Lab 3: Hardware-based True Random Number Generator and Timer Control on FPGA

Thomas Vo
2179861

ECE 5440

Introduction

This game is a binary arithmetic game where a player toggles switches to reach a target sum of 1111 (binary 15). A hardware-based Random Number Generator (RNG) determines the second number.

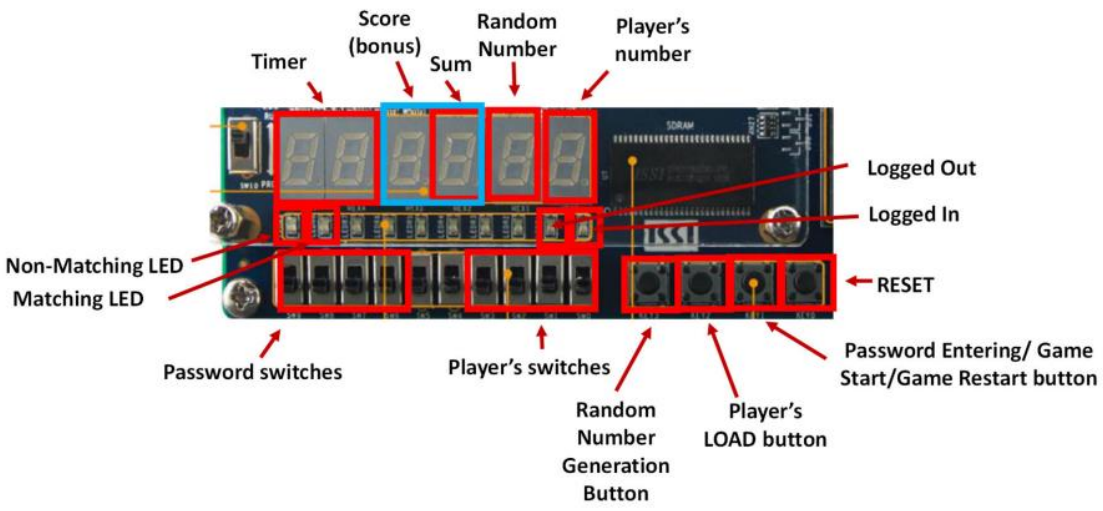


Figure 1. Diagram for Controls and Displays

As shown in Figure 1, the 7-segment displays show the number entered by the player, the randomly generated number, and their sum. LED indicators show whether the entered numbers match 1111, as well as the player's login status. The 4 right-most switches are for the player's input, while a push button is used to generate a random number. The 4 left-most switches are used to enter a password at the start of the game.
The game begins with the password entry phase. If the correct password is entered, the system unlocks, and the player can start the game. If the password is incorrect, the system remains locked, preventing any inputs.
Once logged in, the player selects a number using the RNG button. The player generates a second number, which determines the sum. If the sum matches 1111, the player earns a point. Otherwise, the round continues with new numbers.

A game timer (default: 99 seconds) controls gameplay, counting down while the player tries to complete as many successful rounds as possible. The timer remains at 00 until the password is verified, after which pressing the GAME button starts the countdown. Once time runs out, no further inputs are accepted. Pressing the GAME button again resets the timer to 99 seconds, allowing for a new session.

At the end of the game, a bonus scoring feature displays the total number of successful rounds on two 7-segment displays (in base-10). The scoring display is only active after the game ends.

If the user wishes to reset the game at any time, they can press the reset button, which clears all progress and returns the system to the login phase.

System Architecture Design

Shown below in Figure 2 is the system architectural design, which shows the modules, input signals, and output signals.

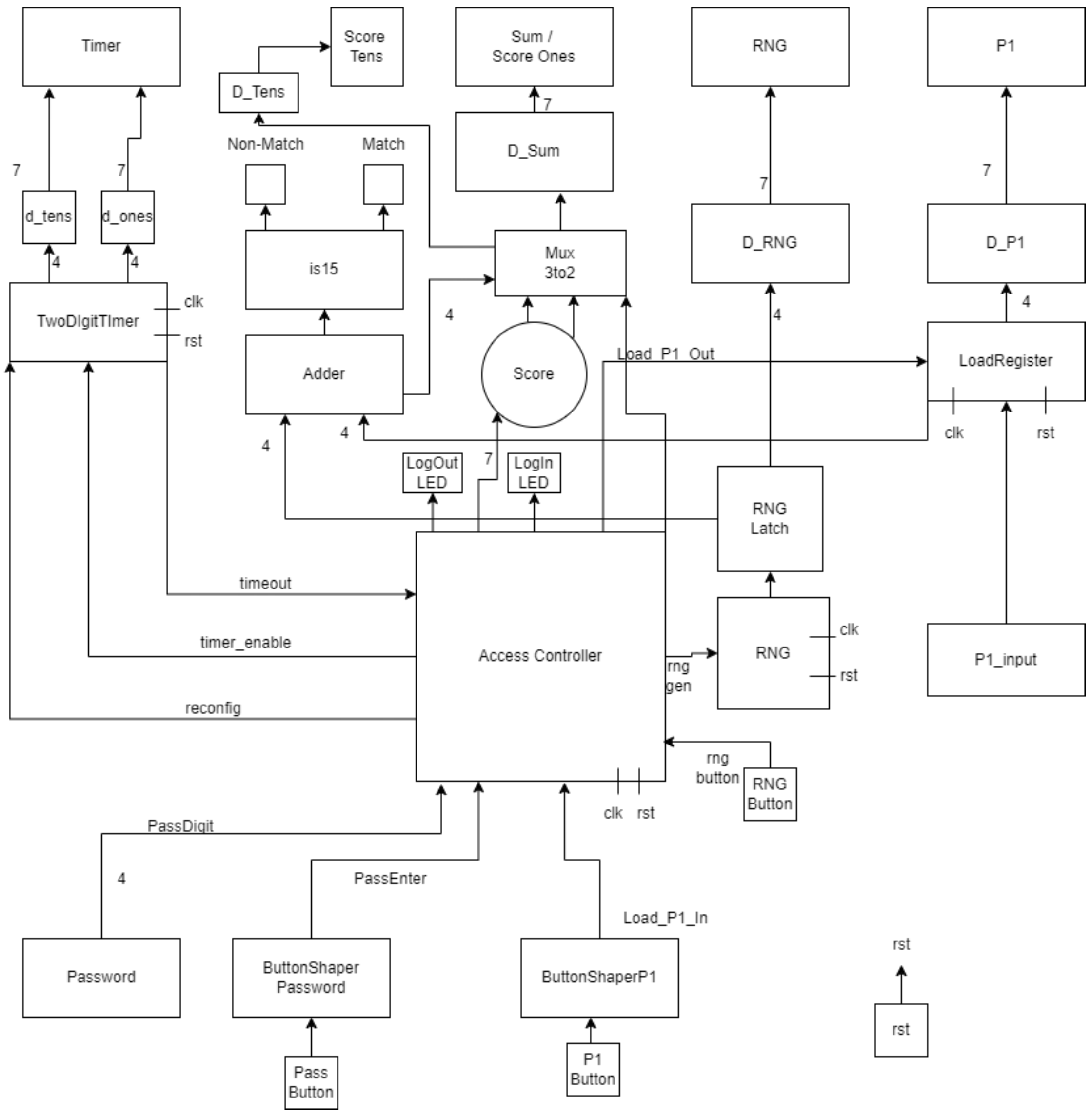


Figure 2. Top-level System Architectural Design

The top-level system module connects the adder, decoder, verification, button shaper, load register, and access controller modules to perform the game's functions. The input signals are player 1 and 2's inputs, which are 4-bit signals, and player 1's, player 2's, password/ log out's button, and reset button, which are 1-bit signals. The 7-bit output signals are the displays of player 1, player 2, the sum. The 1-bit output signals are the matching, non-matching, logged-in, and logged-out LEDs.

