LAB 3: HARDWARE-BASED TRUE RANDOM NUMBER GENERATOR AND TIMER 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, however the logout feature has been removed as a result of button saturation. 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. Finally, 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).

System Architecture

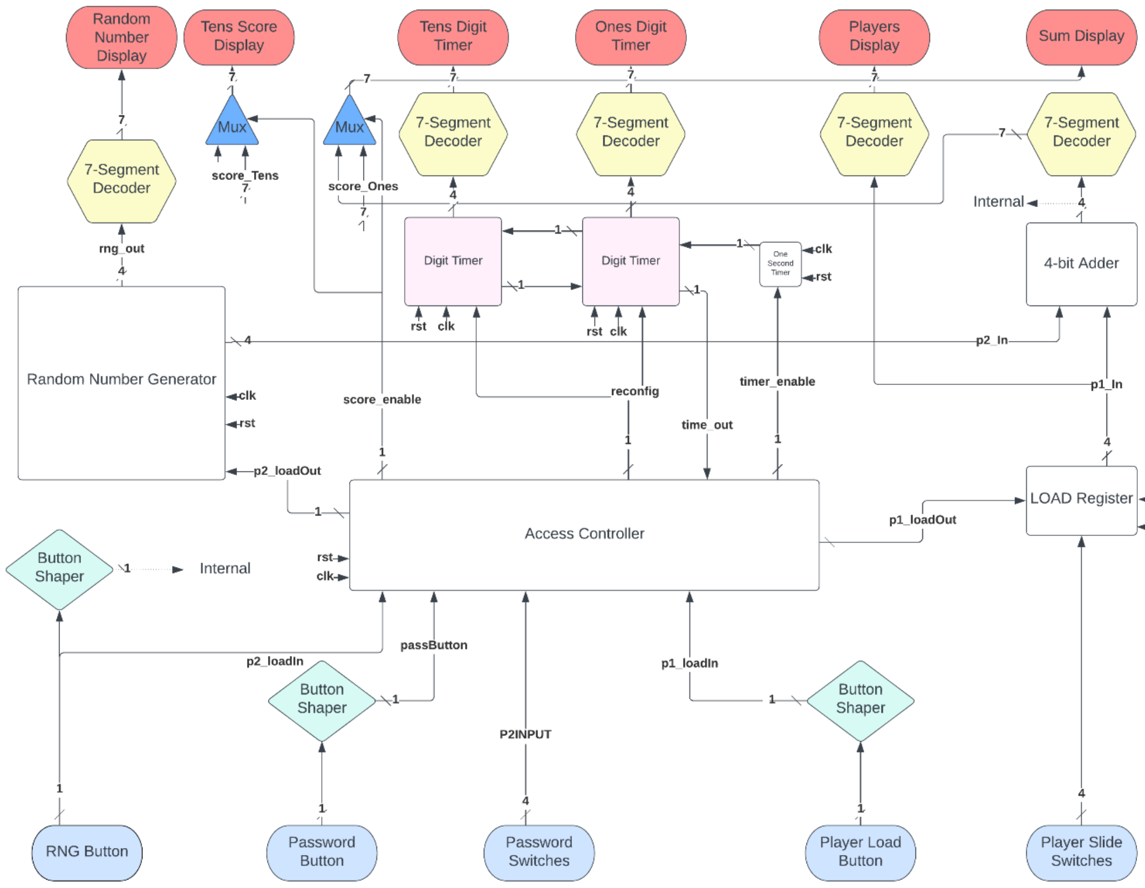


Figure 1: Overall System Architecture

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.

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.

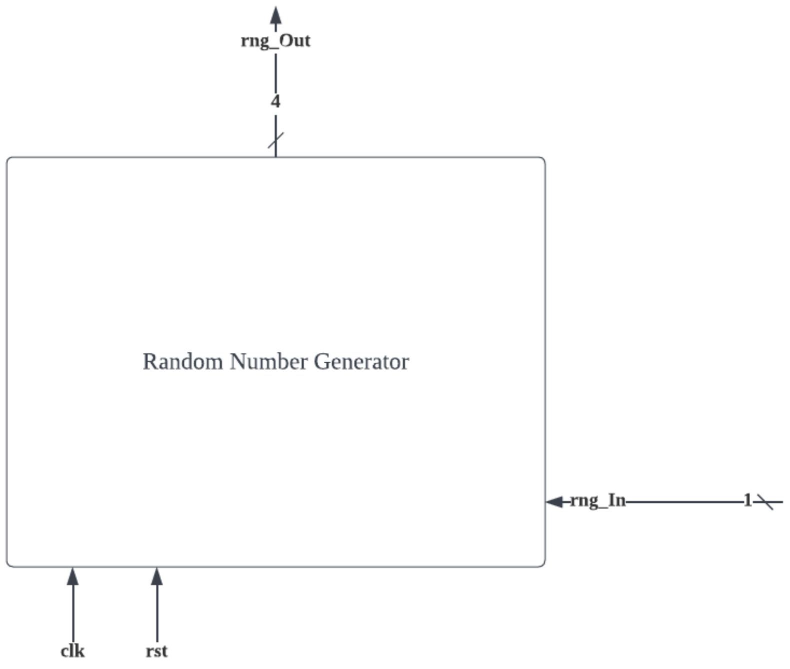


Figure 2: Module Depiction of RNG

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 incrementing an internal register for each clock cycle that it detects input is high and rolling over back to 0 at 4-bit maximum. 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.

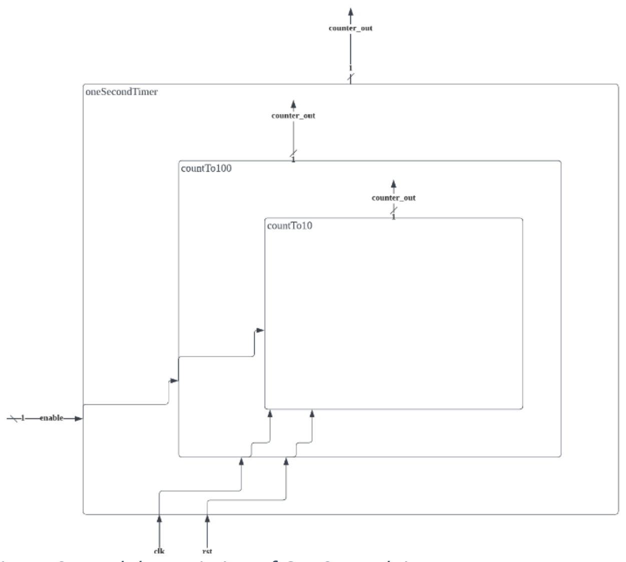


Figure 3: Module Depiction of OneSecondTimer

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.

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.

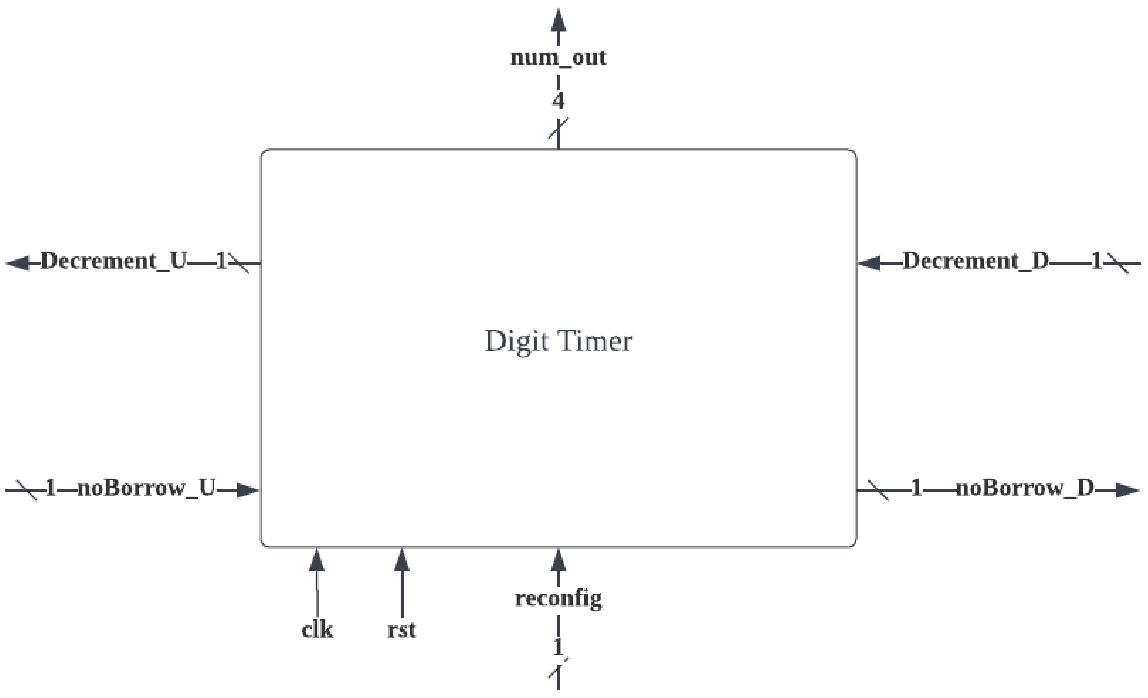


Figure 4: Module Depiction of Digit Timer

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.

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.

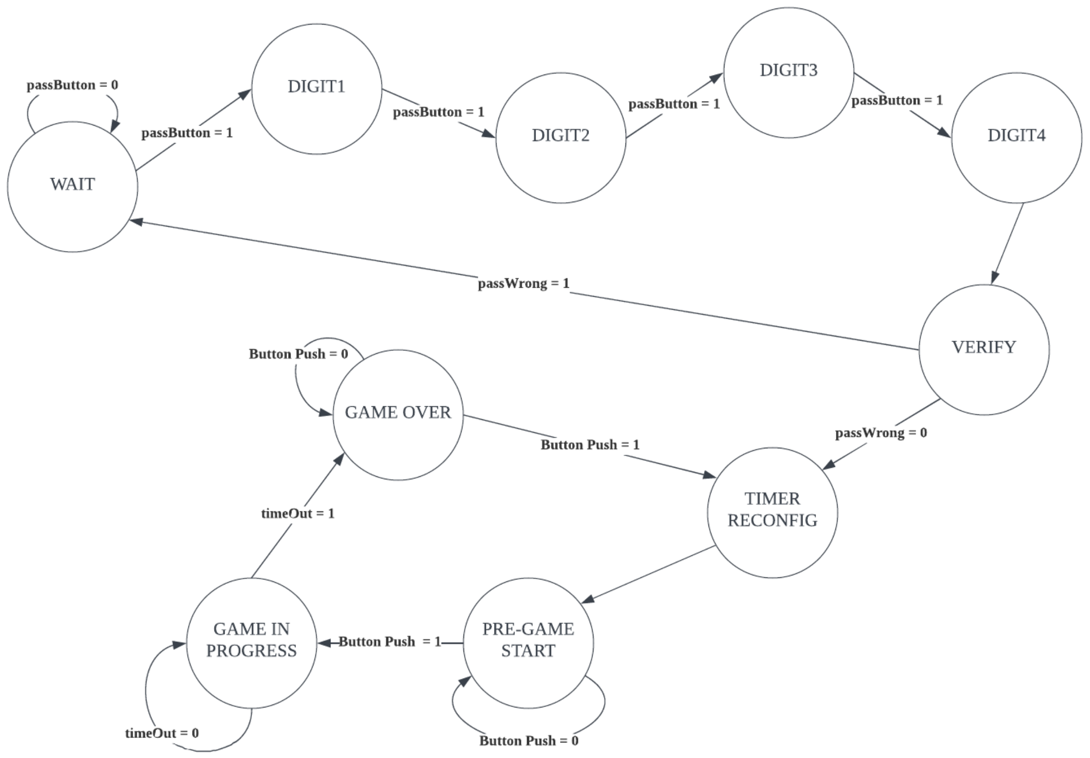


Figure 5: Finite State Machine for Modified Access Controller

Depicted in Figure 5 is the modified FSM for the Access Controller. After VERIFY, 4 new states have been added that correspond to correct operation of the game. 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 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.

Simulation Results

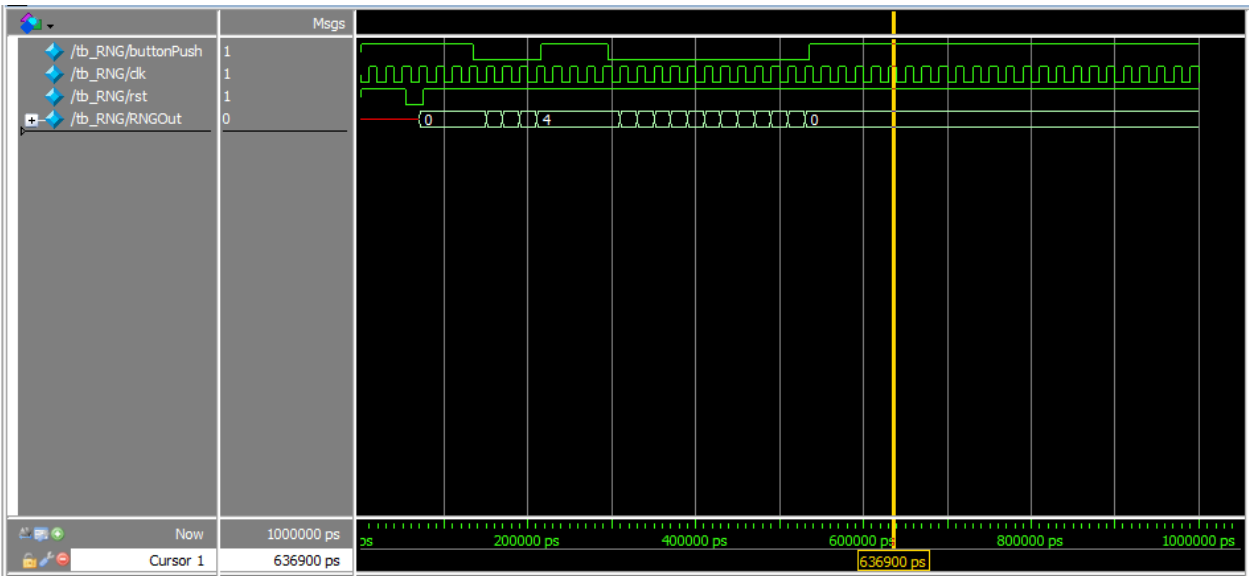
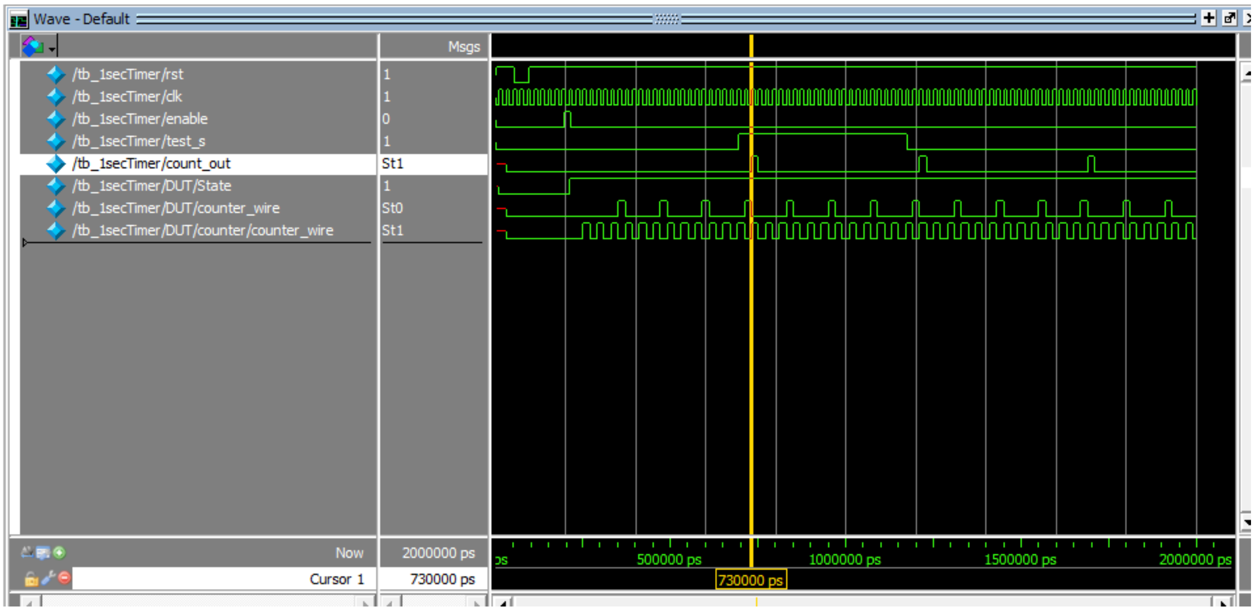


