1;                //Reset signal and Reconfig to load Timers.
42                  @(posedge clk);
43                  reconfig = 1'b0;
44                  @(posedge clk);
45                  @(posedge clk);
46                  enable = 1'b1;
47                  @(posedge clk);
48                  enable = 1'b0;
49                  //Arbitrary amount of clock cycles, should decrement correctly from 999 one at a time per pulsed input.
50          end
51  endmodule
52
53
```

Figure 23: Testbench Definition for 3-instance Digit Timer

Appendix E: Verilog Code for 2-to-1 Multiplexer

```
Ln#
 1  //ECE6370
 2  //David Zhang, 2213233
 3  //Multiplexer 2 to 1
 4
 5  //This module defines the mux needed for implementing the bonus feature of lab 3. It takes in two 7-bit
 6  //inputs and a 1-bit select to output a 7-bit signal. The inputs and outputs are designed to be matching
 7  //signals of the 7-segment decoders, and thus do not represent their literal binary value.
 8
 9  module mux_2to1( input [6:0] a,                 // 7-bit input called a
10                   input [6:0] b,                     // 7-bit input called b
11                   input sel,                 // input sel used to select between a,b
12                   output reg [6:0] out);          // 7-bit output based on input sel
13
14     // This always block gets executed whenever a/b/sel changes value
15     // When it happens, output is assigned to either a/b
16     always @ (a,b,sel) begin
17        case (sel)
18           1'b0 : out <= a;  //sel 0 choose A
19           1'b1 : out <= b;  //sel 1 choose B
20
21        endcase
22     end
23  endmodule
24
```

Figure 24: Module Definition for the 2-to-1 7-bit Multiplexer

Appendix F: Verilog Code for Modified Access Controller

```
1  //ECE 6370
2  //Author: David Zhang, 3233
3  //Access Controller Module, Lab 4
4
5  //This module defines the modified access controller for the mental binary game, which replaces the logout feature and button
6  //with a button to load 99 seconds into the timer and start the subsequent game. After the timer reaches 0, all features are
7  //locked out until the button is pushed again to add 99 seconds, then pushed again to start a new round. In addition, the timer
8  //and authentication functions have been split into two separate modules named GameController and Authentication.
9
10 module AccessControllerSplit(passIn, passButton, loggedIn, loggedOut, p1loadIn, p2loadIn, p1loadOut, p2loadOut,
11                rst, clk, timer_reconfig, timer_enable, time_out, score_enable, addressROM, dataBusROM,
12                RAMWR, addressRAM, dataRAM_out, dataRAM_in);
13    input [3:0] passIn, dataBusROM, dataRAM_out,  dataRAM_in;
14    input passButton, p1loadIn, p2loadIn, rst, clk;
15    input time_out;
16    output timer_reconfig, timer_enable, score_enable;
17    output loggedIn, loggedOut, p1loadOut, p2loadOut;
18    output RAMWR;
19    output [4:0] addressROM, addressRAM;
20    wire loggedIn, loggedOut, p1loadOut, p2loadOut, timer_reconfig, timer_enable, score_enable;
21
22    wire enable, logout_s, passReset;
23    wire RAMWR; //To be sized and changed to outputs later
24
25    GameController MASTER(enable, passButton, p1loadIn, p2loadIn, p1loadOut, p2loadOut, rst, clk, timer_reconfig,
26                timer_enable, time_out, score_enable, logout_s, passReset);
27    Authentication MASUTAH(passIn, passButton, loggedIn, loggedOut, addressROM, dataBusROM, enable, passReset,
28                rst, clk, logout_s, RAMWR, addressRAM, dataRAM_out, dataRAM_in);
29
30 endmodule
31
32
```

Figure 25: Module Definition for Modified Access Controller

```
//ECE 6370
//David Zhang, 2213233
//Authentication Module

//This module defines the Authentication half of the Access Controller. It takes in a shaped signal from the Password button
//along with a 4-bit binary input from the slide switches. It reads these inputs and determines whether the correct password
//defined in the ROM/RAM has been input, after which case it sets enable high, signalling to the GameController half to start
//normal operation of the Game.