Digit Timer

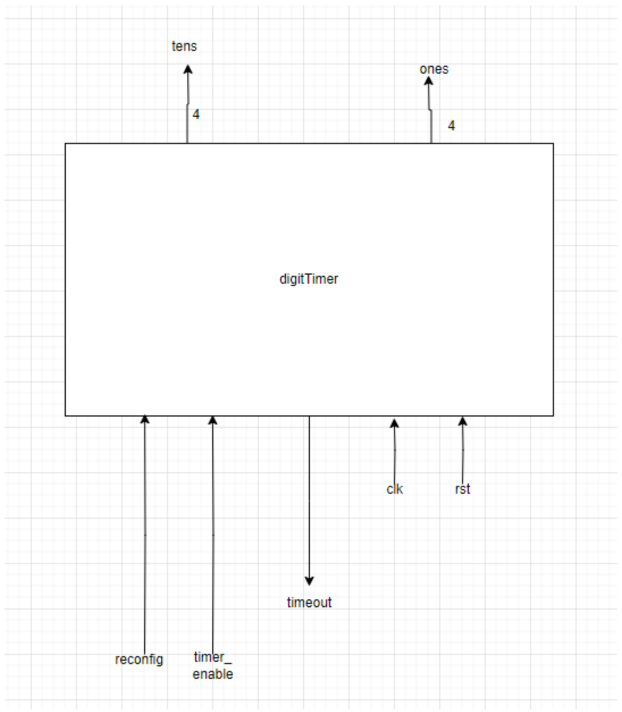


Figure 3. Digit Timer

The digitTimer module implements a two-digit countdown timer using two dt submodules to represent the ones and tens places. It operates based on a one-second timer pulse generated by the oneSecTimer module. The timer starts decrementing when timer_enable is asserted, counting down from 99 to 00. The dt module handles individual digit decrements, borrowing from the next digit when necessary. If the countdown reaches zero, the timeout signal is asserted. The timer can be reset via rst or reconfigured to restart at 99 using reconfig. The input signals include clk, rst, timer_enable, and reconfig, while the output signals are ones, tens, and timeout.

RNG and RNG Latch

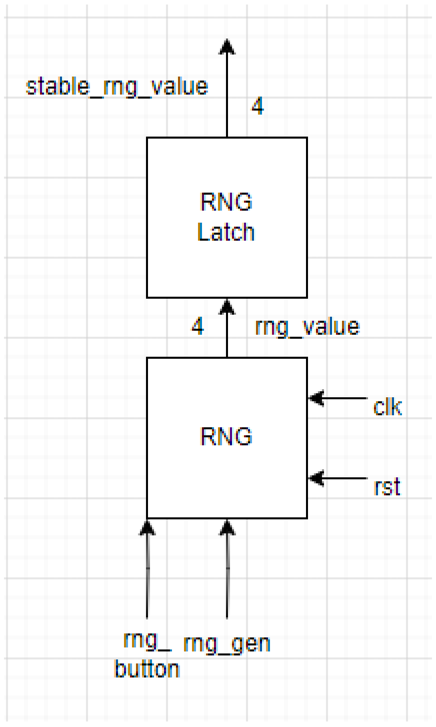


Figure 4. RNG and RNG Latch

The rng module generates a 4-bit random number using a counter-based approach. The random number updates when rng_gen is toggled, with an inverted signal (inv_in) controlling the counter module. The rng_latch module ensures stability by capturing and storing the random number only when the rng_button is released (falling edge detection). This prevents rapid changes and ensures a consistent output. The rst signal resets the stored value to 0000. The input signals for the system include rng_gen, rng_button, clk, rst, and rng_value, while the output signal is stable_rng_value, representing the latched random number.

MUX 3 to 2

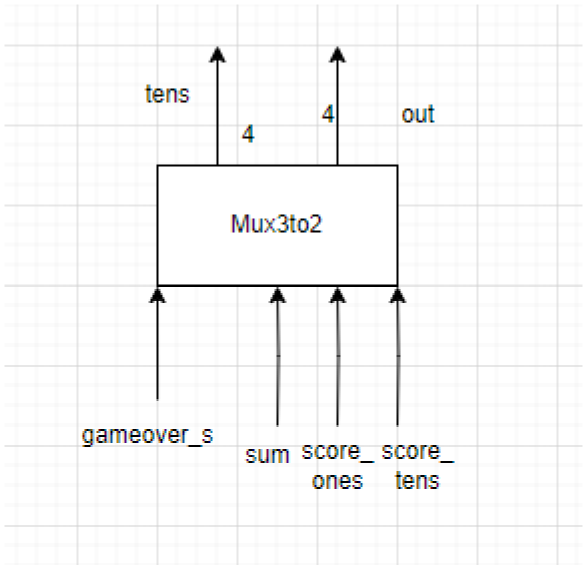


Figure 5. MUX 3 to 2

The mux3to2 module is a multiplexer that selects between two sets of 4-bit values based on the gameover_s signal. If gameover_s is 1 (game over), the module outputs score_ones and score_tens, representing the final score. Otherwise, it outputs sum and sets tens_out to 0. This allows dynamic switching between the current game sum and the final stored score. The module takes sum, score_ones, score_tens, and gameover_s as inputs and produces out and tens_out as outputs.

Score

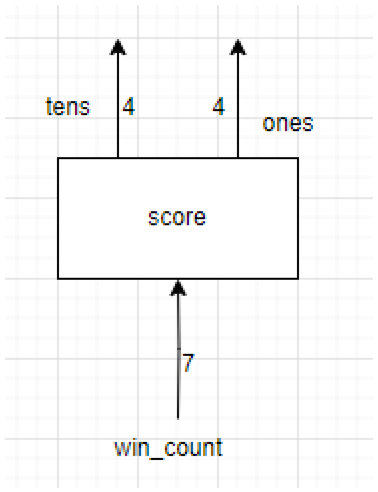


Figure 6. Score

The score module extracts the ones and tens digits from a 7-bit win count (ranging from 0 to 99). It computes the tens digit by performing integer division by 10 and the ones digit using the modulus operation. This allows the win_count value to be split into two separate 4-bit outputs, making it easier to display or process. The module takes win_count as an input and outputs ones and tens.

One Second Timer

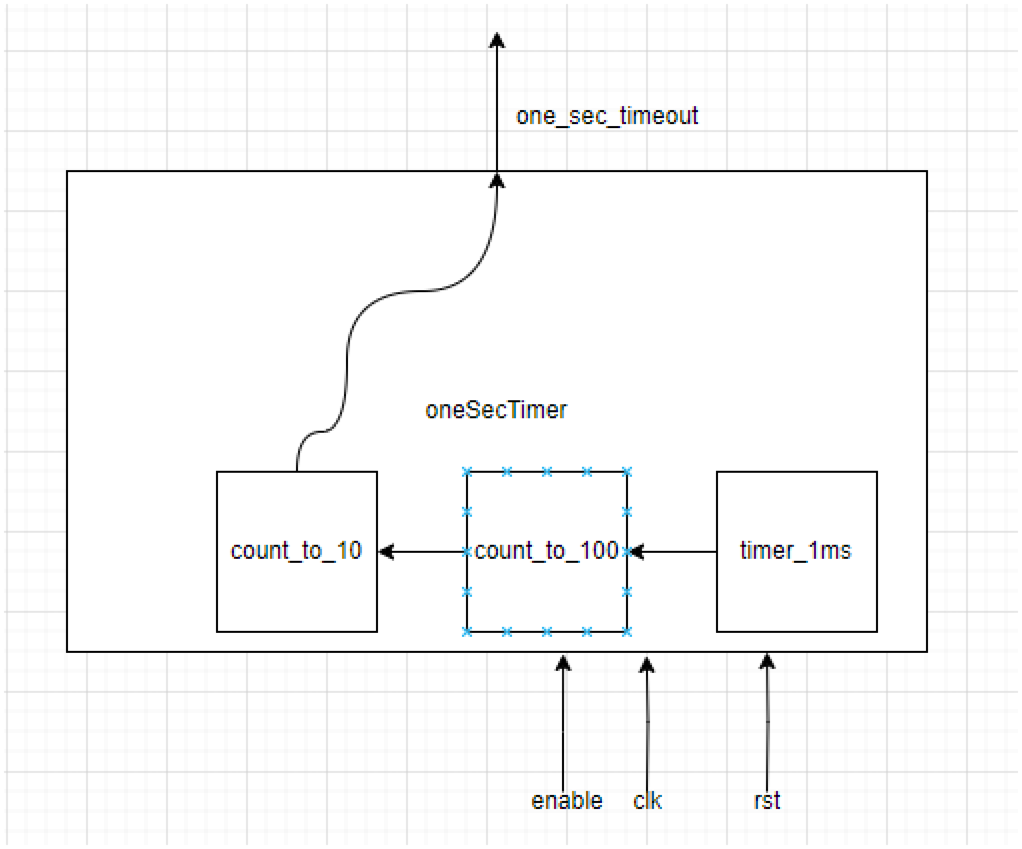


Figure 7. One Second Timer

The oneSecTimer module generates a one-second timeout pulse by cascading three counters: timer_1ms, count_to_100, and count_to_10. The timer_1ms module produces a 1 ms pulse by counting 50,000 clock cycles (assuming a 50 MHz clock). This pulse feeds into the count_to_100 module, which counts 100 pulses to generate a 100 ms timeout signal. Finally, the count_to_10 module accumulates ten 100 ms pulses to produce a 1-second timeout pulse. The module resets when rst is asserted and operates when enable is active. The input signals include clk, rst, and enable, while the output signal is one_sec_timeout, indicating the completion of a one-second interval.