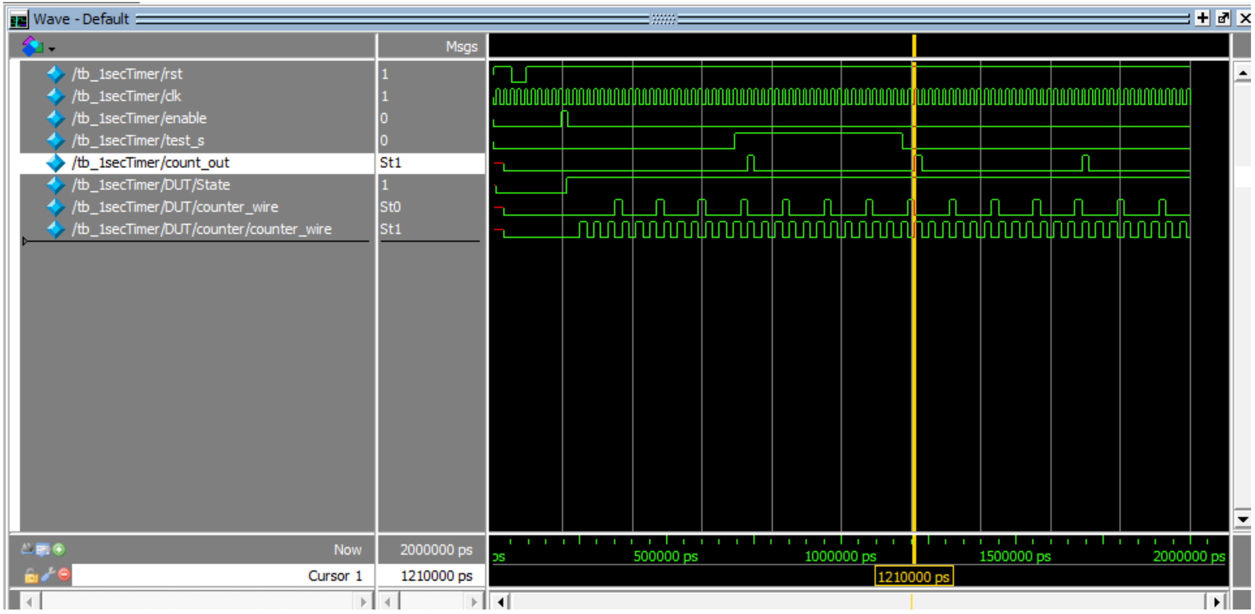
Figure 6: Simulation Results of RNG Module

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.



```
/tb_1secTimer/rst                          1
/tb_1secTimer/clk                          1
/tb_1secTimer/enable                       0
/tb_1secTimer/test_s                       1
/tb_1secTimer/count_out                    St1
/tb_1secTimer/DUT/State                    1
/tb_1secTimer/DUT/counter_wire             St0
/tb_1secTimer/DUT/counter/counter_wire     St1

Now        2000000 ps
Cursor 1   730000 ps
```

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



```
/tb_1secTimer/rst                          1
/tb_1secTimer/clk                          1
/tb_1secTimer/enable                       0
/tb_1secTimer/test_s                       0
/tb_1secTimer/count_out                    St1
/tb_1secTimer/DUT/State                    1
/tb_1secTimer/DUT/counter_wire             St0
/tb_1secTimer/DUT/counter/counter_wire     St1

Now        2000000 ps
Cursor 1   1210000 ps
```

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.

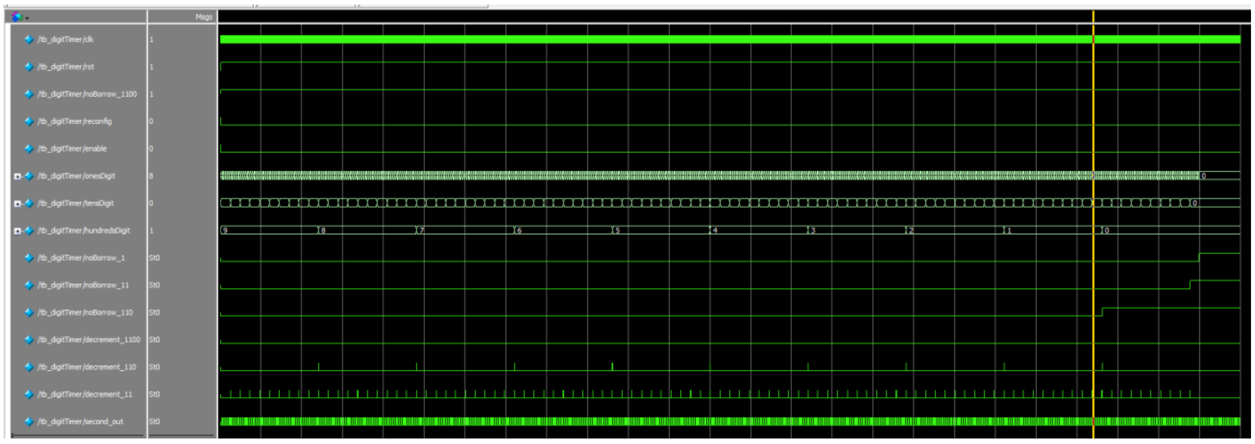

Msgs
/tb_digitTimer/clk 1
/tb_digitTimer/rst 1
/tb_digitTimer/noBorrow_1100 1
/tb_digitTimer/reconfig 0
/tb_digitTimer/enable 0
/tb_digitTimer/onesDigit 8
/tb_digitTimer/tensDigit 0
/tb_digitTimer/hundredsDigit 1
/tb_digitTimer/noBorrow_1 St0
/tb_digitTimer/noBorrow_11 St0
/tb_digitTimer/noBorrow_110 St0
/tb_digitTimer/decrement_1100 St0
/tb_digitTimer/decrement_110 St0
/tb_digitTimer/decrement_11 St0
/tb_digitTimer/second_out St0
9
8
7
6
5
4
3
2
1
0

Figure 9: Output Waveform of 3 digitTimer Modules

Depicted above in Figure 9 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.