module Authentication(passIn, passButton, loggedIn, loggedOut, addressROM, dataBusROM, enable, passChange, rst, clk,
            logout_s, RAMWR, addressRAM, dataRAM_out, dataRAM_in);
    input passButton, rst, clk, passChange, logout_s;        //New signals from GameController
    input [3:0] passIn, dataBusROM, dataRAM_in;           //New input from the ROM,bus signal.
    output loggedIn, loggedOut, enable, RAMWR;
    output [4:0] addressROM, addressRAM;              //New 2-bit output to ROM
    output [3:0] dataRAM_out;               //4-bit data out to RAM

    parameter DIGIT = 0, VERIFY = 4, OPERATIONAL = 5;  //Importantly these states are needed for new operation.
    reg passwordRight, loggedIn, loggedOut, enable;
    reg[3:0] State;
    reg[2:0] digits_place;              //Which digit am I on?
    reg[1:0] WaitingForROM;                 //Timer to stay in states until ROM/RAM done reading.
    reg switchToRAM;                    //New flag for bonus feature
    reg[4:0] addressROM, addressRAM;    //Address bits for ROM/RAM, minimum 5 bits
    reg[3:0] dataROM;                      //Internal storage for whatever comes from dataBusROM
    reg RAMWR;                        //Write enable signal for RAM
    reg[3:0] dataRAM_out;                   //Outgoing data bits to RAM
    reg[3:0] dataRAM_storage;               //Storage for ingoing data bits of RAM
    reg[1:0] RAMWait1, RAMWait2;

    always@(posedge clk) begin
        if (rst == 1'b0) begin                  //Reset of all signals.
            State <= DIGIT;
            passwordRight <= 1'b1;
            loggedIn <= 1'b0;
            loggedOut <=1'b1;
            WaitingForROM <= 1'b0;
            enable <= 1'b0;
            switchToRAM <= 1'b0;              //Only gets wiped at reset, not logout.
            addressROM <= 0;
            dataRAM_out <= 0;
            addressRAM <= 0;
            digits_place <= 0;
            RAMWR <= 0;                     //Set to always be reading from RAM
            dataRAM_storage <= 0;
            RAMWait1 <= 0;
            RAMWait2 <= 0;
        end
        else
            case(State)
                DIGIT: begin                //Setting all signals to correct baseline
                    loggedIn <= 1'b0;
                    loggedOut <=1'b1;
                    enable <= 1'b0;
                    if (digits_place != 4) begin     //If we are on the "5th" digit, dont even bother just go to end.
                        if (WaitingForROM != 2'b11) begin   //Ensures that we don't have looping behavior
                            WaitingForROM <= WaitingForROM + 1'b1;
                        end
                        if (WaitingForROM == 2'b11) begin   //Wait at least 3 clock cycles for number from ROM/RAM to come back
                            dataROM <= dataBusROM;      //Store whatever was retrieved onto internal store
                            if (switchToRAM == 1'b0) begin   //Flag has not been triggered, assume ROM
                                if (passButton == 1'b1)begin //Operation should ONLY proceed whenever button depressed.
                                    if (passIn == dataROM)begin   //Checks player switches vs. whatever came from ROM
                                        State <= DIGIT;
                                        WaitingForROM <= 1'b0;  //Reset the wait time each transition.
                                        addressROM <= addressROM + 1'b1; //Move to next ROM address.
                                        digits_place <= digits_place + 1;
                                    end
                                    else begin
                                        passwordRight <= 1'b0;  //Mark as wrong if it was wrong (duh)
                                        State <= DIGIT;
                                        WaitingForROM <= 1'b0;
                                        addressROM <= addressROM + 1'b1;
                                        digits_place <= digits_place + 1;
                                    end
                                end
                            end
                            else begin        //FLag has been triggered, do all subsequent stuff in RAM
                                  if (RAMWR == 1'b1) begin      //Beginning of password overwrite
                                    //dataRAM_storage <= passIn;
                                    if (passButton == 1'b1) begin
                                        dataRAM_out <= passIn;  //Pass the inputs to the RAM
                                        addressRAM <= addressRAM + 1;    //Advance to next address
                                        digits_place <= digits_place + 1; //Advance to next digit
                                        WaitingForROM <= 1'b0;     //Reset wait timer
                                        State <= DIGIT;
                                    end
                                end
                                else begin                    //Subsequent READ cycles, identical to ROM
                                    dataRAM_storage <= dataRAM_in;
                                    //if (RAMWait1 != 2'b11) begin
                                        //RAMWait1 <= RAMWait1 + 1;
                                        if (RAMWait2 != 1'b1) begin
                                            addressRAM <= addressRAM +1;
                                            RAMWait2 <= RAMWait2 + 1;
                                            WaitingForROM <= 0;
                                        end
                                    //end
                                    else begin
                                        if (passButton == 1'b1)begin
                                            if (passIn == dataRAM_storage) begin   //Checks player switches vs. whatever came from RAM
```