Access Controller

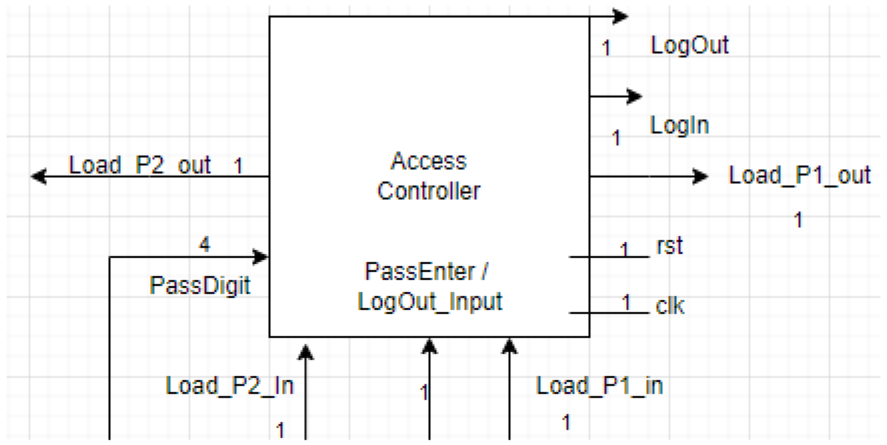


Figure 8. Access Controller

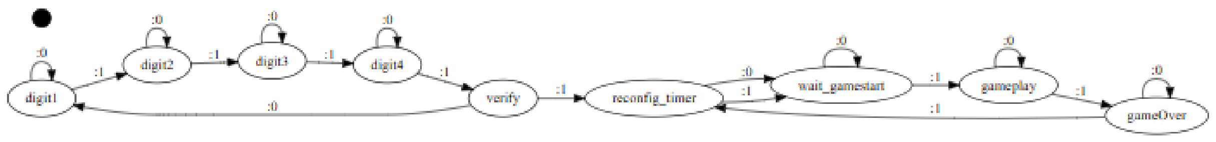


Figure 9. Finite State Machine for Access Controller

The Access module manages user authentication and system control by processing a four-digit passcode and handling various states for gameplay and reconfiguration. It transitions through multiple states to verify the entered passcode, granting access if the correct sequence (9-8-6-1) is entered. Once verified, the module enables system reconfiguration, gameplay activation, and win tracking. It controls various system components, including a random number generator (rng_gen), a timer (timer_enable), and indicators (loginLED, logoutLED, gameover). If an incorrect passcode is entered, the module resets to the initial state. Additionally, it monitors gameplay, detecting timeouts and tracking the number of wins (win_count). The input signals are PassDigit, PassEnter, Load_P1_In, rng_button, timeout, clk, rst, and win, while the output signals include Load_P1_Out, rng_gen, timer_enable, reconfig, logoutLED, loginLED, and gameover.

Button Shaper

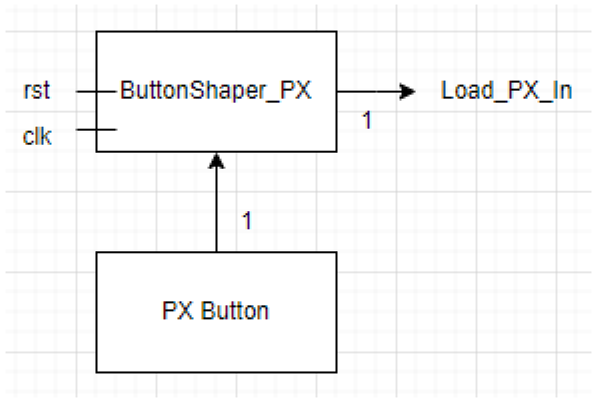


Figure 10. Button Shaper

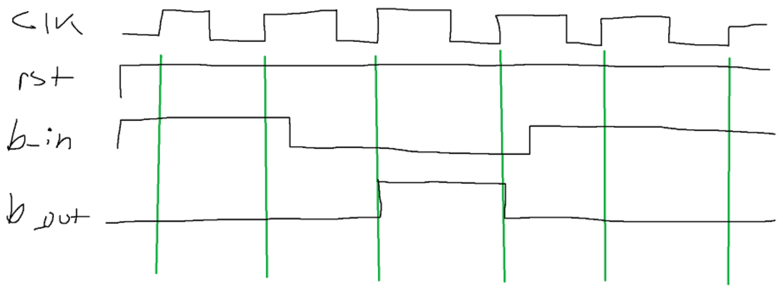


Figure 11. Expected Waveform for Button Shaper

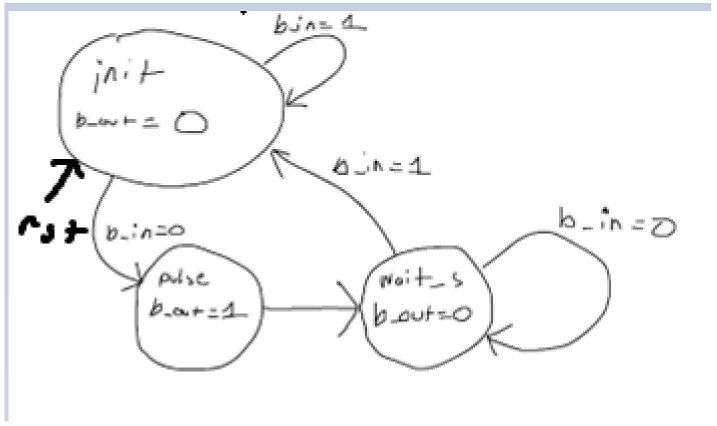


Figure 12. Finite State Machine drawing for Button Shaper

The buttonShaper module converts a button input into a predictable digital signal. It uses a finite state machine with three states: init, pulse, and wait_s. When the button is pressed, the module outputs a clean pulse signal on b_out. The input signals are b_in, clk, and rst, and the output signal is b_out.

Load Register

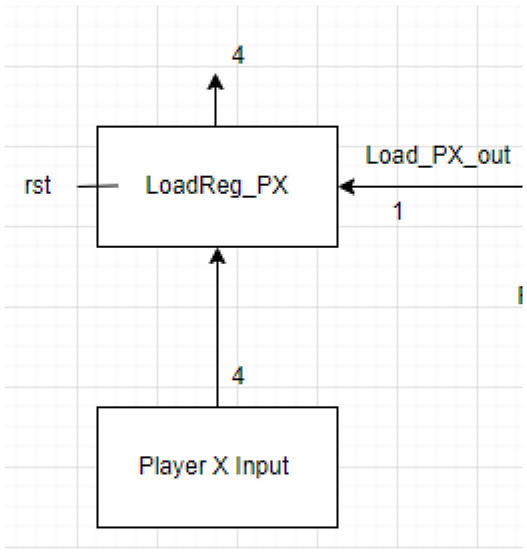


Figure 13. Load Register

The LoadRegister module is a 4-bit register that stores data from the input D_in until a button press. It updates its output D_out on the rising edge of the clock when the Load signal is high. If the reset (rst) is low, the register is cleared to 0000. The input signals are D_in, clk, rst, and Load, and the output signal is D_out.

Adder

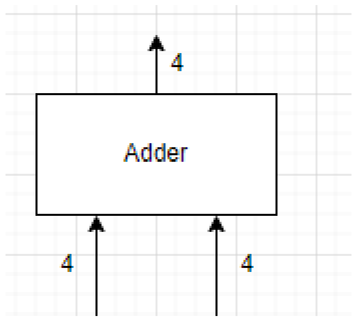


Figure 14. Adder Module

The adder module takes two 4-bit inputs and performs a logical AND, outputting a 4-bit output. The adder module also outputs two 1-bit outputs for the matching and non-matching LEDs. The signal names for the inputs are: “num1” and “num2”, and the output names are: “out”, “led_match_on”, and “led_match_off”, respectively.