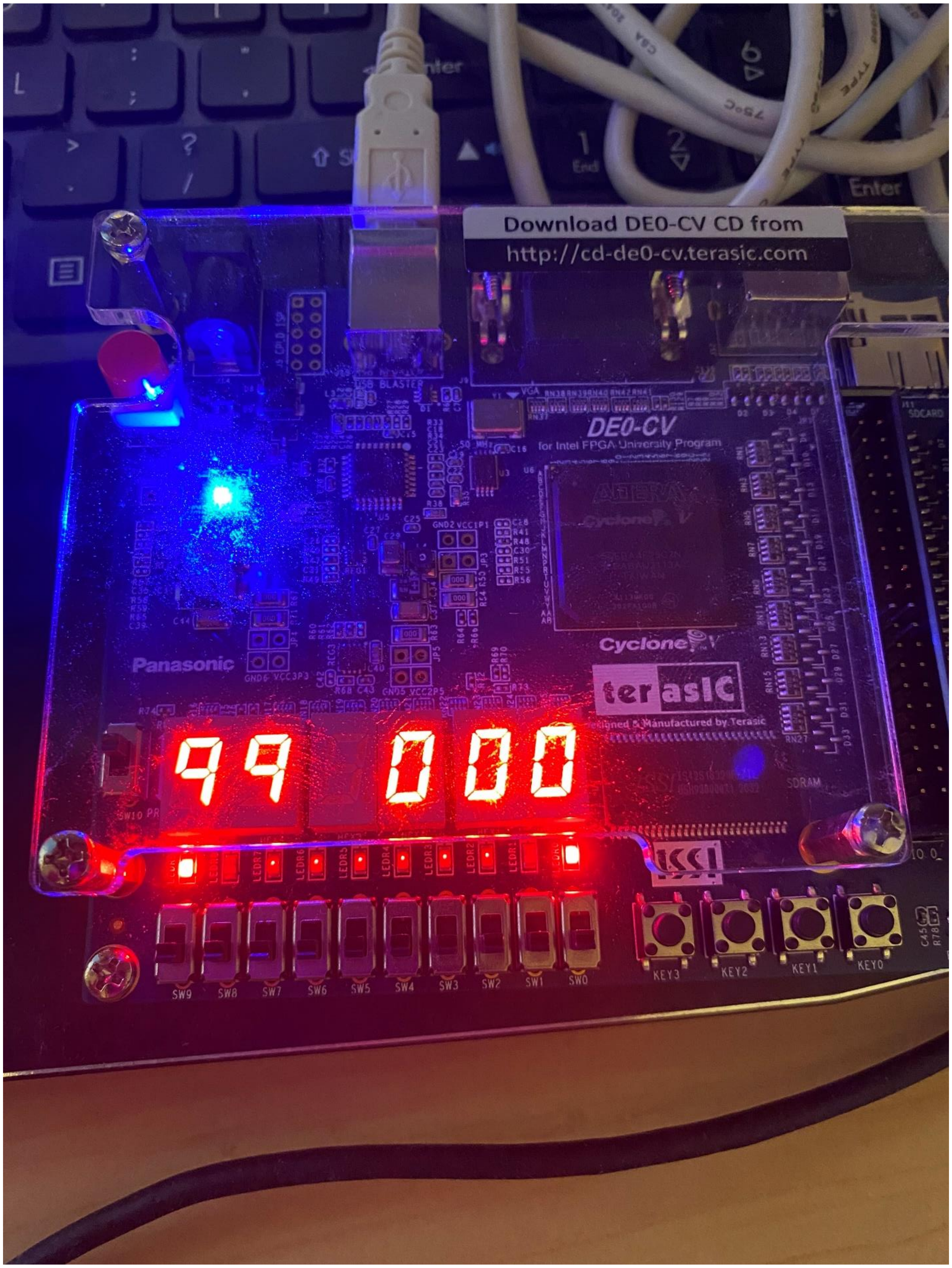


Figure 10: FPGA After Logging In: Pre-Game Start

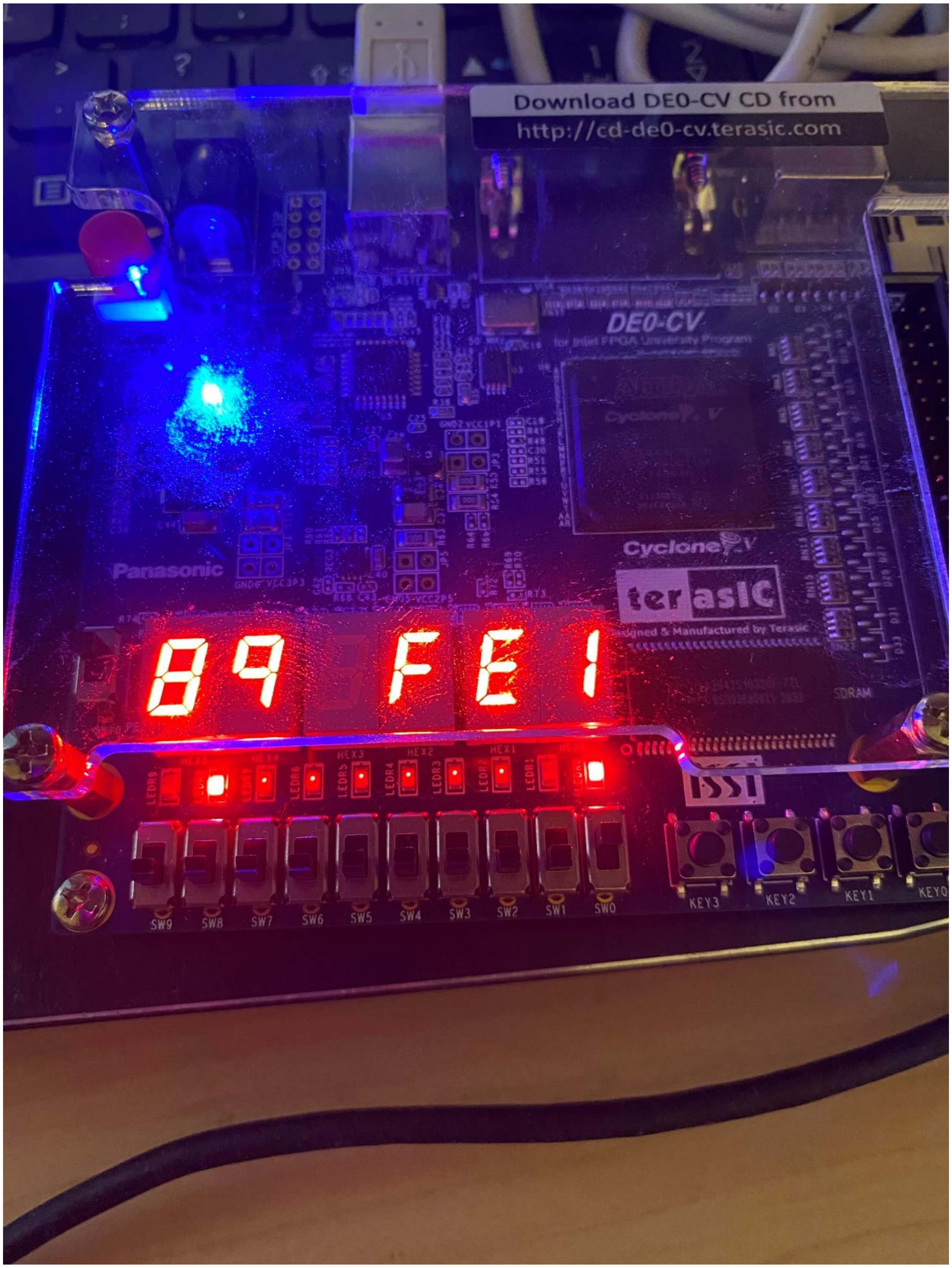


Figure 11: Game in Progress

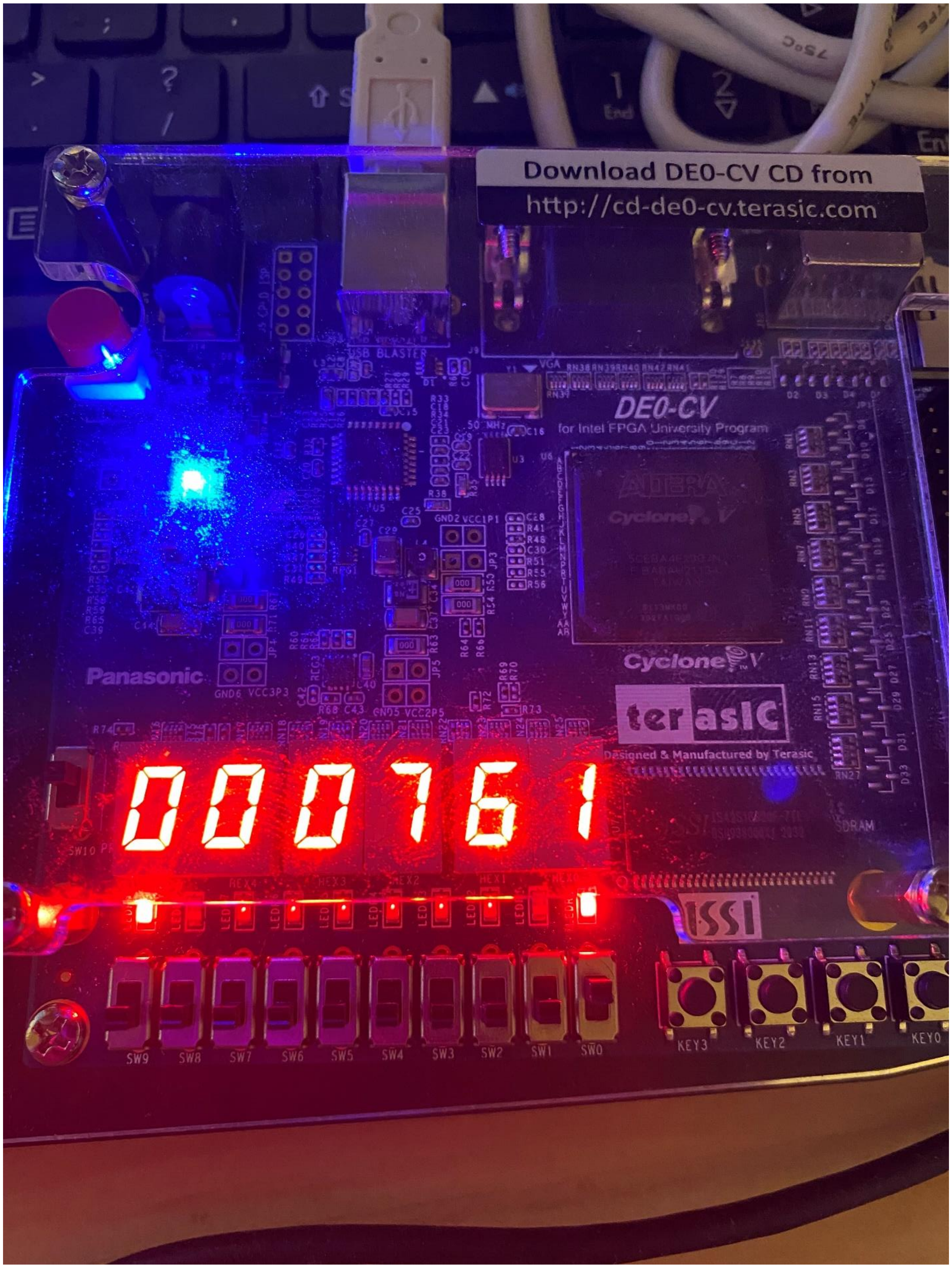


Figure 12: Game Over with Score Depicted

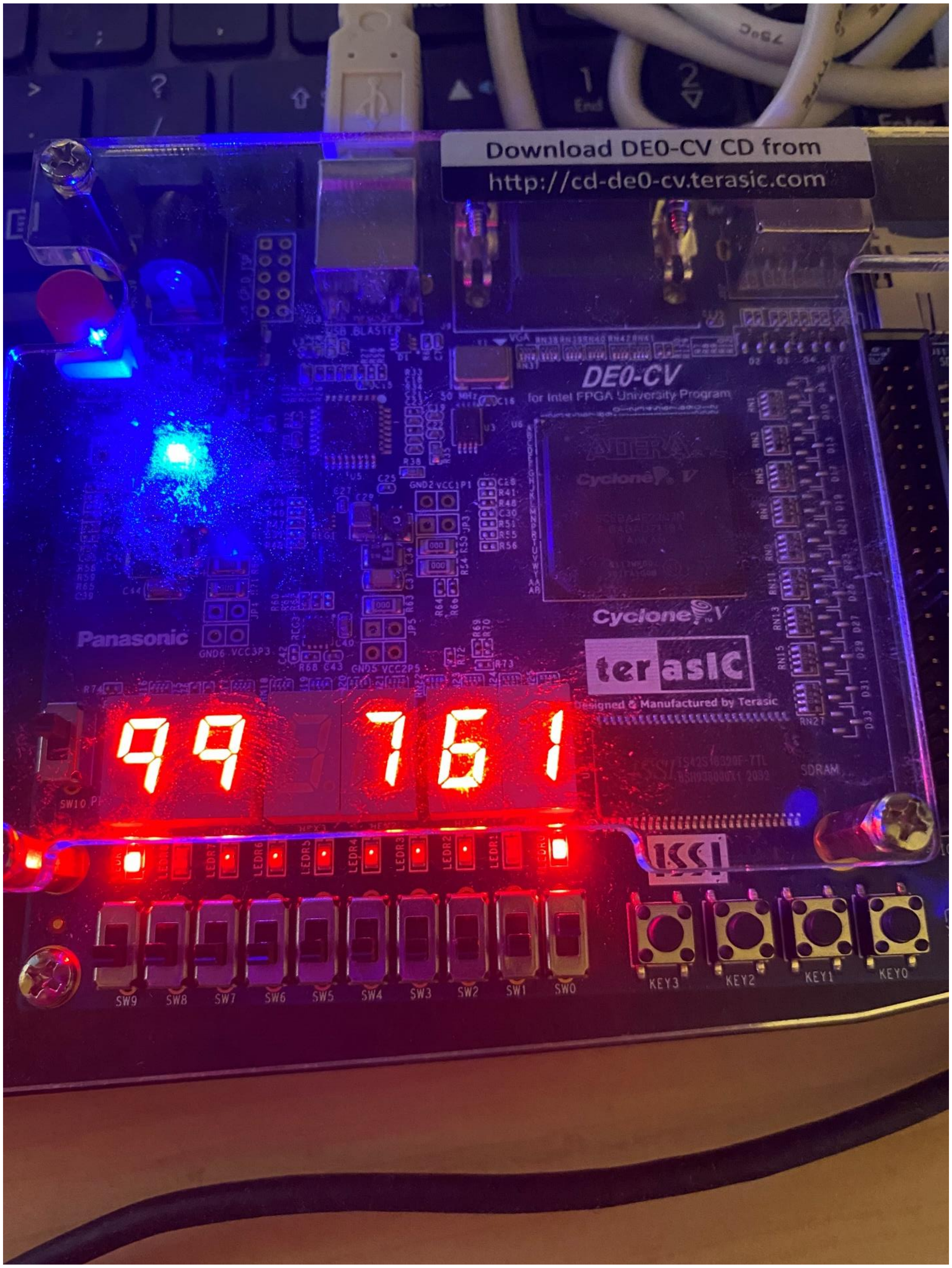


Figure 13: Pre-Game Start of Game 2

It is important to note that in Figure 12 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.

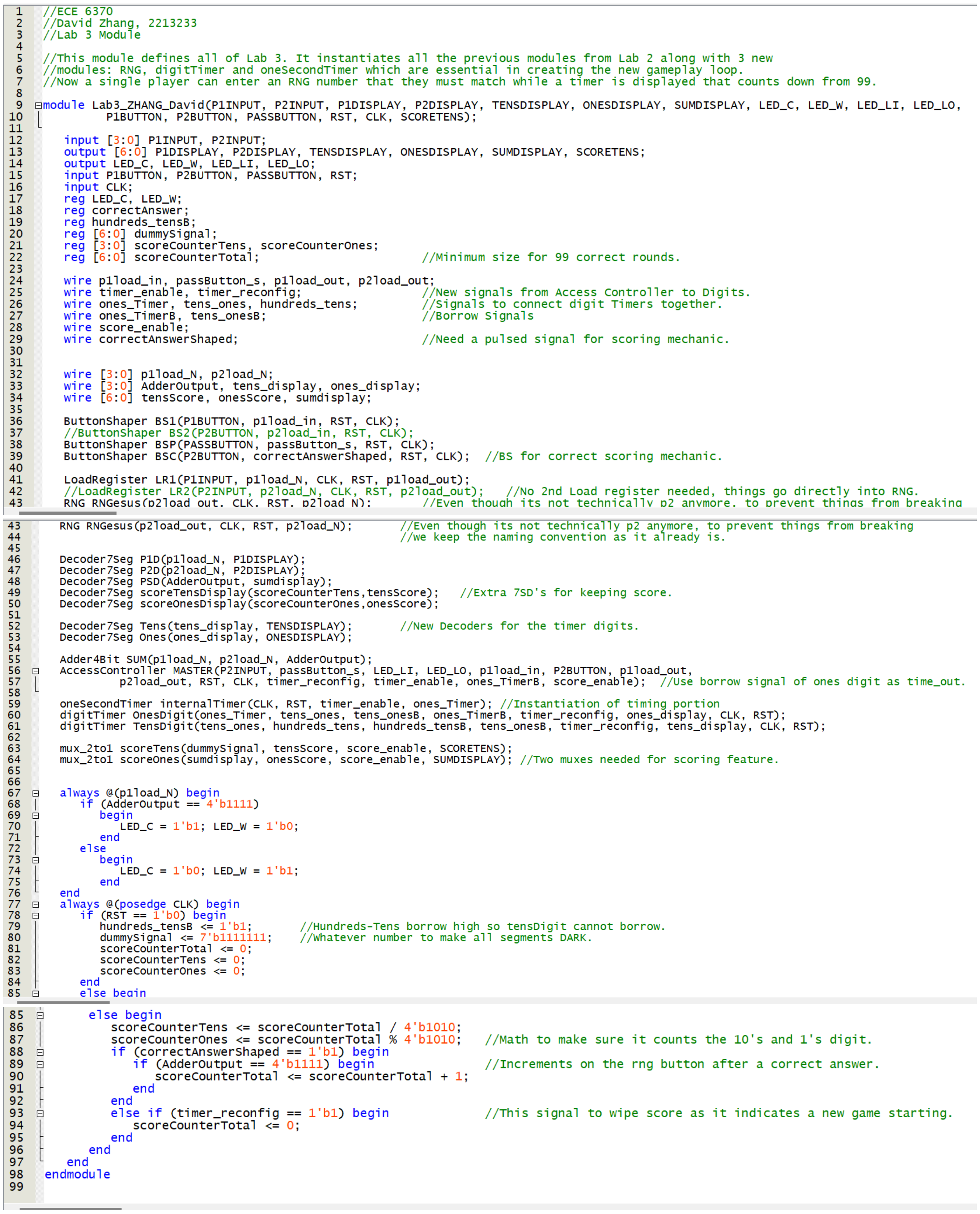
Conclusion

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.

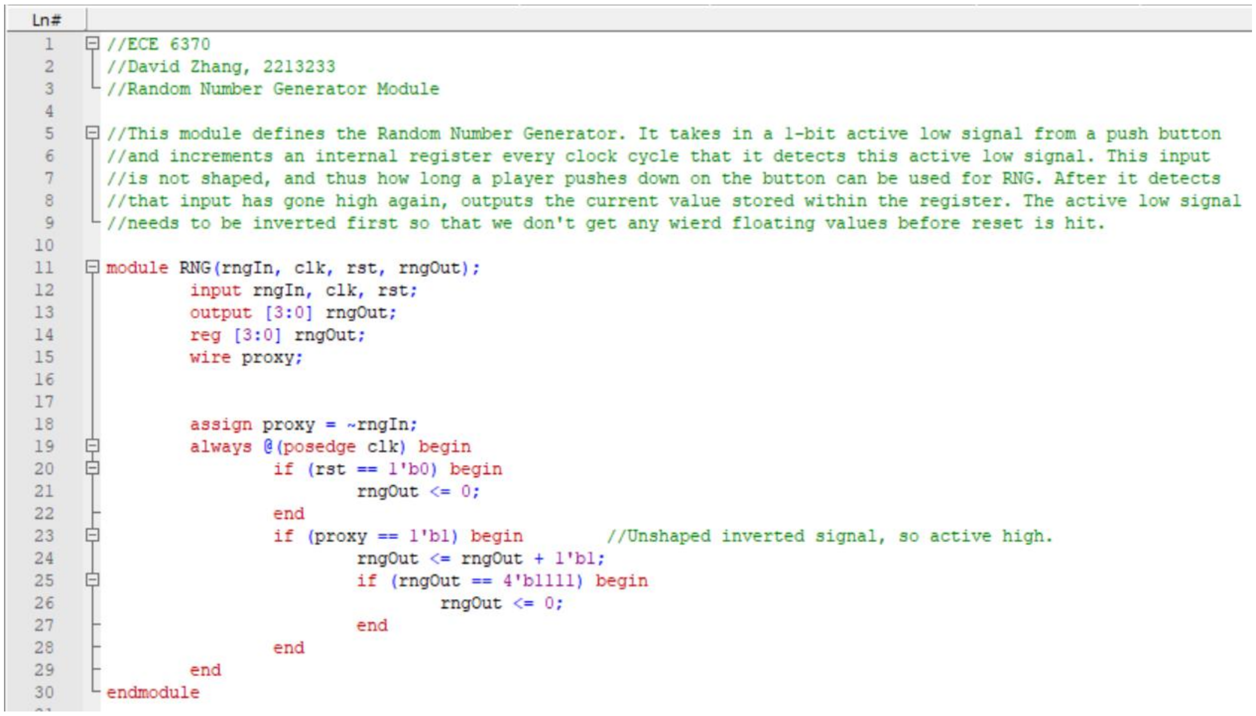
Appendix A: Verilog Code for Lab 3



```verilog
//ECE 6370
//David Zhang, 2213233
//Lab 3 Module

//This module defines all of Lab 3. It instantiates all the previous modules from Lab 2 along with 3 new
//modules: RNG, digitTimer and oneSecondTimer which are essential in creating the new gameplay loop.
//Now a single player can enter an RNG number that they must match while a timer is displayed that counts down from 99.

module Lab3_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 [3:0] p1load_N, p2load_N;
   wire [3:0] AdderOutput, tens_display, ones_display;
   wire [6:0] tensScore, onesScore, sumdisplay;

   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 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);
   AccessController 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);  //Use borrow signal of ones digit as time_out.

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


   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
```