```
                                    if (passIn == dataRAM_storage) begin   //Checks player switches vs. whatever came from RAM
                                        State <= DIGIT;
                                        WaitingForROM <= 1'b0;
                                        addressRAM <= addressRAM + 1'b1;
                                        digits_place <= digits_place + 1;
                                        //WaitingForRAM <= 0;
                                    end
                                    else begin
                                        passwordRight <= 1'b0;
                                        State <= DIGIT;
                                        WaitingForROM <= 1'b0;
                                        addressRAM <= addressRAM + 1'b1;
                                        digits_place <= digits_place + 1;
                                        //WaitingForRAM <= 0;
                                    end
                                end
                            end
                        end
                    end
                end
                else begin
                    State <= VERIFY;  //Move to verify on the last digit.
                end
            end

            VERIFY: begin
                digits_place <= 0;
                RAMWR <= 0;
                if (passwordRight == 1'b1)begin  //Verification check. All numbers should have
                    State <= OPERATIONAL;   //been entered correctly up to this stage.
                    addressROM <= 0;
                    addressRAM <= 0;
                    RAMWait1 <= 0;
                    RAMWait2 <= 0;
                end
                else begin
                    State <= DIGIT;
                    addressROM <= 0;  //Arbitrary where to put, but reset address here
                    addressRAM <= 0;
                    passwordRight <= 1'b1;  //Set pass to right and return to start.
                    RAMWait1 <= 0;
                    RAMWait2 <= 0;
                end
            end
            OPERATIONAL: begin
                loggedIn <= 1'b1;
                loggedOut <= 1'b0;      //Arbitrary where to put, but at this point should be logged in
                enable <= 1'b1;         //Keep enable high for correct operation
                if (passChange == 1'b1) begin
                    State <= DIGIT;
                    switchToRAM <= 1'b1;    //Switch to RAM since player wanted to password change.
                    enable <= 1'b0;   //In either of these cases, should lockout player control
                    RAMWR <= 1'b1;    //write enable, so 1
                end
                if (logout_s == 1'b1) begin
                    State <= DIGIT;
                    enable <= 1'b0;
                end
            end

            default: begin
                State <= DIGIT;
                passwordRight <= 1'b1;
                loggedIn <= 1'b0;
                loggedOut <=1'b1;
                WaitingForROM <= 1'b0;
                enable <= 1'b0;
                addressROM <= 0;
                dataRAM_out <= 0;
                addressRAM <= 0;
                digits_place <= 0;
                RAMWR <= 0;
            end
        endcase
    end
endmodule
```

Figure 26: Module Definition for Authentication Half of Access Controller

```
//ECE 6370
//David Zhang, 2213233
//Game Controller Module

//This module defines the GameController half of the Access Controller. It takes in a long active high enable signal from the
//Authentication half to start operation. It loads 99 seconds into the timer displays and blocks all player inputs until the
//Game Start button has been pushed. Insert stuff about logout here, when done. After 99 seconds have run out, the system goes
//to the Game Over state where it turns on the score display, showing the amount of correct matches that the player did.