Seven Segment Decoder

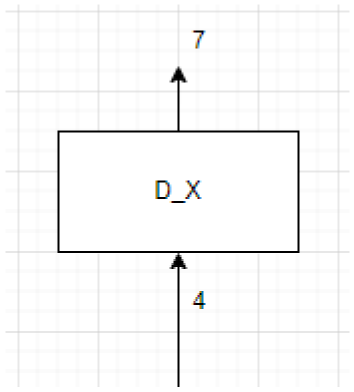


Figure 15. Seven Segment Decoder

The sevenSegDecoder_X module takes a 4-bit input signal and decodes it to be able to be read by the seven-segment display in the form of a 7-bit output signal. The input signal is “decode_in” and the output signal is “decode_out”.

Is15 Verification Module

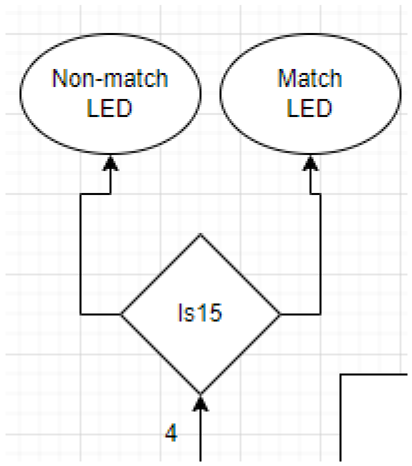


Figure 16. Is15 Verification Module

The Is15 module is a verification module that takes a 4-bit input from the Adder module and determines if it is 15 or not. It outputs two signals , “led_match_on” and “led_match_off”, which are 1-bit signals to turn on/off the LEDs.

Simulation Results

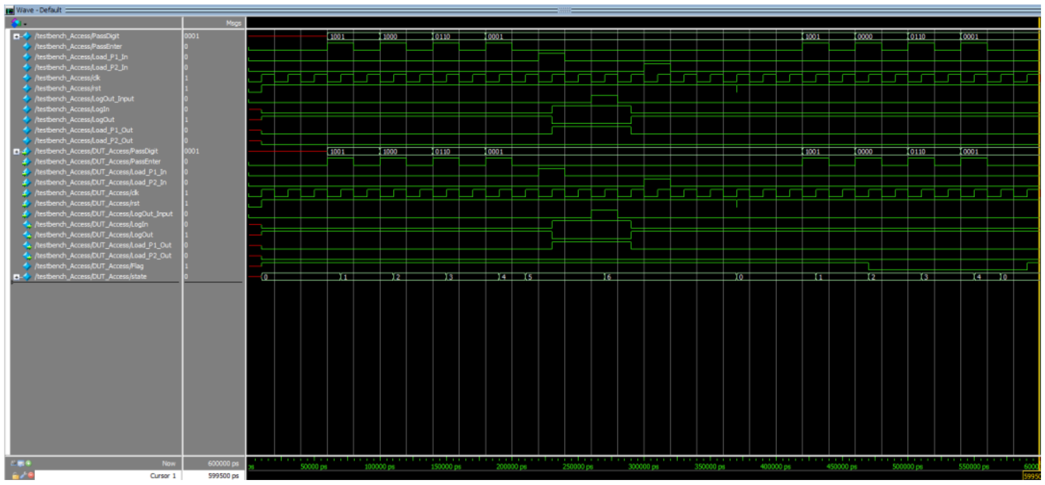

Wave - Default
Msgs
/testbench_Access/PassDigit 0001
/testbench_Access/PassEnter 0
/testbench_Access/Load_P1_In 0
/testbench_Access/Load_P2_In 0
/testbench_Access/clk 1
/testbench_Access/rst 1
/testbench_Access/LogOut_Input 0
/testbench_Access/LogIn 0
/testbench_Access/LogOut 1
/testbench_Access/Load_P1_Out 0
/testbench_Access/Load_P2_Out 0
/testbench_Access/DUT_Access/PassDigit 0001
/testbench_Access/DUT_Access/PassEnter 0
/testbench_Access/DUT_Access/Load_P1_In 0
/testbench_Access/DUT_Access/Load_P2_In 0
/testbench_Access/DUT_Access/clk 1
/testbench_Access/DUT_Access/rst 1
/testbench_Access/DUT_Access/LogOut_Input 0
/testbench_Access/DUT_Access/LogIn 0
/testbench_Access/DUT_Access/LogOut 1
/testbench_Access/DUT_Access/Load_P1_Out 0
/testbench_Access/DUT_Access/Load_P2_Out 0
/testbench_Access/DUT_Access/Flag 1
/testbench_Access/DUT_Access/state 0
1001
1000
0110
0001
Now 600000 ps
Cursor 1 599500 ps

Figure 17. Simulation for Access Controller

The testbench verifies the Access Controller by testing correct and faulty password entries, login, and logout functions. It begins with a reset, then tests a correct 9-8-6-1 password entry, confirming successful login by asserting Load_P1_In. Logging out is simulated with LogOut_Input, followed by Load_P2_In to test another login attempt. After a reset, a second test introduces an incorrect second digit to confirm access denial. The testbench ensures correct password handling, proper logout behavior, and system reset functionality.

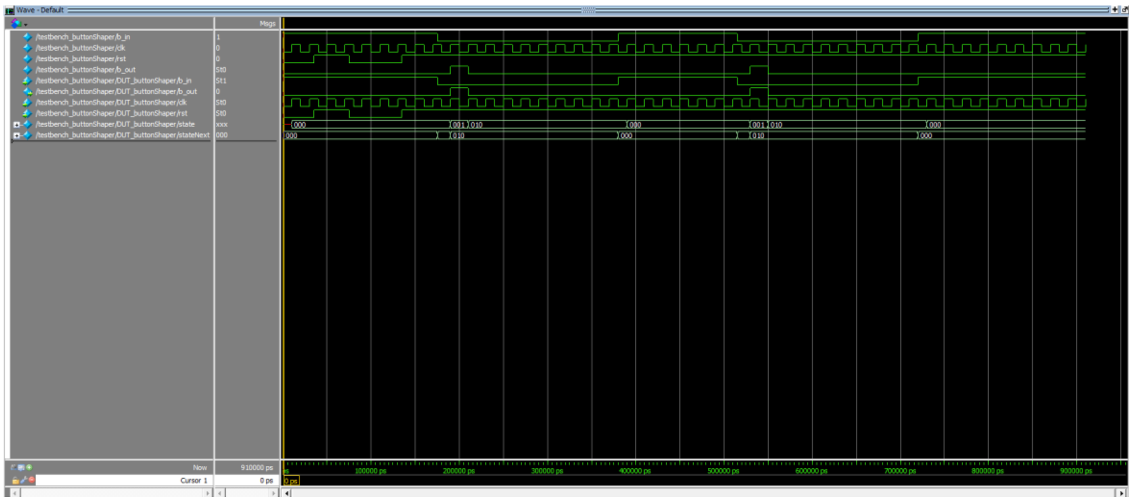
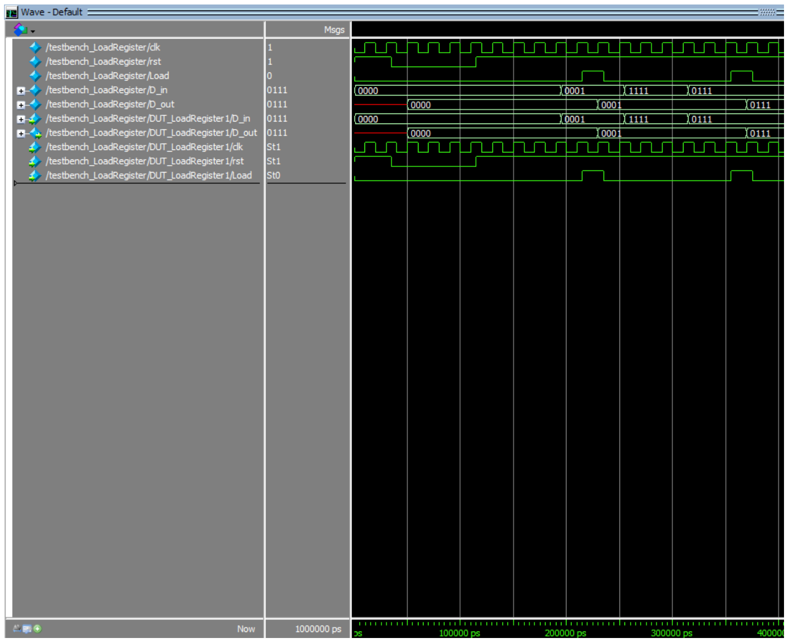