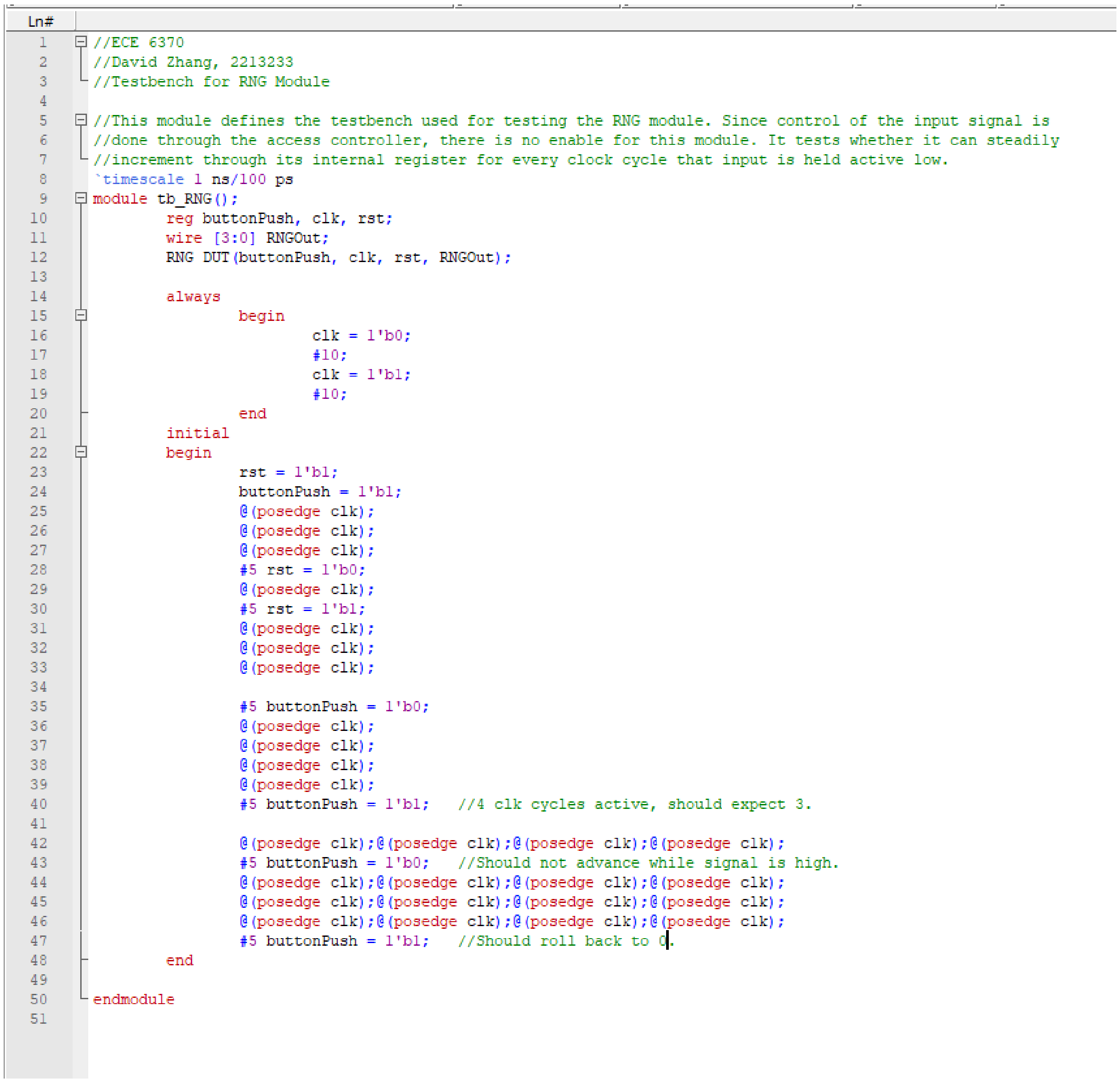
Figure 14: Module Definition for Lab 3 in Quartus

Appendix B: Verilog Code for RNG



```
//ECE 6370
//David Zhang, 2213233
//Random Number Generator Module

//This module defines the Random Number Generator. It takes in a 1-bit active low signal from a push button
//and increments an internal register every clock cycle that it detects this active low signal. This input
//is not shaped, and thus how long a player pushes down on the button can be used for RNG. After it detects
//that input has gone high again, outputs the current value stored within the register. The active low signal
//needs to be inverted first so that we don't get any wierd floating values before reset is hit.

module RNG(rngIn, clk, rst, rngOut);
	input rngIn, clk, rst;
	output [3:0] rngOut;
	reg [3:0] rngOut;
	wire proxy;


	assign proxy = ~rngIn;
	always @(posedge clk) begin
		if (rst == 1'b0) begin
			rngOut <= 0;
		end
		if (proxy == 1'b1) begin	//Unshaped inverted signal, so active high.
			rngOut <= rngOut + 1'b1;
			if (rngOut == 4'b1111) begin
				rngOut <= 0;
			end
		end
	end
endmodule
```

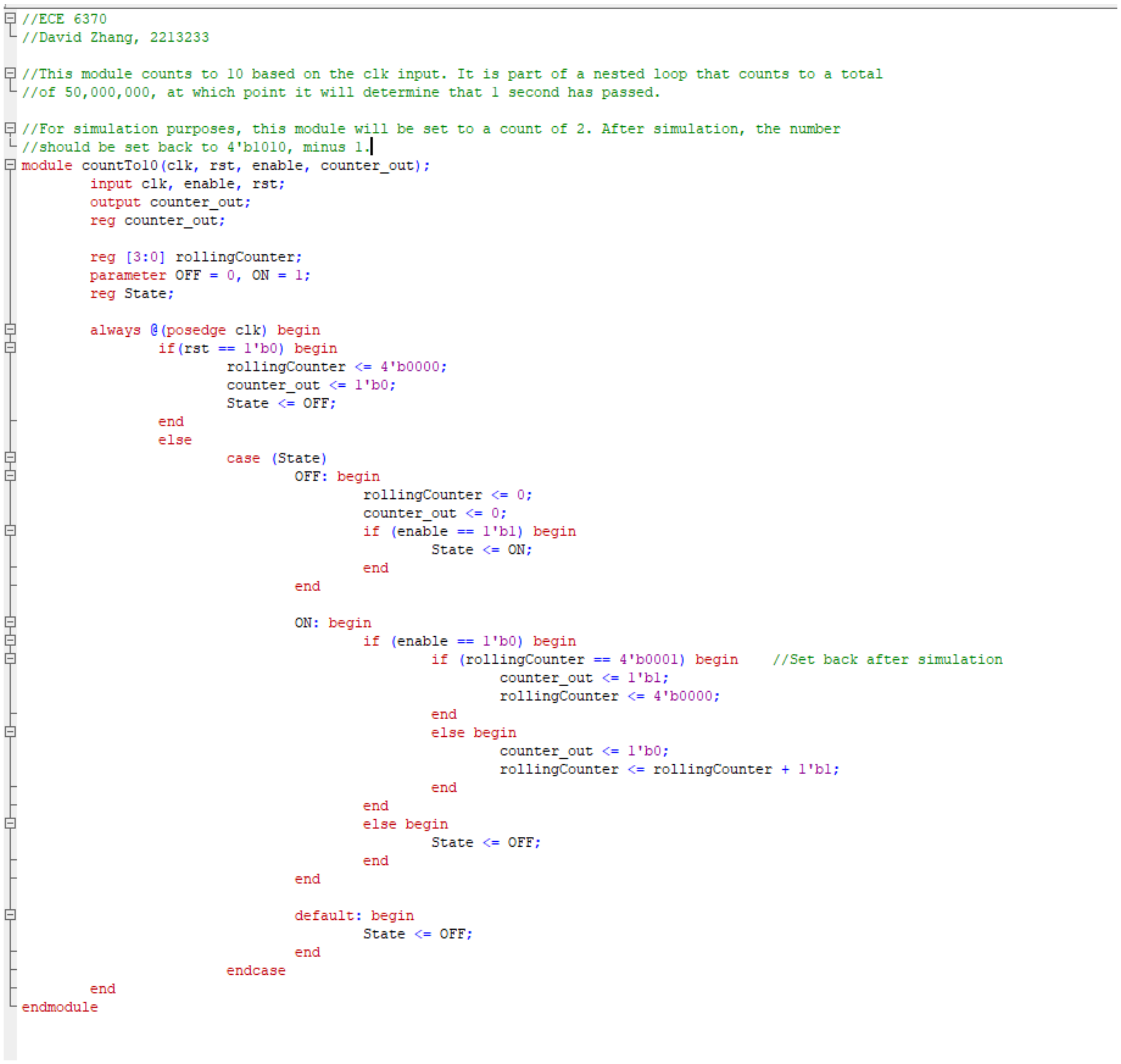
Figure 15: Module Definition for RNG



```verilog
//ECE 6370
//David Zhang, 2213233
//Testbench for RNG Module

//This module defines the testbench used for testing the RNG 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 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);
                #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);
                #5 buttonPush = 1'b1;   //Should roll back to 0.
        end

endmodule
```