module GameController(enable, passButton, p1loadIn, p2loadIn, p1loadOut,
            p2loadOut, rst, clk, timer_reconfig, timer_enable, time_out, score_enable, logout_s, passReset);
    input enable;
    input passButton, p1loadIn, p2loadIn, rst, clk;
    input time_out;
    output timer_reconfig, timer_enable, score_enable;
    output p1loadOut, p2loadOut;
    output logout_s, passReset;                     //New output signals for logging out and resetting password.
    reg p1loadOut, p2loadOut, timer_reconfig, timer_enable, score_enable;

    parameter DIGIT0 = 0, DIGIT1 = 1, DIGIT2 = 2, DIGIT3 = 3, VERIFY = 4, RECONFIG_TIMER = 5; //Rename of some states to match up with FSM
    parameter PREGAMESTART = 6, GAMEINPROGRESS = 7, GAMEOVER = 8;              //Importantly these states are needed for new operation.
    parameter WAIT = 9;                         //New starting state since AccessController split
    reg[3:0] State;
    reg logout_s, passReset;
    reg[2:0] wait_timer;                        //Arbitrarily long wait timer for initial state

    always @(posedge clk) begin
       if (rst == 1'b0) begin                   //Reset of all signals.
          State <= WAIT;
          p1loadOut <= 1'b0;
          p2loadOut <= 1'b1;                    //Default is high for a pushbutton.
          timer_enable <= 1'b0;
          timer_reconfig <= 1'b0;
          score_enable <= 1'b0;
          logout_s <= 1'b0;
          passReset <= 1'b0;
          wait_timer <= 0;
       end
       else begin
          case(State)
             WAIT: begin       //Needs to wait some clock cycles for enable to update--necessary for logout and passreset function.
                if (wait_timer != 2'b11) begin
                   wait_timer <= wait_timer + 1'b1; //This happens in <1 us regardless, so arbitrary for human perception
                end
                else begin
                   logout_s <= 1'b0;
                   passReset <= 1'b0;
                   if (enable == 1'b1) begin
                      State <= RECONFIG_TIMER;
                   end
                end
             end
             RECONFIG_TIMER: begin
                wait_timer <= 0;
                //p1loadOut <= p1loadIn;   //Player inputs must still be blocked at this stage.
                //p2loadOut <= p2loadIn;
                timer_reconfig <= 1'b1;    //Add to timers and immediately move to next state
                score_enable <= 1'b0;      //Turn off score when starting new round.
                State <= PREGAMESTART;
                //if (passButton == 1'b1)begin   //Implementation of logout function, no longer needed.
                // State <= DIGIT0;
                //end

             end

             PREGAMESTART: begin
                timer_reconfig <= 1'b0;
                if (passButton == 1'b1) begin //Ensures reconfig is high for only 1 cycle and waits till button is pressed.
                   timer_enable <= 1'b1;
                   State <= GAMEINPROGRESS;//If button pressed, timer starts going down and move into gameplay phase.
                end
                if (p2loadIn == 1'b0) begin   //p2 is analog to RNG, so its active LOW
                   passReset <= 1'b1;
                   State <= WAIT;
                end
                if (p1loadIn == 1'b1) begin   //Player Load button is logout
                   logout_s <= 1'b1;
                   State <= WAIT;
                end
             end

             GAMEINPROGRESS: begin
                p1loadOut <= p1loadIn;
                p2loadOut <= p2loadIn;  //Finally player inputs are unblocked, game works as usual.
                if (time_out == 1'b1) begin
                   State <= GAMEOVER;   //Moves to Game Over once it detects active high time out signal (noBorrow)
                end
             end

             GAMEOVER: begin
                p1loadOut <= 1'b0;
                p2loadOut <= 1'b1;      //Blocks all player inputs again.

                timer_enable <= 1'b0;      //Turn off countdown.
                score_enable <= 1'b1;      //Turn on score to switch MUX output (bonus Feature)
                if (passButton == 1'b1) begin
                   State <= WAIT;    //Move to start whenever game button is pressed.
                end
                if (p2loadIn == 1'b0) begin
                   passReset <= 1'b1;
                   State <= WAIT;
```