Wave - Default
Msgs
/testbench_buttonShaper/b_in 1
/testbench_buttonShaper/clk 0
/testbench_buttonShaper/rst 0
/testbench_buttonShaper/b_out St0
/testbench_buttonShaper/DUT_buttonShaper/b_in St1
/testbench_buttonShaper/DUT_buttonShaper/b_out 0
/testbench_buttonShaper/DUT_buttonShaper/clk St0
/testbench_buttonShaper/DUT_buttonShaper/rst St0
/testbench_buttonShaper/DUT_buttonShaper/state xxx
/testbench_buttonShaper/DUT_buttonShaper/stateNext 000
Now 910000 ps
Cursor 1 0 ps

Figure 18. Simulation for Button Shaper

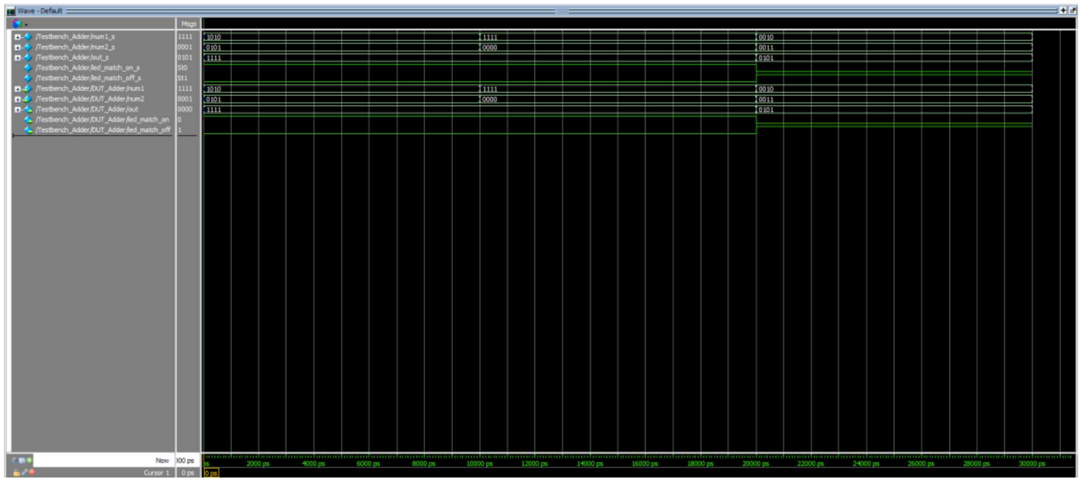
The testbench initializes the ButtonShaper module by setting up the clock, reset, and button input signals. It begins with a reset sequence to ensure a known initial state. The first test simulates a long button press by setting b_in low for multiple clock cycles before releasing it back to high. After a delay, the button is pressed and released again to verify consistent behavior. This test ensures that the ButtonShaper correctly processes and stabilizes button signals, effectively handling debounce effects.



```
Wave - Default
                                                   Msgs
/testbench_LoadRegister/clk                        1
/testbench_LoadRegister/rst                        1
/testbench_LoadRegister/Load                       0
/testbench_LoadRegister/D_in                       0111
/testbench_LoadRegister/D_out                      0111
/testbench_LoadRegister/DUT_LoadRegister1/D_in     0111
/testbench_LoadRegister/DUT_LoadRegister1/D_out    0111
/testbench_LoadRegister/DUT_LoadRegister1/clk      St1
/testbench_LoadRegister/DUT_LoadRegister1/rst      St1
/testbench_LoadRegister/DUT_LoadRegister1/Load     St0
Now                                                1000000 ps
```

Figure 19. Load Register Simulation

The testbench initializes the LoadRegister module, generating a clock signal and setting up the reset and load controls. It begins by asserting a reset to ensure a known initial state before releasing it. The first test sets D_in to 0001 and enables the load signal to store the value in the register, then disables load to hold the value. Next, D_in is updated to 1111 without loading to confirm that the register retains its previous value. Finally, D_in is set to 0111, and after enabling load, the new value is stored in the register. This verifies that the register correctly loads and holds data as expected.



```
Wave - Default
                                           Msgs
/Testbench_Adder/num1_s                    1111
/Testbench_Adder/num2_s                    0001
/Testbench_Adder/out_s                     0101
/Testbench_Adder/led_match_on_s            St0
/Testbench_Adder/led_match_off_s           St1
/Testbench_Adder/DUT_Adder/num1            1111
/Testbench_Adder/DUT_Adder/num2            0001
/Testbench_Adder/DUT_Adder/out             0000
/Testbench_Adder/DUT_Adder/led_match_on    0
/Testbench_Adder/DUT_Adder/led_match_off   1
```

Figure 20. Adder Simulation

The testbench tested two matching cases, 1 non-matching case, and 1 overflow case (non-matching). I tested 10+5 & 15 + 0, these both resulted in 1111 (15). I also tested 2 + 3, which resulted in 0101 (5). Lastly, I tested 15 + 1, which resulted in 0000 (0).
In cases where there was a match (1111), the matching LED signal was set to ON and the non-matching

LED was set to OFF. When there was not a match, the matching LED signal was set to OFF and the non-matching LED was set to ON.These results were expected and correct.

For Figure 4 below, the simulation for the sevenSegDecoder is shown.

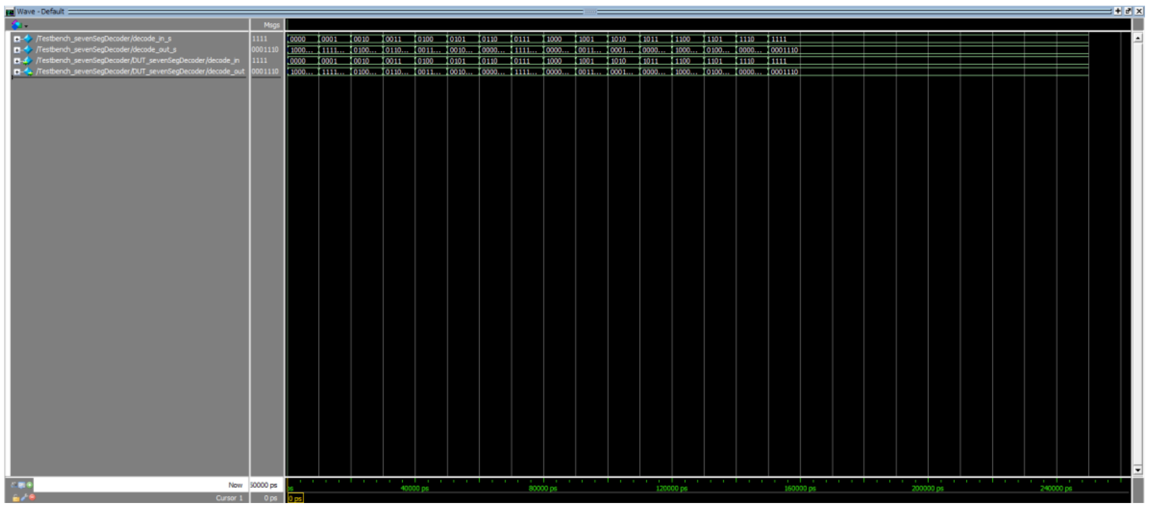

Wave - Default
Msgs
/Testbench_sevenSegDecoder/decode_in_s
/Testbench_sevenSegDecoder/decode_out_s
/Testbench_sevenSegDecoder/DUT_sevenSegDecoder/decode_in
/Testbench_sevenSegDecoder/DUT_sevenSegDecoder/decode_out
Now
Cursor 1

Figure 21. Simulation for sevenSegDecoder

The testbench was simple and tested for each input and output its corresponding 7-segment signal. All outputs were expected and correct.