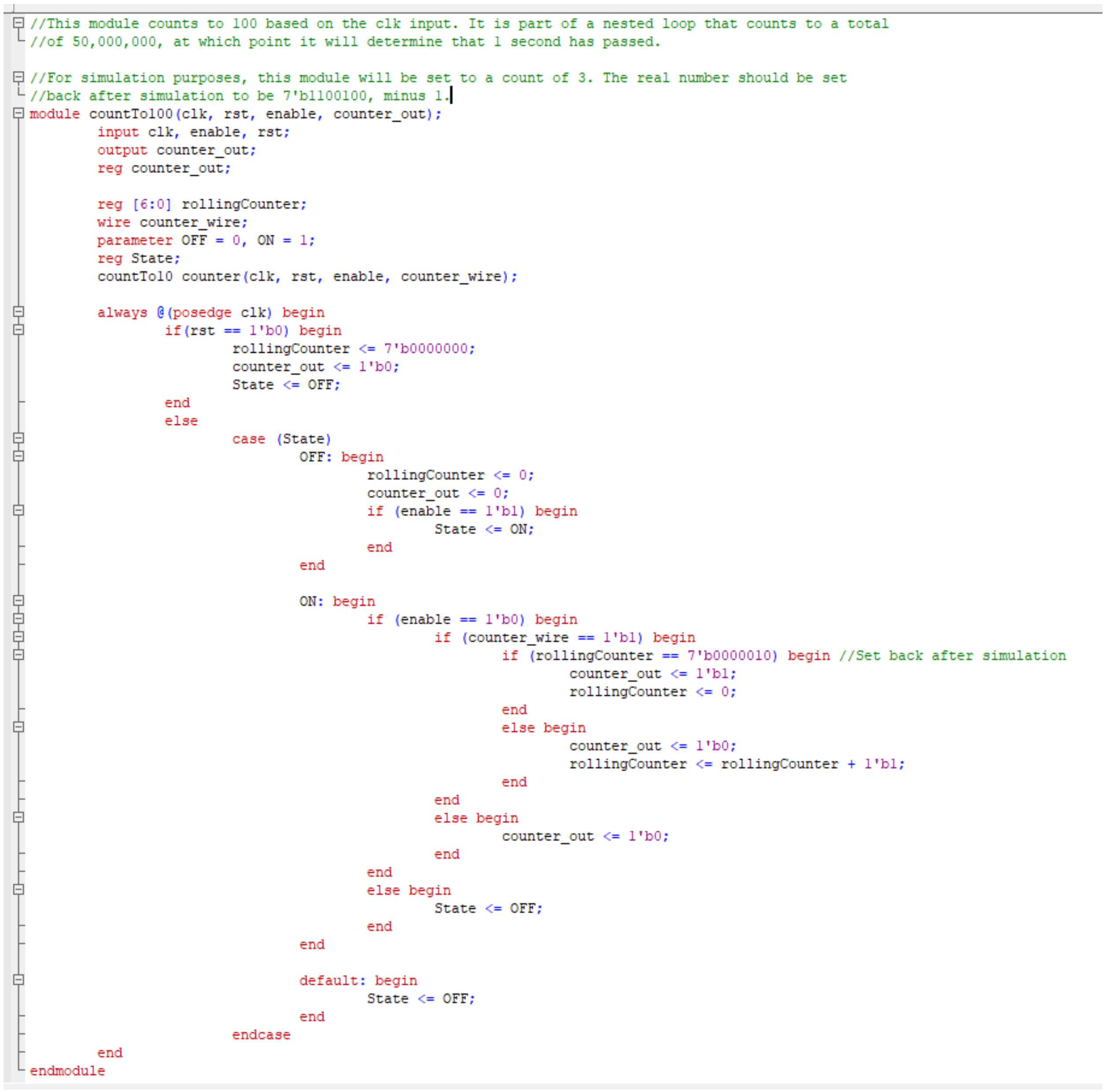
Figure 16: Testbench Definition for RNG

Appendix C: Verilog Code for 1 second Timer



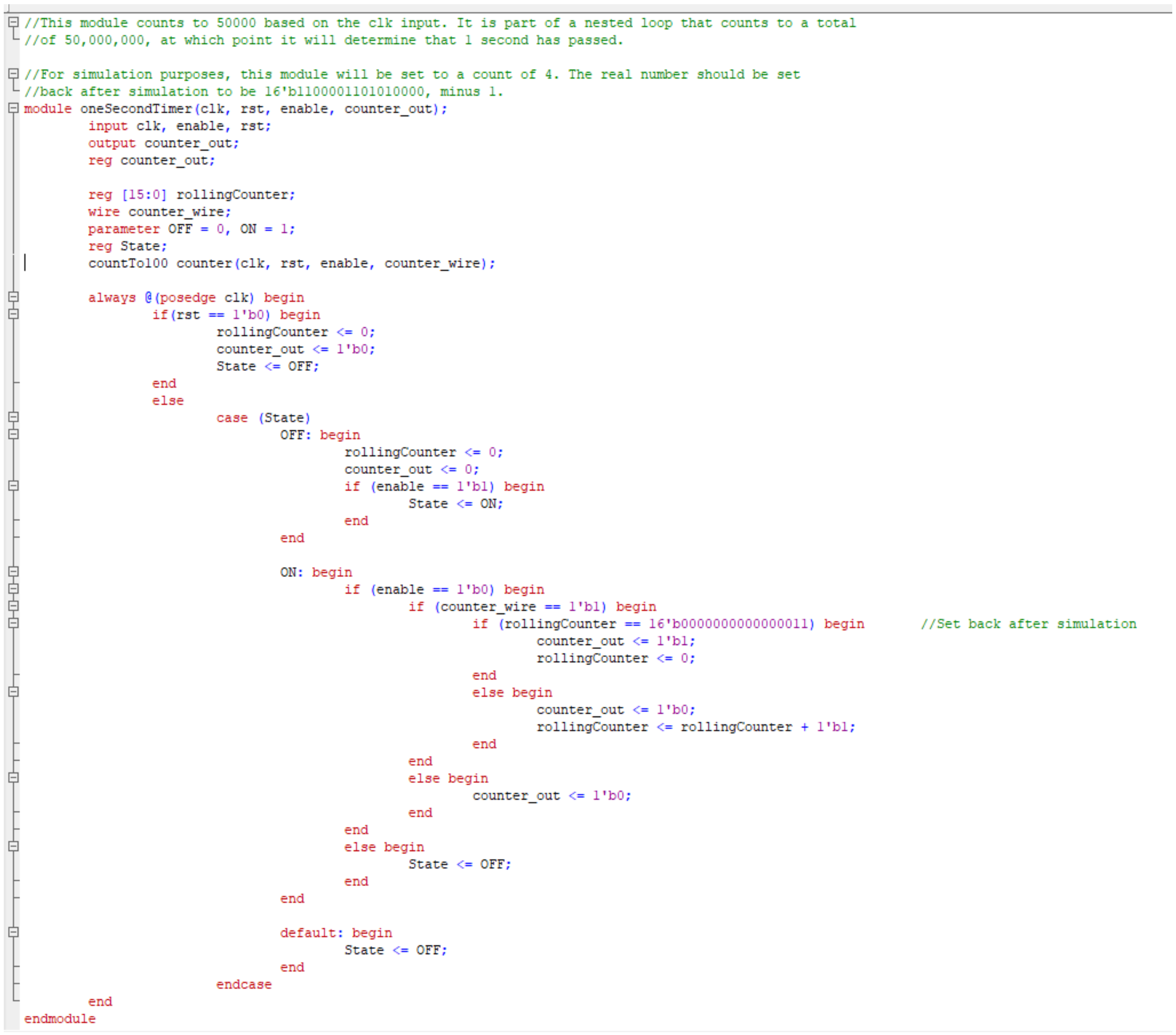
```
//ECE 6370
//David Zhang, 2213233

//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 17: Module Definition for countTo10, the 3rd nested module.



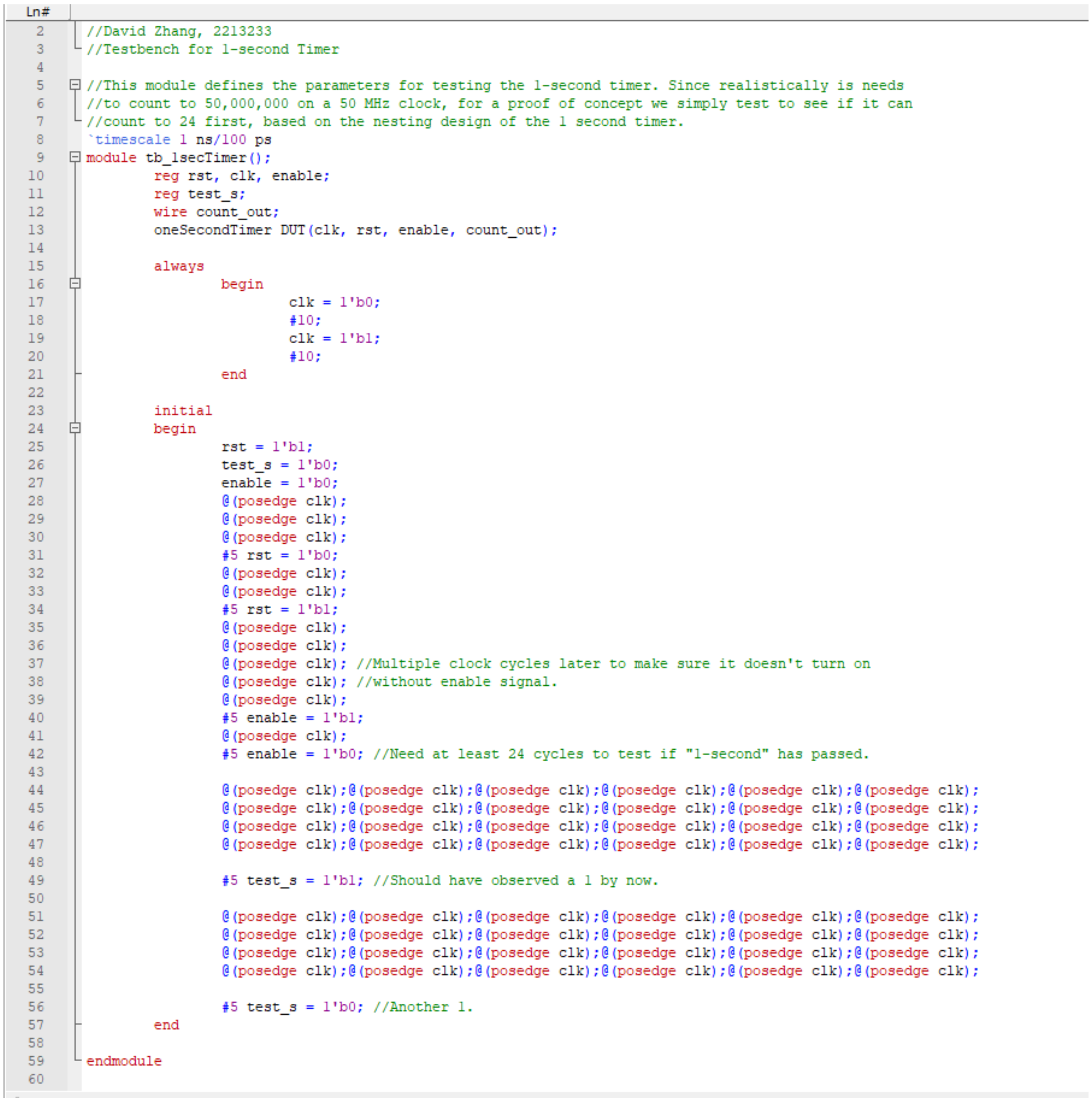
```
//This module counts to 100 based on the clk input. It is part of a nested loop that counts to a total
//of 50,000,000, at which point it will determine that 1 second has passed.

//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 18: Module Definition for countTo100, the 2nd nested module



```verilog
//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
```

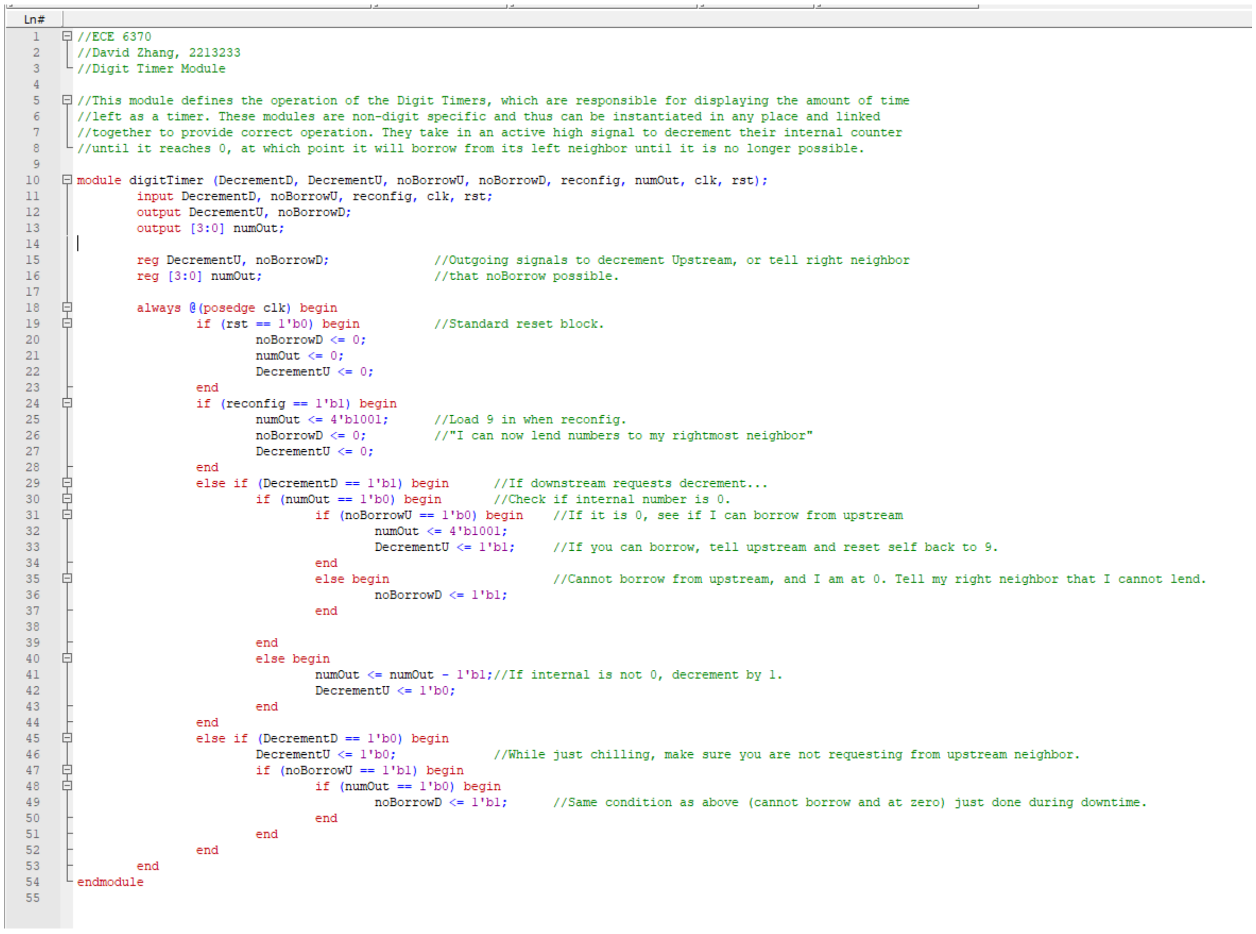
Figure 19: Module Definition for 1 second Timer, the highest module



```
//David Zhang, 2213233
//Testbench for 1-second Timer

//This module defines the parameters for testing the 1-second timer. Since realistically is needs
//to count to 50,000,000 on a 50 MHz clock, for a proof of concept we simply test to see if it can
//count to 24 first, based on the nesting design of the 1 second timer.
`timescale 1 ns/100 ps
module tb_1secTimer();
        reg rst, clk, enable;
        reg test_s;
        wire count_out;
        oneSecondTimer DUT(clk, rst, enable, count_out);

        always
                begin
                        clk = 1'b0;
                        #10;
                        clk = 1'b1;
                        #10;
                end

        initial
        begin
                rst = 1'b1;
                test_s = 1'b0;
                enable = 1'b0;
                @(posedge clk);
                @(posedge clk);
                @(posedge clk);
                #5 rst = 1'b0;
                @(posedge clk);
                @(posedge clk);
                #5 rst = 1'b1;
                @(posedge clk);
                @(posedge clk);
                @(posedge clk); //Multiple clock cycles later to make sure it doesn't turn on
                @(posedge clk); //without enable signal.
                @(posedge clk);
                #5 enable = 1'b1;
                @(posedge clk);
                #5 enable = 1'b0; //Need at least 24 cycles to test if "1-second" has passed.

                @(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);@(posedge clk);@(posedge clk);
                @(posedge clk);@(posedge clk);@(posedge clk);@(posedge clk);@(posedge clk);@(posedge clk);

                #5 test_s = 1'b1; //Should have observed a 1 by now.

                @(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);@(posedge clk);@(posedge clk);
                @(posedge clk);@(posedge clk);@(posedge clk);@(posedge clk);@(posedge clk);@(posedge clk);

                #5 test_s = 1'b0; //Another 1.
        end

endmodule
```

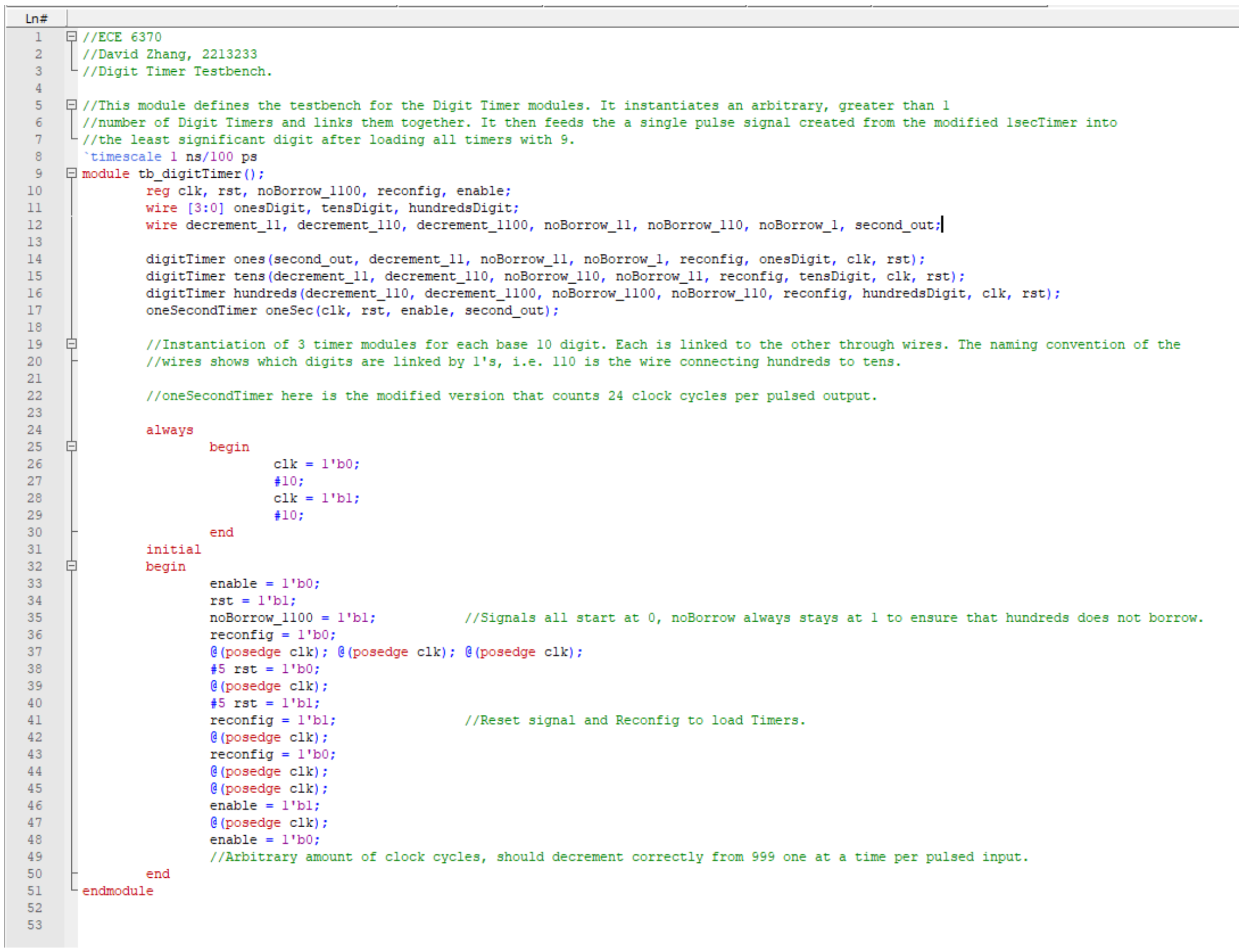
Figure 20: Testbench Definition for the Modified 1 second Timer