```
101                 State <= WAIT;
102             end
103             if (p1loadIn == 1'b1) begin
104                 logout_s <= 1'b1;
105                 State <= WAIT;
106             end
107         end
108 
109         default: begin
110             State <= WAIT;
111             p1loadOut <= 1'b0;
112             p2loadOut <= 1'b1;          //Default is high for a pushbutton.
113             timer_enable <= 1'b0;
114             timer_reconfig <= 1'b0;
115             score_enable <= 1'b0;
116             logout_s <= 1'b0;
117             passReset <= 1'b0;
118             wait_timer <= 0;
119         end
120 
121     endcase
122   end
123 end
124 endmodule
125 
```

Figure 27: Game Controller Module half of Access Controller

Appendix G: Legacy Verilog Code

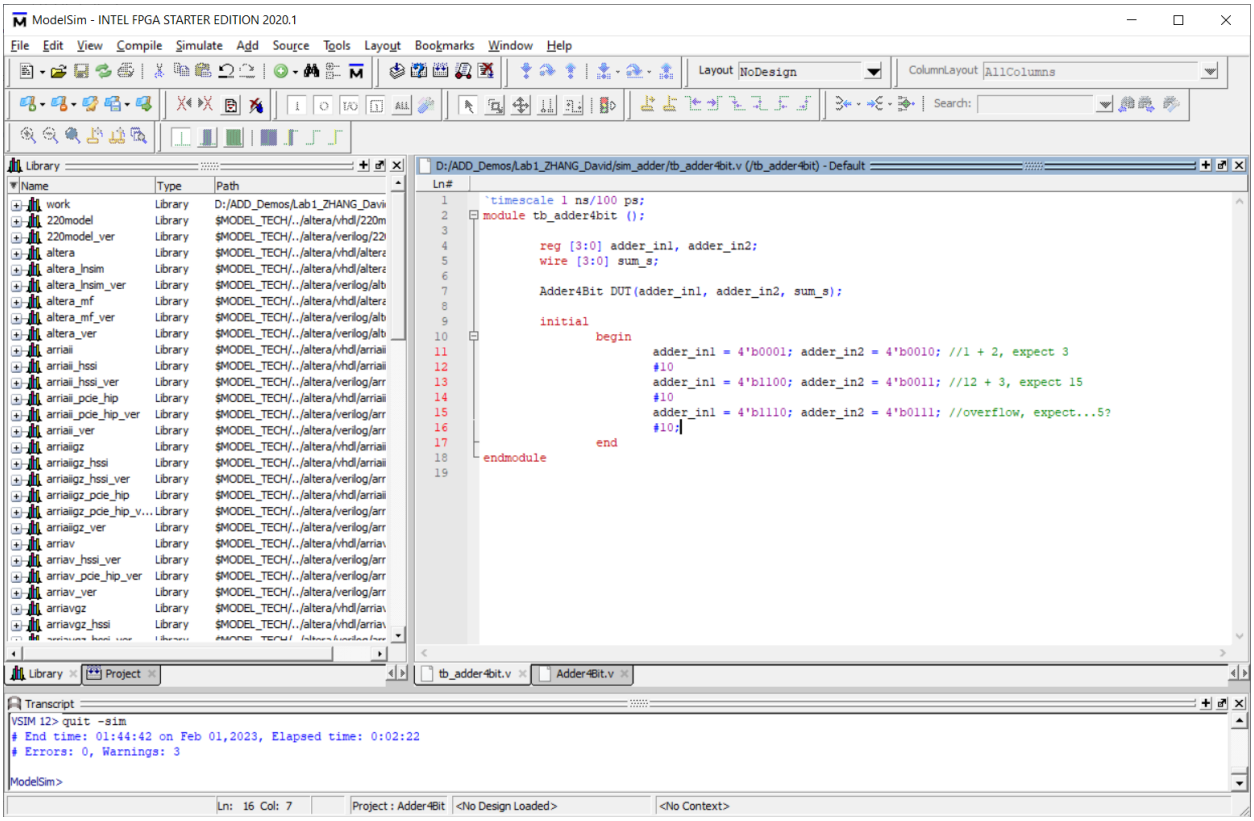


Figure 28: Module Definition for Adder

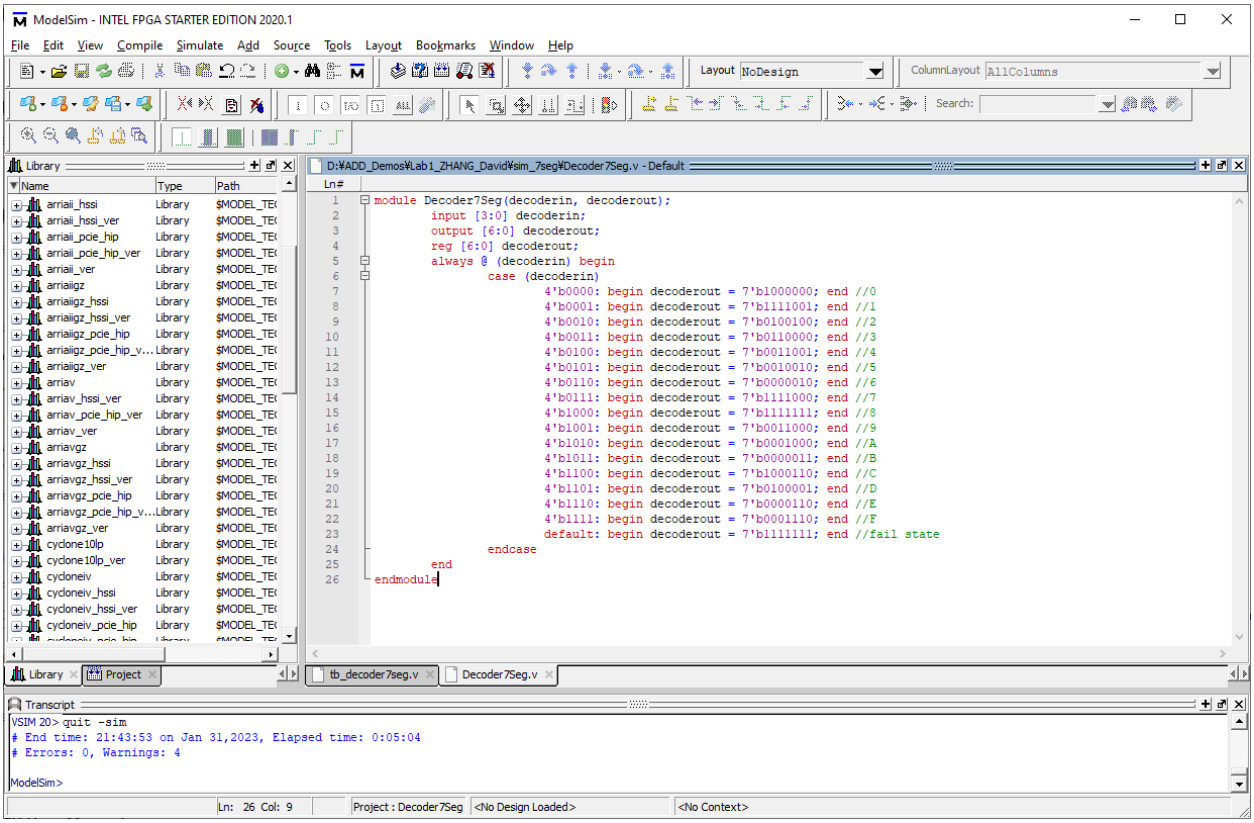


Figure 29: Module Definition for 7-Segment Decoder

```
1  module ButtonShaper (D, Q, rst, clk);
2          input D, rst, clk;
3          output Q;
4          reg Q;
5          parameter INIT = 0, PULSE = 1, WAIT = 2;
6
7          reg[2:0] State, StateNext;
8
9          //CombLogic
10         always @(State, D) begin
11                 case (State)
12                         INIT: begin
13                                 Q = 1'b0;
14                                 if (D == 1'b0)
15                                         StateNext = PULSE;
16                                 else
17                                         StateNext = INIT;
18                         end
19                         PULSE: begin
20                                 Q = 1'b1;
21                                 StateNext = WAIT;
22                         end
23                         WAIT: begin
24                                 Q = 1'b0;
25                                 if (D == 1'b1)
26                                         StateNext = INIT;
27                                 else
28                                         StateNext = WAIT;
29                         end
30                         default: begin
31                                 Q = 1'b0;
32                                 StateNext = INIT;
33                         end
34                 endcase
35         end
```

Figure 30: Module Definition for Button Shaper

```
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                                 begin
16                                         if (Load == 1'b1)
17                                          begin
18                                                 D_out <= D_in;
19                                          end
20                                 end
21                 end
22
23
24
25 endmodule
26
```

Figure 31: Module Definition for Load Register