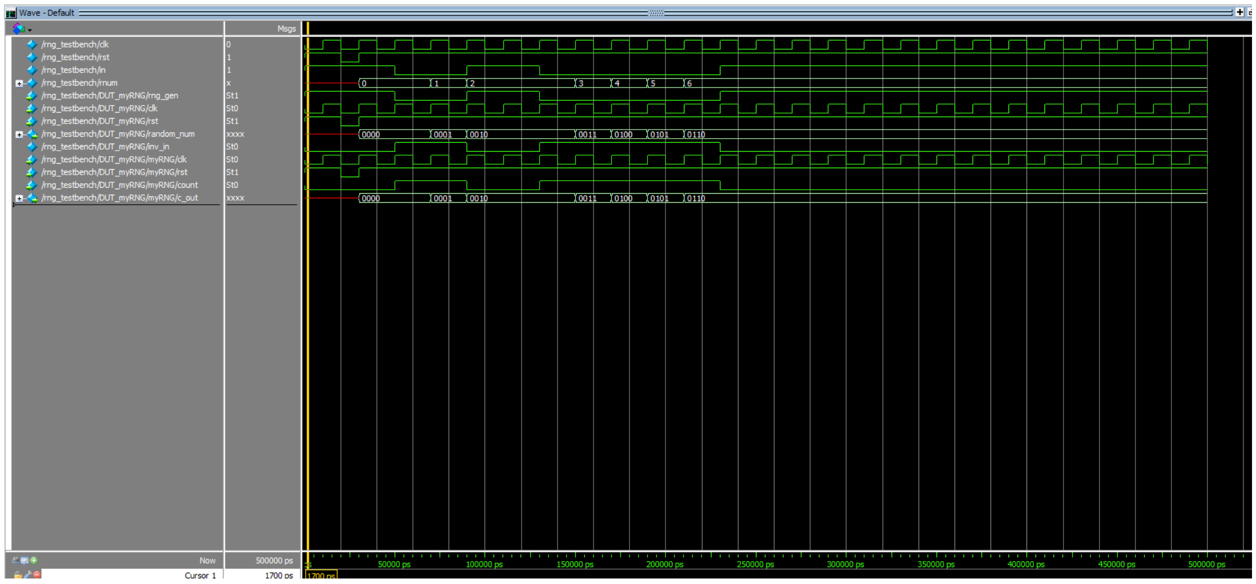

Wave - Default
Msgs
/rng_testbench/clk
/rng_testbench/rst
/rng_testbench/in
/rng_testbench/rnum
/rng_testbench/DUT_myRNG/rng_gen
/rng_testbench/DUT_myRNG/clk
/rng_testbench/DUT_myRNG/rst
/rng_testbench/DUT_myRNG/random_num
/rng_testbench/DUT_myRNG/inv_in
/rng_testbench/DUT_myRNG/myRNG/clk
/rng_testbench/DUT_myRNG/myRNG/rst
/rng_testbench/DUT_myRNG/myRNG/count
/rng_testbench/DUT_myRNG/myRNG/c_out
Now
Cursor 1

Figure 22. Simulation for RNG

The testbench initializes the rng module, generating a clock signal and controlling the reset and in signals. It starts by asserting a reset to initialize the counter. The first test sets in = 1, preventing the counter from incrementing. Then, in = 0 allows the counter to increment for two clock cycles, after which in is set back to 1, stopping the counter. Finally, in is set to 0 again, and the counter resumes incrementing. This verifies the counter's behavior based on the in signal and the reset functionality.

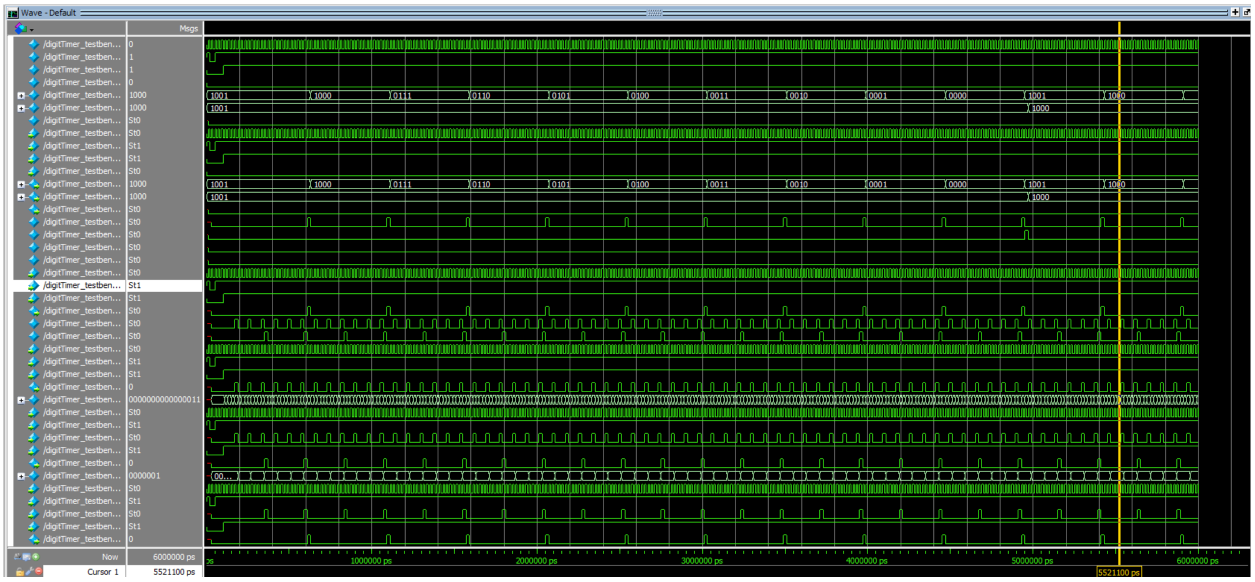


Figure 23. Simulation for Digit Timer

The testbench initializes the digitTimer module by generating a clock signal and controlling the reset, timer enable, and reconfig signals. Initially, the reset (rst = 1) is asserted, ensuring a known state. After 20 ns, the reset is deasserted (rst = 0) to allow the timer to function. Then, the reset is asserted again for 30 ns before being deasserted once more. After 50 ns, the timer_enable signal is set to 1, activating the timer. The testbench runs for 5000 ns to observe the behavior of the timer and the output signals (ones, tens, and timeout). The ones and tens digits should decrement over time, and the timeout signal will be set based on the conditions defined in the digitTimer and dt modules.

This simulation verifies the operation of the digitTimer module, including proper digit counting, timeout behavior, and response to reset and timer enable signals.

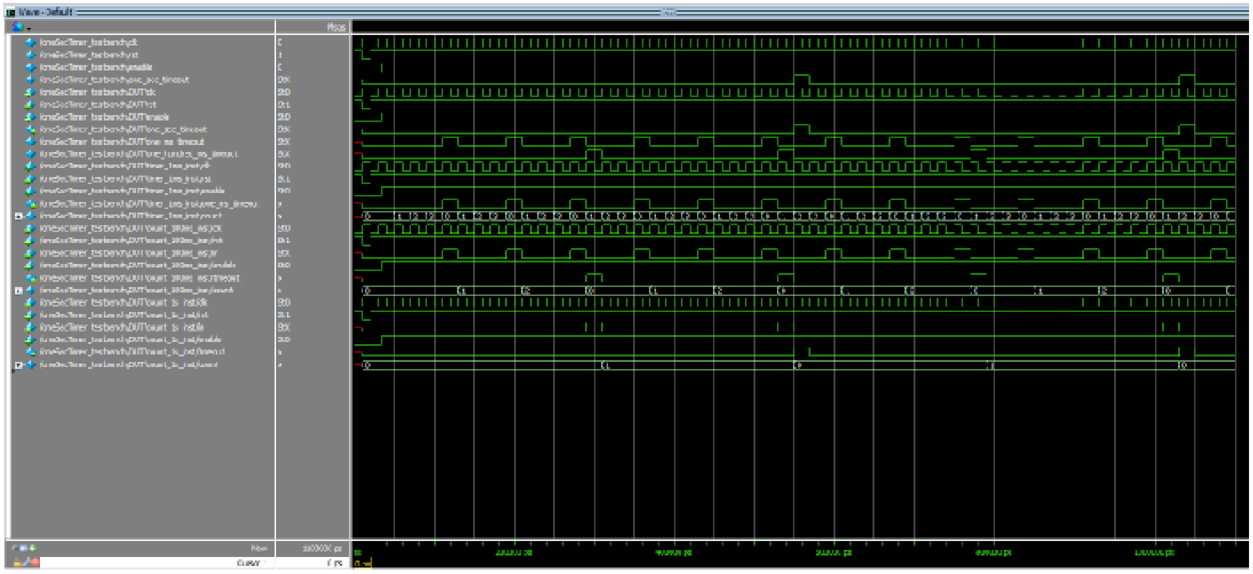


Figure 24. 1 Second Timer Simulation

The testbench initializes the oneSecTimer module, generating a clock signal and controlling the reset (rst) and enable signals. Initially, the reset (rst = 1) is asserted to ensure a known state. After 20 ns, the reset is deasserted (rst = 0), and after another 20 ns, it is asserted again. Then, a 10 ns delay is followed by enabling the timer (enable = 1) for 24000 ns, allowing the timer to run and generate the one_sec_timeout signal.