Appendix D: Verilog Code for Digit Timer



```verilog
//ECE 6370
//David Zhang, 2213233
//Digit Timer Module

//This module defines the operation of the Digit Timers, which are responsible for displaying the amount of time
//left as a timer. These modules are non-digit specific and thus can be instantiated in any place and linked
//together to provide correct operation. They take in an active high signal to decrement their internal counter
//until it reaches 0, at which point it will borrow from its left neighbor until it is no longer possible.

module digitTimer (DecrementD, DecrementU, noBorrowU, noBorrowD, reconfig, numOut, clk, rst);
        input DecrementD, noBorrowU, reconfig, clk, rst;
        output DecrementU, noBorrowD;
        output [3:0] numOut;

        reg DecrementU, noBorrowD;              //Outgoing signals to decrement Upstream, or tell right neighbor
        reg [3:0] numOut;                       //that noBorrow possible.

        always @(posedge clk) begin
                if (rst == 1'b0) begin          //Standard reset block.
                        noBorrowD <= 0;
                        numOut <= 0;
                        DecrementU <= 0;
                end
                if (reconfig == 1'b1) begin
                        numOut <= 4'b1001;      //Load 9 in when reconfig.
                        noBorrowD <= 0;         //"I can now lend numbers to my rightmost neighbor"
                        DecrementU <= 0;
                end
                else if (DecrementD == 1'b1) begin      //If downstream requests decrement...
                        if (numOut == 1'b0) begin       //Check if internal number is 0.
                                if (noBorrowU == 1'b0) begin    //If it is 0, see if I can borrow from upstream
                                        numOut <= 4'b1001;
                                        DecrementU <= 1'b1;     //If you can borrow, tell upstream and reset self back to 9.
                                end
                                else begin                      //Cannot borrow from upstream, and I am at 0. Tell my right neighbor that I cannot lend.
                                        noBorrowD <= 1'b1;
                                end

                        end
                        else begin
                                numOut <= numOut - 1'b1;//If internal is not 0, decrement by 1.
                                DecrementU <= 1'b0;
                        end
                end
                else if (DecrementD == 1'b0) begin
                        DecrementU <= 1'b0;             //While just chilling, make sure you are not requesting from upstream neighbor.
                        if (noBorrowU == 1'b1) begin
                                if (numOut == 1'b0) begin
                                        noBorrowD <= 1'b1;      //Same condition as above (cannot borrow and at zero) just done during downtime.
                                end
                        end
                end
        end
endmodule
```

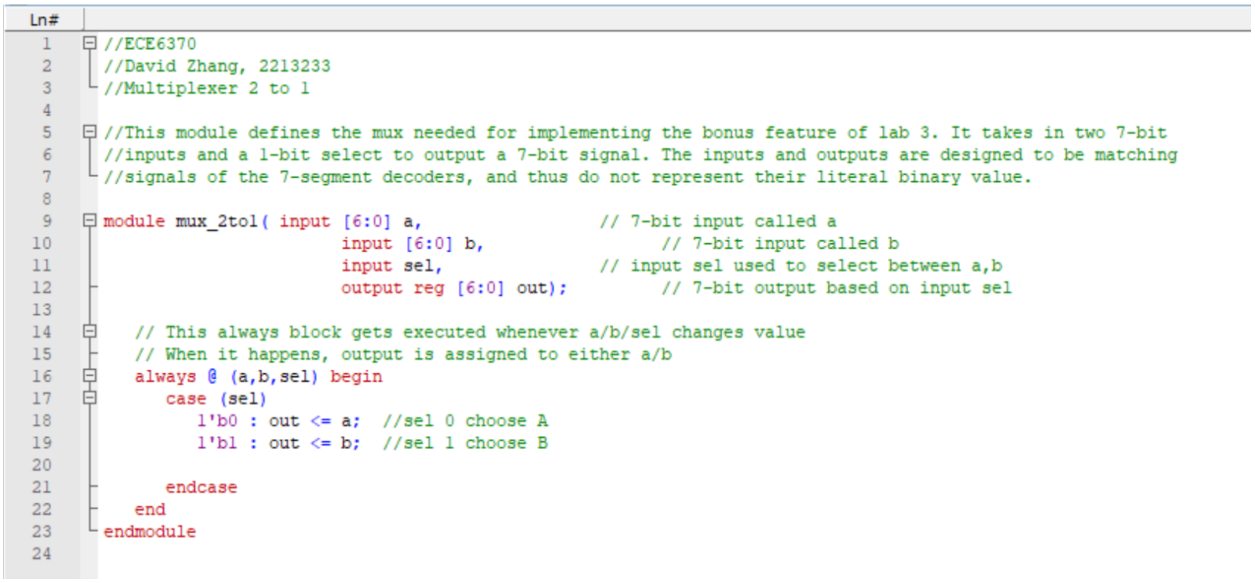
Figure 21: Module Definition for Digit Timer



```verilog
//ECE 6370
//David Zhang, 2213233
//Digit Timer Testbench.

//This module defines the testbench for the Digit Timer modules. It instantiates an arbitrary, greater than 1
//number of Digit Timers and links them together. It then feeds the a single pulse signal created from the modified 1secTimer into
//the least significant digit after loading all timers with 9.
`timescale 1 ns/100 ps
module tb_digitTimer();
        reg clk, rst, noBorrow_1100, reconfig, enable;
        wire [3:0] onesDigit, tensDigit, hundredsDigit;
        wire decrement_11, decrement_110, decrement_1100, noBorrow_11, noBorrow_110, noBorrow_1, second_out;

        digitTimer ones(second_out, decrement_11, noBorrow_11, noBorrow_1, reconfig, onesDigit, clk, rst);
        digitTimer tens(decrement_11, decrement_110, noBorrow_110, noBorrow_11, reconfig, tensDigit, clk, rst);
        digitTimer hundreds(decrement_110, decrement_1100, noBorrow_1100, noBorrow_110, reconfig, hundredsDigit, clk, rst);
        oneSecondTimer oneSec(clk, rst, enable, second_out);

        //Instantiation of 3 timer modules for each base 10 digit. Each is linked to the other through wires. The naming convention of the
        //wires shows which digits are linked by 1's, i.e. 110 is the wire connecting hundreds to tens.

        //oneSecondTimer here is the modified version that counts 24 clock cycles per pulsed output.

        always
                begin
                        clk = 1'b0;
                        #10;
                        clk = 1'b1;
                        #10;
                end
        initial
        begin
                enable = 1'b0;
                rst = 1'b1;
                noBorrow_1100 = 1'b1;           //Signals all start at 0, noBorrow always stays at 1 to ensure that hundreds does not borrow.
                reconfig = 1'b0;
                @(posedge clk); @(posedge clk); @(posedge clk);
                #5 rst = 1'b0;
                @(posedge clk);
                #5 rst = 1'b1;
                reconfig = 1'b1;                //Reset signal and Reconfig to load Timers.
                @(posedge clk);
                reconfig = 1'b0;
                @(posedge clk);
                @(posedge clk);
                enable = 1'b1;
                @(posedge clk);
                enable = 1'b0;
                //Arbitrary amount of clock cycles, should decrement correctly from 999 one at a time per pulsed input.
        end
endmodule
```

Figure 22: Testbench Definition for 3-instance Digit Timer

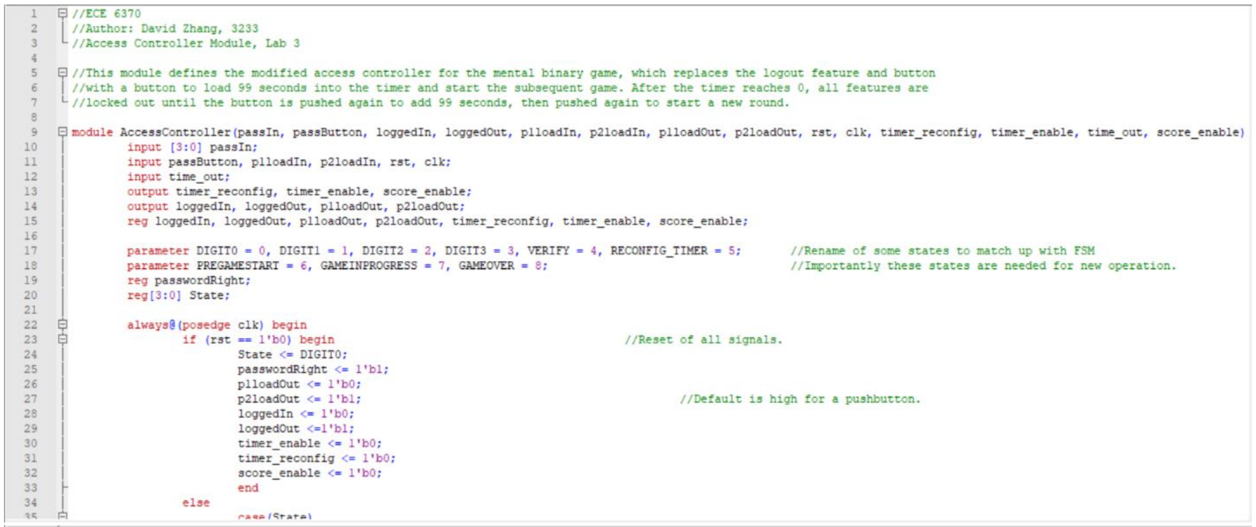
Appendix E: Verilog Code for 2-to-1 Multiplexer



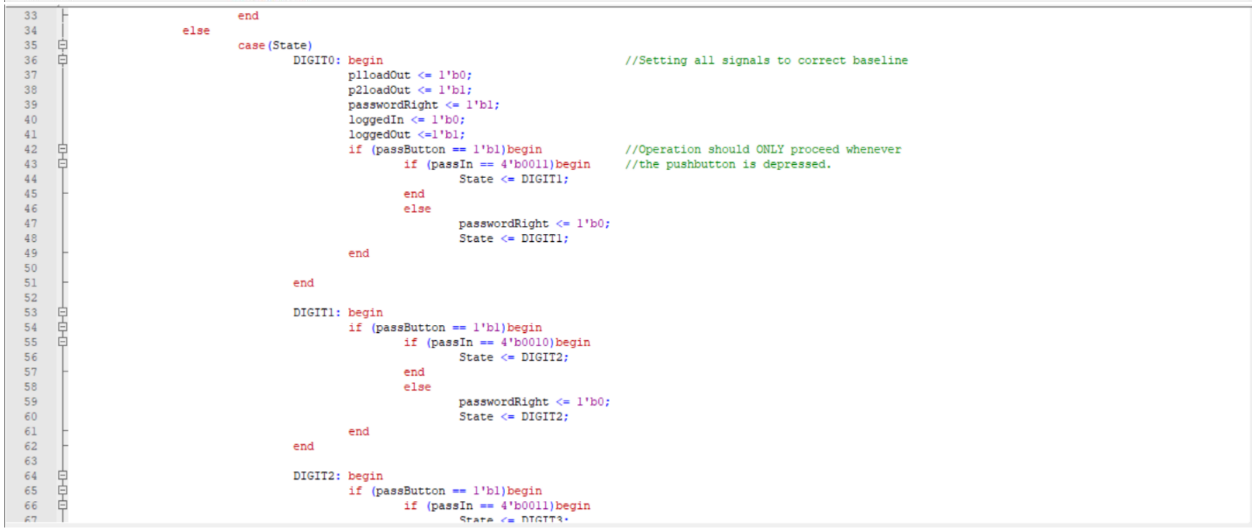
```verilog
//ECE6370
//David Zhang, 2213233
//Multiplexer 2 to 1

//This module defines the mux needed for implementing the bonus feature of lab 3. It takes in two 7-bit
//inputs and a 1-bit select to output a 7-bit signal. The inputs and outputs are designed to be matching
//signals of the 7-segment decoders, and thus do not represent their literal binary value.

module mux_2to1( input [6:0] a,                 // 7-bit input called a
                       input [6:0] b,                 // 7-bit input called b
                       input sel,               // input sel used to select between a,b
                       output reg [6:0] out);         // 7-bit output based on input sel

   // This always block gets executed whenever a/b/sel changes value
   // When it happens, output is assigned to either a/b
   always @ (a,b,sel) begin
      case (sel)
         1'b0 : out <= a;  //sel 0 choose A
         1'b1 : out <= b;  //sel 1 choose B

      endcase
   end
endmodule
```