The oneSecTimer module generates a 1-second timeout using a combination of smaller timers:

1. timer_1ms generates a 1-millisecond timeout using a counter that counts up to 3 (instead of 49999 in the original version) for easier simulation.
2. count_to_100 counts 100 milliseconds, now using a counter that counts up to 3 (instead of 99) to simplify the simulation.
3. count_to_10 counts 1 second, using a counter that counts up to 3 (instead of 9) to make the simulation easier to observe.

The testbench verifies that the one_sec_timeout signal is correctly asserted after 1 second when the enable signal is active. This setup ensures that the smaller timers work together to generate the 1-second timeout in a simplified manner for simulation.

This simulation checks the behavior of the oneSecTimer module and confirms that it produces a 1-second timeout after the enable signal is asserted, with the threshold values lowered for more manageable simulation times.

FPGA Board Testing Results

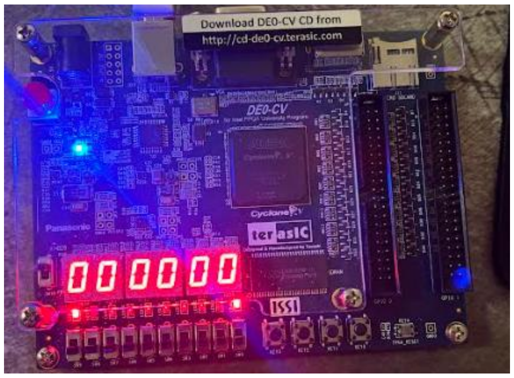


Figure 25. Default State / Reset State

This shows the board when it is first powered on and when it is reset.

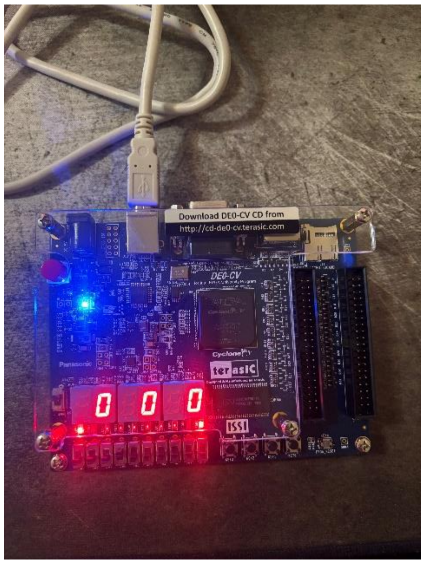


Figure 26. Logged In State

This shows the board whenever the player successfully logs in, as shown by the LED on the right turning on.

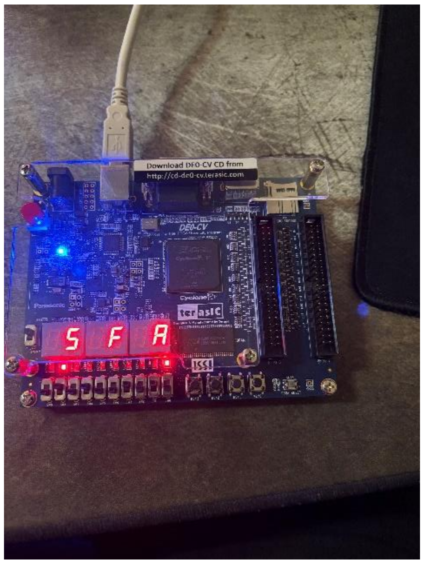


Figure 27. Matching Case

Player 1 inputs a binary 10 (A) and player 2 inputs a binary 5. This adds to 15 (1111 or F) and the matching LED lights up. This functions as expected.

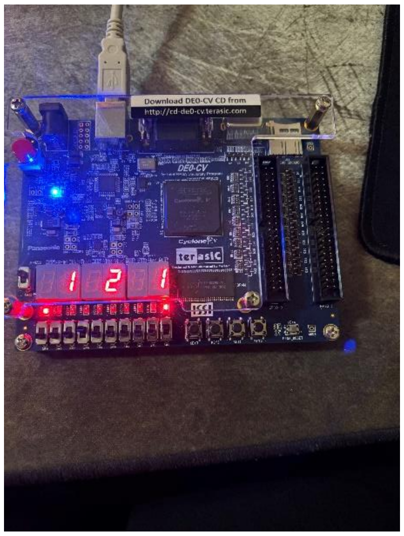


Figure 28. Non-matching Case

Player 1 inputs a binary 1 and player 2 inputs a binary 1. This adds to 2 and the non-matching LED lights up. This functions as expected.

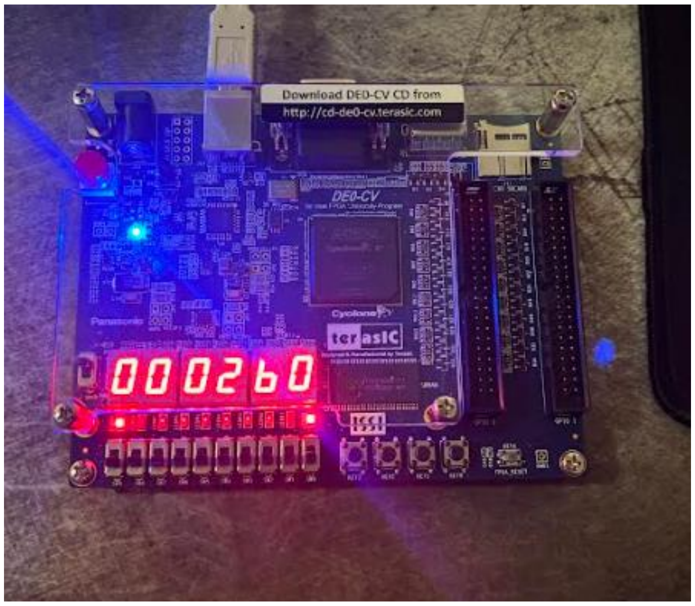


Figure 29. Game over showing score

Shows state of FPGA after game has ended and number of wins.

Video Demo

gameplay.mov

This shows gameplay after being logged in and demonstration of the load registers.

reset.MOV
This shows how the reset button operates.

password entry to log in.MOV

This shows logging in for the first time to operate the game.