Figure 23: Module Definition for the 2-to-1 7-bit Multiplexer

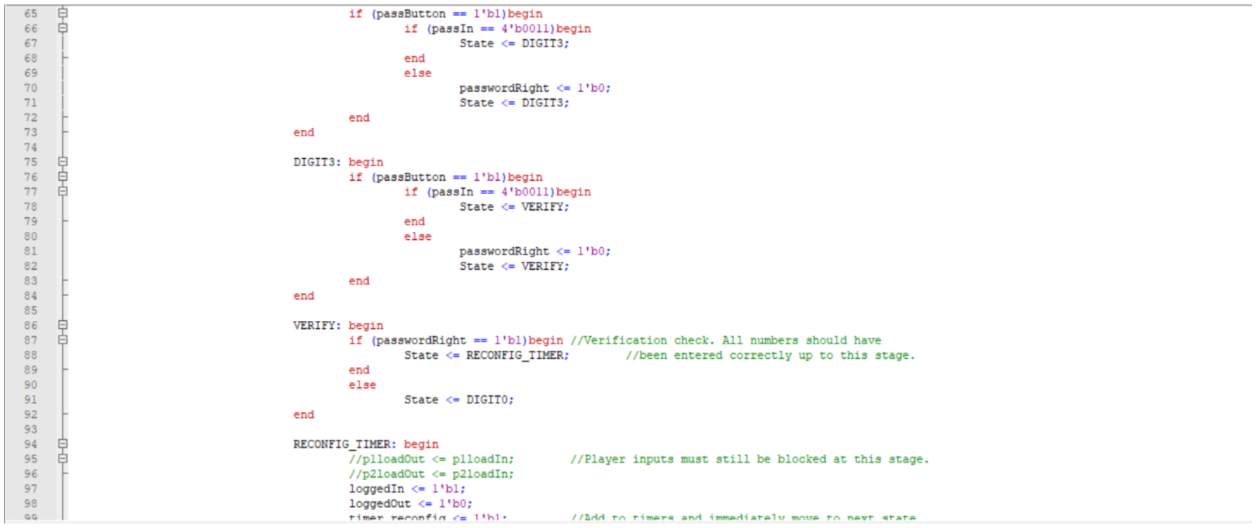
Appendix F: Verilog Code for Modified Access Controller



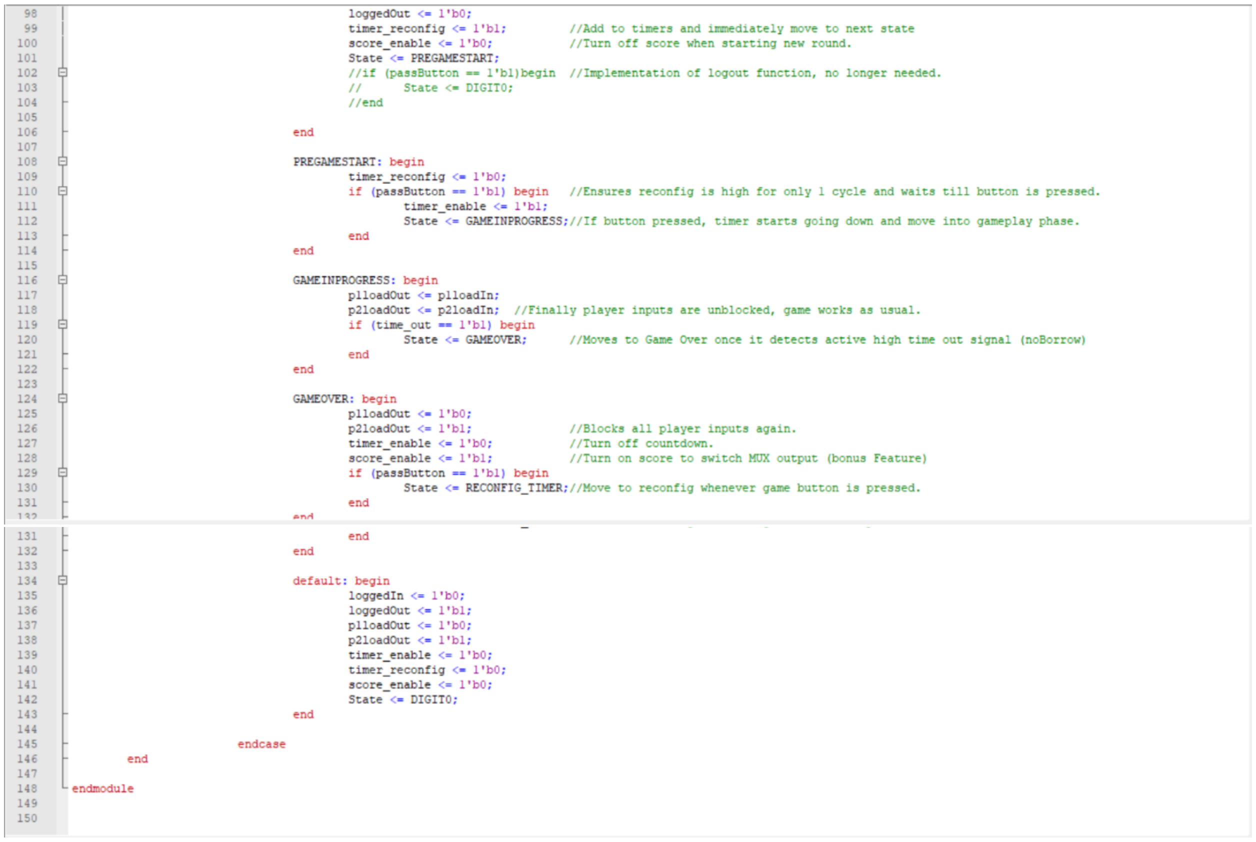
```verilog
//ECE 6370
//Author: David Zhang, 3233
//Access Controller Module, Lab 3

//This module defines the modified access controller for the mental binary game, which replaces the logout feature and button
//with a button to load 99 seconds into the timer and start the subsequent game. After the timer reaches 0, all features are
//locked out until the button is pushed again to add 99 seconds, then pushed again to start a new round.

module AccessController(passIn, passButton, loggedIn, loggedOut, p1loadIn, p2loadIn, p1loadOut, p2loadOut, rst, clk, timer_reconfig, timer_enable, time_out, score_enable)
        input [3:0] passIn;
        input passButton, p1loadIn, p2loadIn, rst, clk;
        input time_out;
        output timer_reconfig, timer_enable, score_enable;
        output loggedIn, loggedOut, p1loadOut, p2loadOut;
        reg loggedIn, loggedOut, 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.
        reg passwordRight;
        reg[3:0] State;

        always@(posedge clk) begin
                if (rst == 1'b0) begin                                  //Reset of all signals.
                        State <= DIGIT0;
                        passwordRight <= 1'b1;
                        p1loadOut <= 1'b0;
                        p2loadOut <= 1'b1;                                      //Default is high for a pushbutton.
                        loggedIn <= 1'b0;
                        loggedOut <=1'b1;
                        timer_enable <= 1'b0;
                        timer_reconfig <= 1'b0;
                        score_enable <= 1'b0;
                        end
                else
```



```verilog
                        end
                else
                        case(State)
                                DIGIT0: begin                           //Setting all signals to correct baseline
                                        p1loadOut <= 1'b0;
                                        p2loadOut <= 1'b1;
                                        passwordRight <= 1'b1;
                                        loggedIn <= 1'b0;
                                        loggedOut <=1'b1;
                                        if (passButton == 1'b1)begin            //Operation should ONLY proceed whenever
                                                if (passIn == 4'b0011)begin     //the pushbutton is depressed.
                                                        State <= DIGIT1;
                                                end
                                                else
                                                        passwordRight <= 1'b0;
                                                        State <= DIGIT1;
                                        end

                                end

                                DIGIT1: begin
                                        if (passButton == 1'b1)begin
                                                if (passIn == 4'b0010)begin
                                                        State <= DIGIT2;
                                                end
                                                else
                                                        passwordRight <= 1'b0;
                                                        State <= DIGIT2;
                                        end
                                end

                                DIGIT2: begin
                                        if (passButton == 1'b1)begin
                                                if (passIn == 4'b0011)begin
```



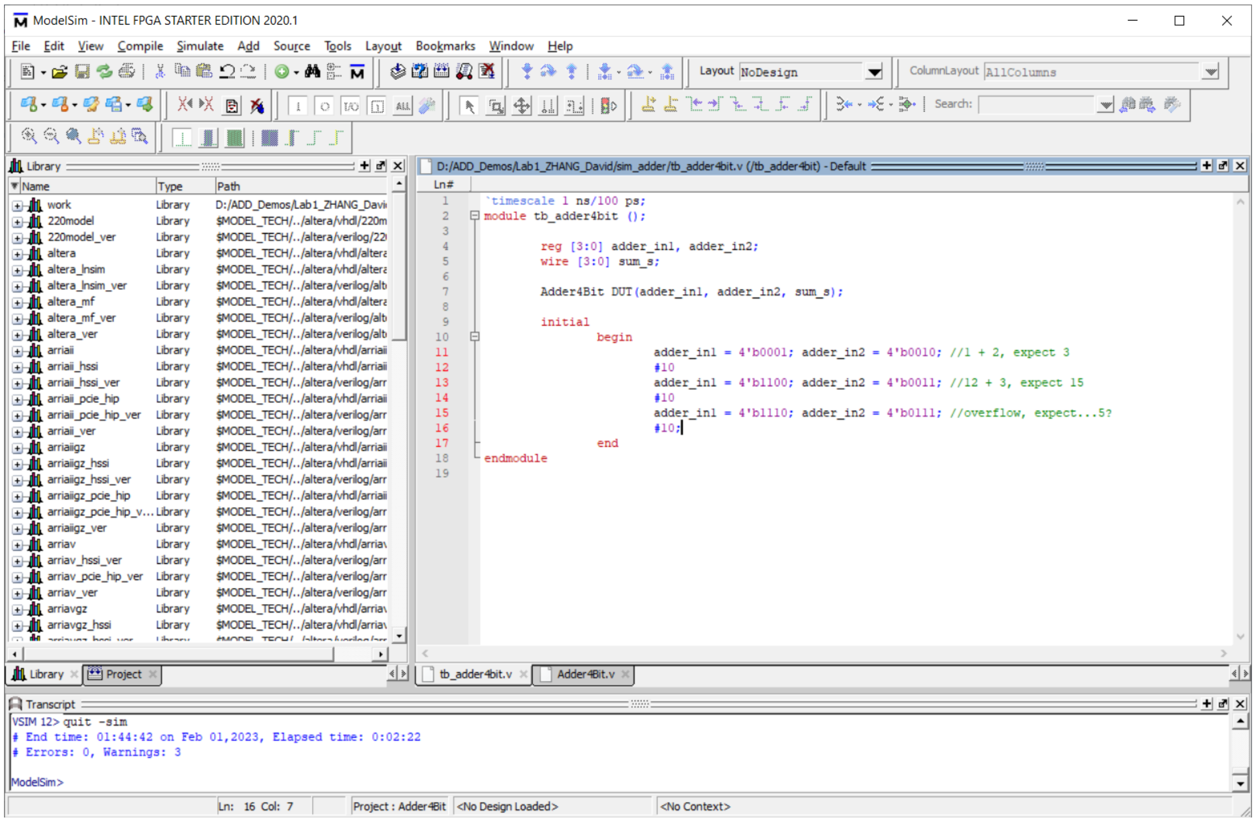
```verilog
                                        if (passButton == 1'b1)begin
                                                if (passIn == 4'b0011)begin
                                                        State <= DIGIT3;
                                                end
                                                else
                                                        passwordRight <= 1'b0;
                                                        State <= DIGIT3;
                                        end
                                end

                                DIGIT3: begin
                                        if (passButton == 1'b1)begin
                                                if (passIn == 4'b0011)begin
                                                        State <= VERIFY;
                                                end
                                                else
                                                        passwordRight <= 1'b0;
                                                        State <= VERIFY;
                                        end
                                end

                                VERIFY: begin
                                        if (passwordRight == 1'b1)begin //Verification check. All numbers should have
                                                State <= RECONFIG_TIMER;        //been entered correctly up to this stage.
                                        end
                                        else
                                                State <= DIGIT0;
                                end

                                RECONFIG_TIMER: begin
                                        //p1loadOut <= p1loadIn;        //Player inputs must still be blocked at this stage.
                                        //p2loadOut <= p2loadIn;
                                        loggedIn <= 1'b1;
                                        loggedOut <= 1'b0;
```



```
                    loggedOut <= 1'b0;
                    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
                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 <= RECONFIG_TIMER;//Move to reconfig whenever game button is pressed.
                    end
                end

                default: begin
                    loggedIn <= 1'b0;
                    loggedOut <= 1'b1;
                    p1loadOut <= 1'b0;
                    p2loadOut <= 1'b1;
                    timer_enable <= 1'b0;
                    timer_reconfig <= 1'b0;
                    score_enable <= 1'b0;
                    State <= DIGIT0;
                end

            endcase
    end

endmodule
```

Figure 24: Module Definition for Modified Access Controller

Appendix G: Legacy Verilog Code



ModelSim - INTEL FPGA STARTER EDITION 2020.1

Name	Type	Path
work	Library	D:/ADD_Demos/Lab1_ZHANG_Davi
220model	Library	$MODEL_TECH/../altera/vhdl/220m
220model_ver	Library	$MODEL_TECH/../altera/verilog/22
altera	Library	$MODEL_TECH/../altera/vhdl/altera
altera_lnsim	Library	$MODEL_TECH/../altera/vhdl/altera
altera_lnsim_ver	Library	$MODEL_TECH/../altera/verilog/alt
altera_mf	Library	$MODEL_TECH/../altera/vhdl/altera
altera_mf_ver	Library	$MODEL_TECH/../altera/verilog/alt
altera_ver	Library	$MODEL_TECH/../altera/verilog/alt
arriaii	Library	$MODEL_TECH/../altera/vhdl/arriaii
arriaii_hssi	Library	$MODEL_TECH/../altera/vhdl/arriaii
arriaii_hssi_ver	Library	$MODEL_TECH/../altera/verilog/arr
arriaii_pcie_hip	Library	$MODEL_TECH/../altera/vhdl/arriaii
arriaii_pcie_hip_ver	Library	$MODEL_TECH/../altera/verilog/arr
arriaii_ver	Library	$MODEL_TECH/../altera/verilog/arr
arriaiigz	Library	$MODEL_TECH/../altera/vhdl/arriaii
arriaiigz_hssi	Library	$MODEL_TECH/../altera/vhdl/arriaii
arriaiigz_hssi_ver	Library	$MODEL_TECH/../altera/verilog/arr
arriaiigz_pcie_hip	Library	$MODEL_TECH/../altera/vhdl/arriaii
arriaiigz_pcie_hip_v...	Library	$MODEL_TECH/../altera/verilog/arr
arriaiigz_ver	Library	$MODEL_TECH/../altera/verilog/arr
arriav	Library	$MODEL_TECH/../altera/vhdl/arriav
arriav_hssi_ver	Library	$MODEL_TECH/../altera/verilog/arr
arriav_pcie_hip_ver	Library	$MODEL_TECH/../altera/verilog/arr
arriav_ver	Library	$MODEL_TECH/../altera/verilog/arr
arriavgz	Library	$MODEL_TECH/../altera/vhdl/arriav
arriavgz_hssi	Library	$MODEL_TECH/../altera/vhdl/arriav

D:/ADD_Demos/Lab1_ZHANG_David/sim_adder/tb_adder4bit.v (/tb_adder4bit) - Default

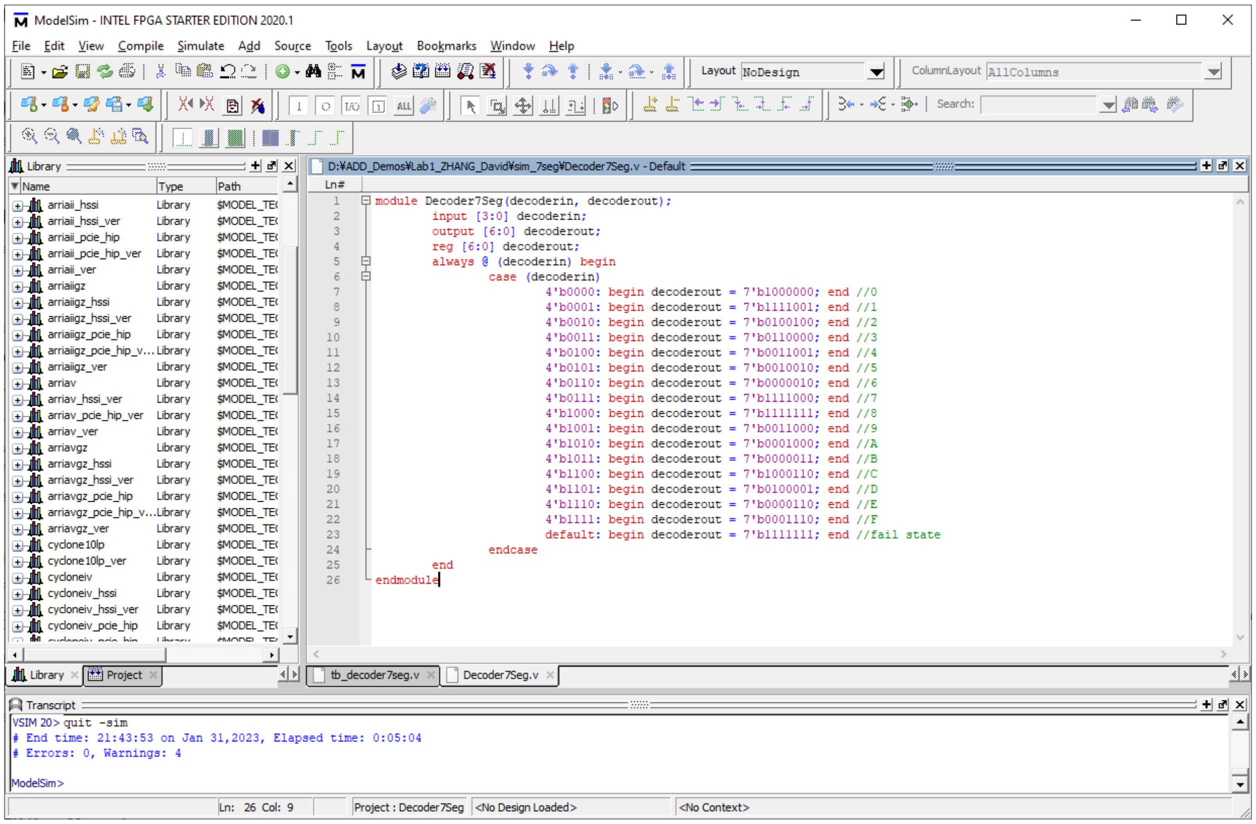
```verilog
`timescale 1 ns/100 ps;
module tb_adder4bit ();

        reg [3:0] adder_in1, adder_in2;
        wire [3:0] sum_s;

        Adder4Bit DUT(adder_in1, adder_in2, sum_s);

        initial
                begin
                        adder_in1 = 4'b0001; adder_in2 = 4'b0010; //1 + 2, expect 3
                        #10
                        adder_in1 = 4'b1100; adder_in2 = 4'b0011; //12 + 3, expect 15
                        #10
                        adder_in1 = 4'b1110; adder_in2 = 4'b0111; //overflow, expect...5?
                        #10;
                end
endmodule
```

Transcript

```
VSIM 12> quit -sim
# End time: 01:44:42 on Feb 01,2023, Elapsed time: 0:02:22
# Errors: 0, Warnings: 3

ModelSim>
```

Ln: 16 Col: 7 Project : Adder4Bit <No Design Loaded> <No Context>

Figure 25: Module Definition for Adder


ModelSim - INTEL FPGA STARTER EDITION 2020.1
File Edit View Compile Simulate Add Source Tools Layout Bookmarks Window Help
Layout NoDesign
ColumnLayout AllColumns
Search:
Library
Project
tb_decoder7seg.v
Decoder7Seg.v
Transcript

Name	Type	Path
arriaii_hssi	Library	$MODEL_TE(
arriaii_hssi_ver	Library	$MODEL_TE(
arriaii_pcie_hip	Library	$MODEL_TE(
arriaii_pcie_hip_ver	Library	$MODEL_TE(
arriaii_ver	Library	$MODEL_TE(
arriaiigz	Library	$MODEL_TE(
arriaiigz_hssi	Library	$MODEL_TE(
arriaiigz_hssi_ver	Library	$MODEL_TE(
arriaiigz_pcie_hip	Library	$MODEL_TE(
arriaiigz_pcie_hip_v...	Library	$MODEL_TE(
arriaiigz_ver	Library	$MODEL_TE(
arriav	Library	$MODEL_TE(
arriav_hssi_ver	Library	$MODEL_TE(
arriav_pcie_hip_ver	Library	$MODEL_TE(
arriav_ver	Library	$MODEL_TE(
arriavgz	Library	$MODEL_TE(
arriavgz_hssi	Library	$MODEL_TE(
arriavgz_hssi_ver	Library	$MODEL_TE(
arriavgz_pcie_hip	Library	$MODEL_TE(
arriavgz_pcie_hip_v...	Library	$MODEL_TE(
arriavgz_ver	Library	$MODEL_TE(
cyclone10lp	Library	$MODEL_TE(
cyclone10lp_ver	Library	$MODEL_TE(
cycloneiv	Library	$MODEL_TE(
cycloneiv_hssi	Library	$MODEL_TE(
cycloneiv_hssi_ver	Library	$MODEL_TE(
cycloneiv_pcie_hip	Library	$MODEL_TE(

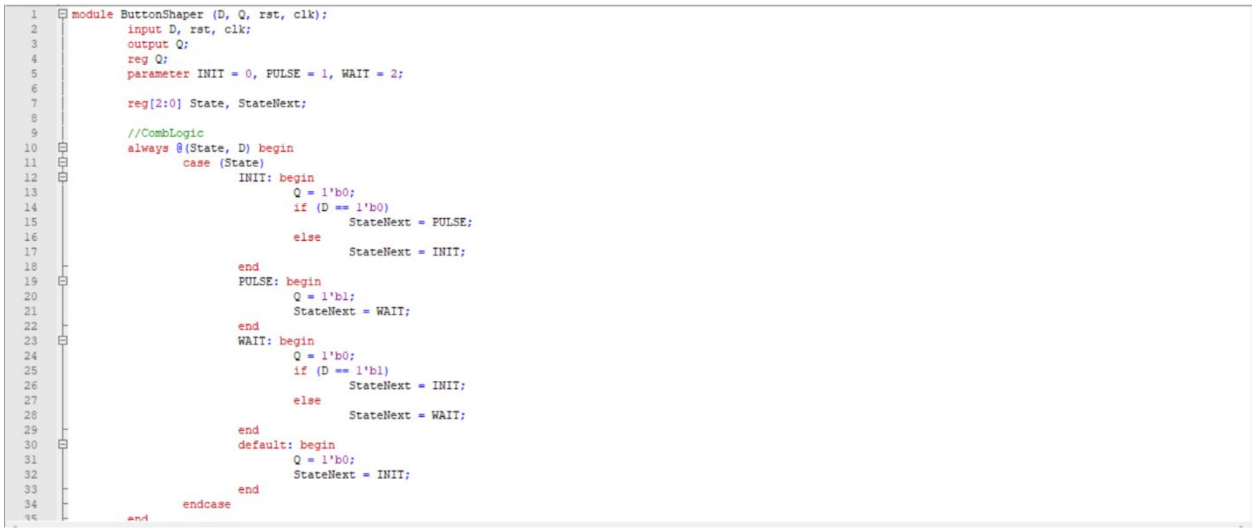
D:¥ADD_Demos¥Lab1_ZHANG_David¥sim_7seg¥Decoder7Seg.v - Default

```verilog
module Decoder7Seg(decoderin, decoderout);
        input [3:0] decoderin;
        output [6:0] decoderout;
        reg [6:0] decoderout;
        always @ (decoderin) begin
                case (decoderin)
                        4'b0000: begin decoderout = 7'b1000000; end //0
                        4'b0001: begin decoderout = 7'b1111001; end //1
                        4'b0010: begin decoderout = 7'b0100100; end //2
                        4'b0011: begin decoderout = 7'b0110000; end //3
                        4'b0100: begin decoderout = 7'b0011001; end //4
                        4'b0101: begin decoderout = 7'b0010010; end //5
                        4'b0110: begin decoderout = 7'b0000010; end //6
                        4'b0111: begin decoderout = 7'b1111000; end //7
                        4'b1000: begin decoderout = 7'b1111111; end //8
                        4'b1001: begin decoderout = 7'b0011000; end //9
                        4'b1010: begin decoderout = 7'b0001000; end //A
                        4'b1011: begin decoderout = 7'b0000011; end //B
                        4'b1100: begin decoderout = 7'b1000110; end //C
                        4'b1101: begin decoderout = 7'b0100001; end //D
                        4'b1110: begin decoderout = 7'b0000110; end //E
                        4'b1111: begin decoderout = 7'b0001110; end //F
                        default: begin decoderout = 7'b1111111; end //fail state
                endcase
        end
endmodule
```

```
VSIM 20> quit -sim
# End time: 21:43:53 on Jan 31,2023, Elapsed time: 0:05:04
# Errors: 0, Warnings: 4

ModelSim>
```

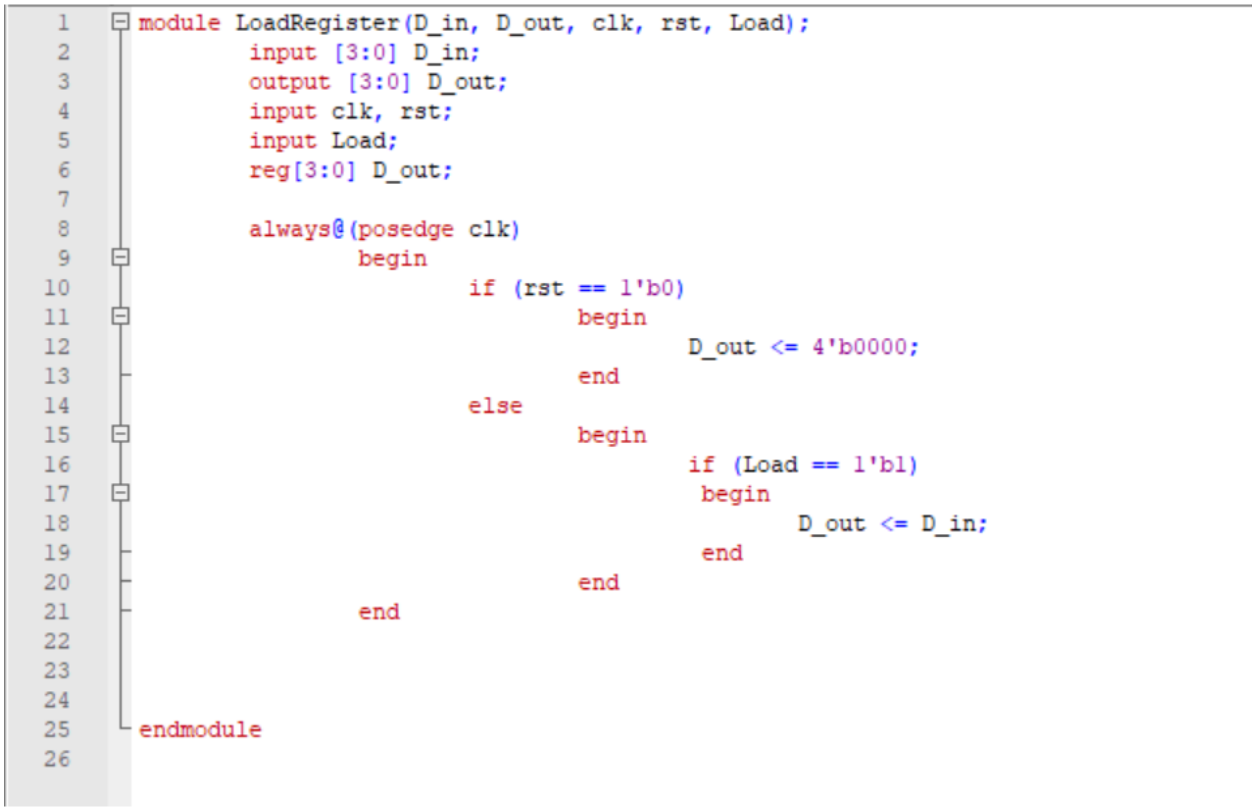
Ln: 26 Col: 9 Project : Decoder7Seg <No Design Loaded> <No Context>

Figure 26: Module Definition for 7-Segment Decoder



```verilog
module ButtonShaper (D, Q, rst, clk);
        input D, rst, clk;
        output Q;
        reg Q;
        parameter INIT = 0, PULSE = 1, WAIT = 2;

        reg[2:0] State, StateNext;

        //CombLogic
        always @(State, D) begin
                case (State)
                        INIT: begin
                                Q = 1'b0;
                                if (D == 1'b0)
                                        StateNext = PULSE;
                                else
                                        StateNext = INIT;
                        end
                        PULSE: begin
                                Q = 1'b1;
                                StateNext = WAIT;
                        end
                        WAIT: begin
                                Q = 1'b0;
                                if (D == 1'b1)
                                        StateNext = INIT;
                                else
                                        StateNext = WAIT;
                        end
                        default: begin
                                Q = 1'b0;
                                StateNext = INIT;
                        end
                endcase
        end
```

Figure 27: Module Definition for Button Shaper



```
module LoadRegister(D_in, D_out, clk, rst, Load);
        input [3:0] D_in;
        output [3:0] D_out;
        input clk, rst;
        input Load;
        reg[3:0] D_out;

        always@(posedge clk)
                begin
                        if (rst == 1'b0)
                                begin
                                        D_out <= 4'b0000;
                                end
                        else
                                begin
                                        if (Load == 1'b1)
                                         begin
                                                D_out <= D_in;
                                         end
                                end
                end



endmodule
```

Figure 28: Module Definition for Load Register