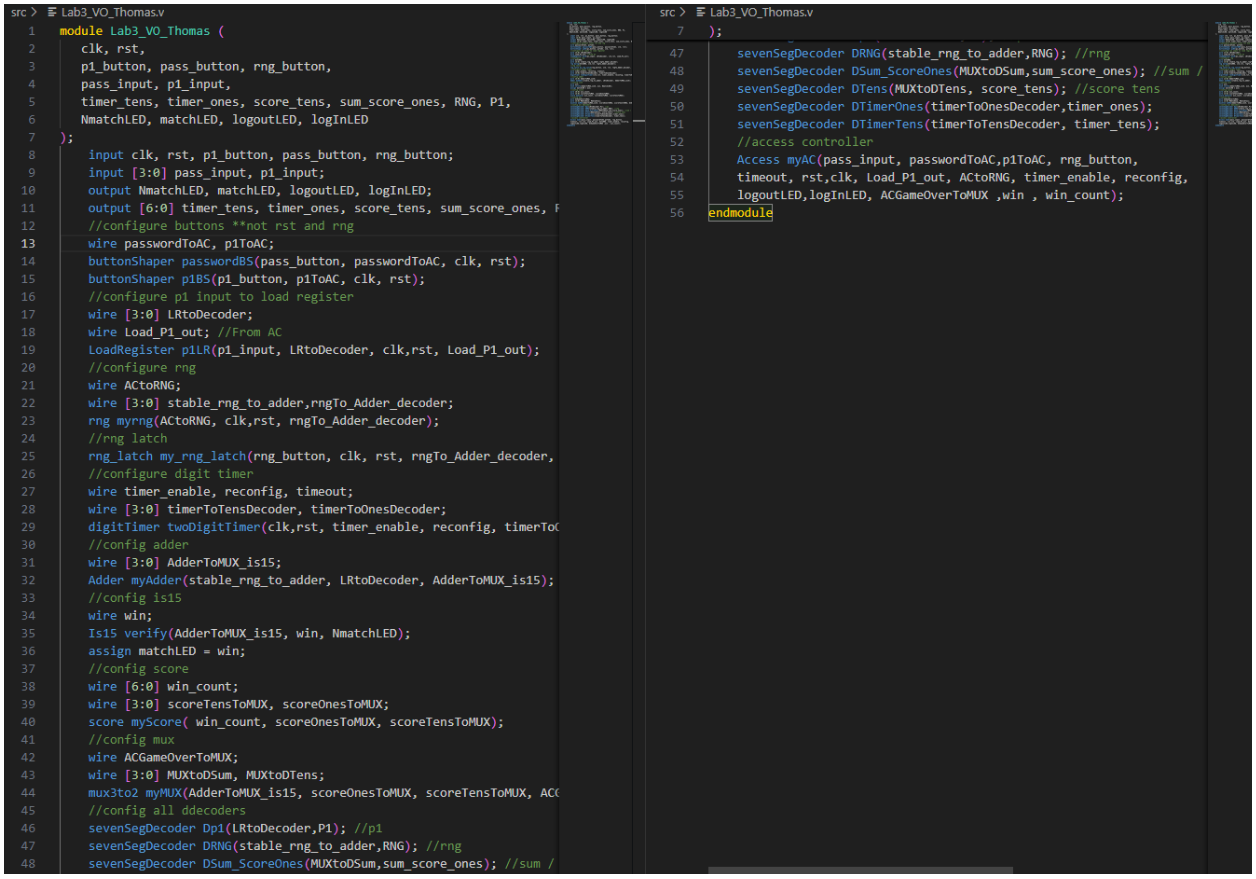
reconfigure.MOV

This shows the reconfigure feature

Conclusion

I have successfully built and completed lab 3, which was implementing a mental binary math game on an FPGA with load registers, button shapers, and access controller modules. The bonus features implemented were the matching, non-matching LEDs, and the automatic scoring.

Appendix



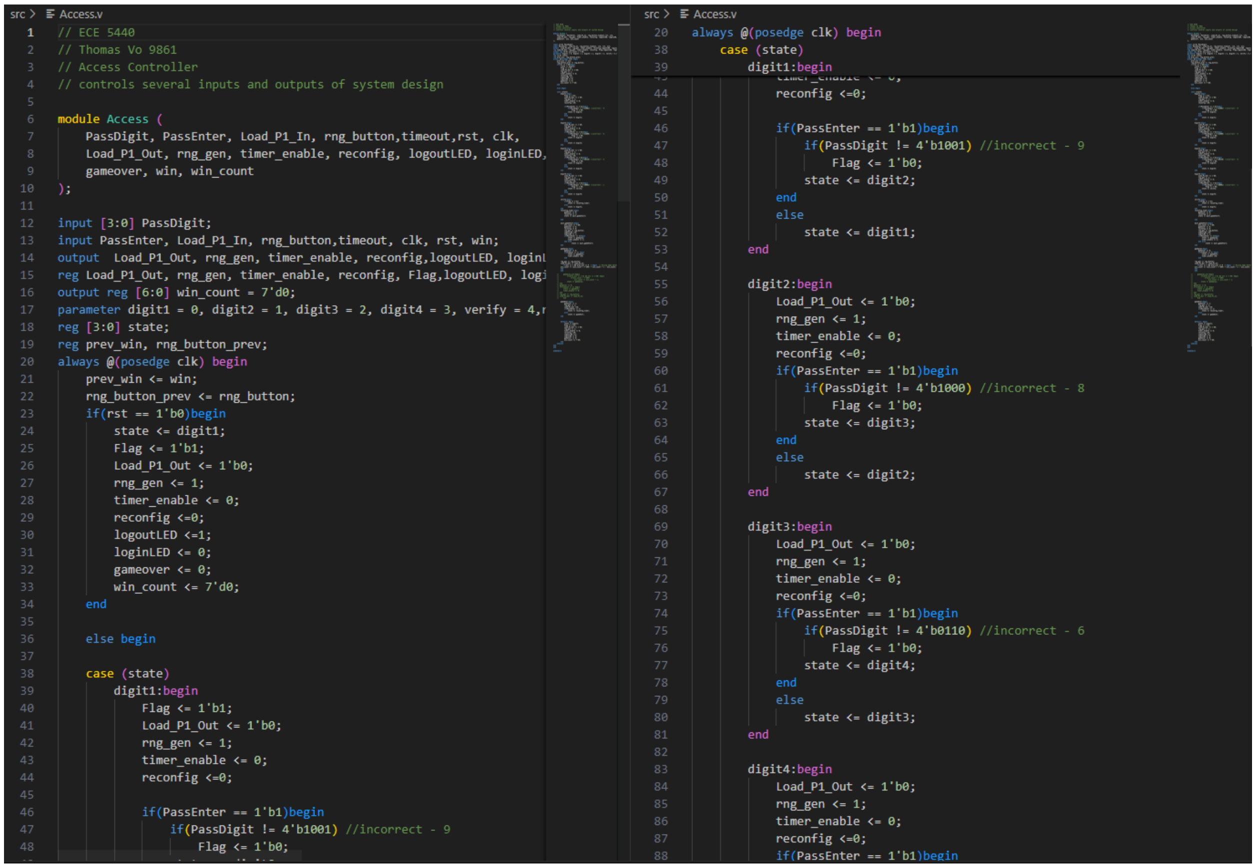
src > Lab3_VO_Thomas.v

```
module Lab3_VO_Thomas (
   clk, rst,
   p1_button, pass_button, rng_button,
   pass_input, p1_input,
   timer_tens, timer_ones, score_tens, sum_score_ones, RNG, P1,
   NmatchLED, matchLED, logoutLED, logInLED
);
    input clk, rst, p1_button, pass_button, rng_button;
    input [3:0] pass_input, p1_input;
    output NmatchLED, matchLED, logoutLED, logInLED;
    output [6:0] timer_tens, timer_ones, score_tens, sum_score_ones,
    //configure buttons **not rst and rng
    wire passwordToAC, p1ToAC;
    buttonShaper passwordBS(pass_button, passwordToAC, clk, rst);
    buttonShaper p1BS(p1_button, p1ToAC, clk, rst);
    //configure p1 input to load register
    wire [3:0] LRtoDecoder;
    wire Load_P1_out; //From AC
    LoadRegister p1LR(p1_input, LRtoDecoder, clk,rst, Load_P1_out);
    //configure rng
    wire ACtoRNG;
    wire [3:0] stable_rng_to_adder,rngTo_Adder_decoder;
    rng myrng(ACtoRNG, clk,rst, rngTo_Adder_decoder);
    //rng latch
    rng_latch my_rng_latch(rng_button, clk, rst, rngTo_Adder_decoder,
    //configure digit timer
    wire timer_enable, reconfig, timeout;
    wire [3:0] timerToTensDecoder, timerToOnesDecoder;
    digitTimer twoDigitTimer(clk,rst, timer_enable, reconfig, timerTo
    //config adder
    wire [3:0] AdderToMUX_is15;
    Adder myAdder(stable_rng_to_adder, LRtoDecoder, AdderToMUX_is15);
    //config is15
    wire win;
    Is15 verify(AdderToMUX_is15, win, NmatchLED);
    assign matchLED = win;
    //config score
    wire [6:0] win_count;
    wire [3:0] scoreTensToMUX, scoreOnesToMUX;
    score myScore( win_count, scoreOnesToMUX, scoreTensToMUX);
    //config mux
    wire ACGameOverToMUX;
    wire [3:0] MUXtoDSum, MUXtoDTens;
    mux3to2 myMUX(AdderToMUX_is15, scoreOnesToMUX, scoreTensToMUX, AC
    //config all ddecoders
    sevenSegDecoder Dp1(LRtoDecoder,P1); //p1
    sevenSegDecoder DRNG(stable_rng_to_adder,RNG); //rng
    sevenSegDecoder DSum_ScoreOnes(MUXtoDSum,sum_score_ones); //sum /
```

src > Lab3_VO_Thomas.v

```
);
    sevenSegDecoder DRNG(stable_rng_to_adder,RNG); //rng
    sevenSegDecoder DSum_ScoreOnes(MUXtoDSum,sum_score_ones); //sum /
    sevenSegDecoder DTens(MUXtoDTens, score_tens); //score tens
    sevenSegDecoder DTimerOnes(timerToOnesDecoder,timer_ones);
    sevenSegDecoder DTimerTens(timerToTensDecoder, timer_tens);
    //access controller
    Access myAC(pass_input, passwordToAC,p1ToAC, rng_button,
    timeout, rst,clk, Load_P1_out, ACtoRNG, timer_enable, reconfig,
    logoutLED,logInLED, ACGameOverToMUX ,win , win_count);
endmodule
```

Figure 30. Top Level Module Code



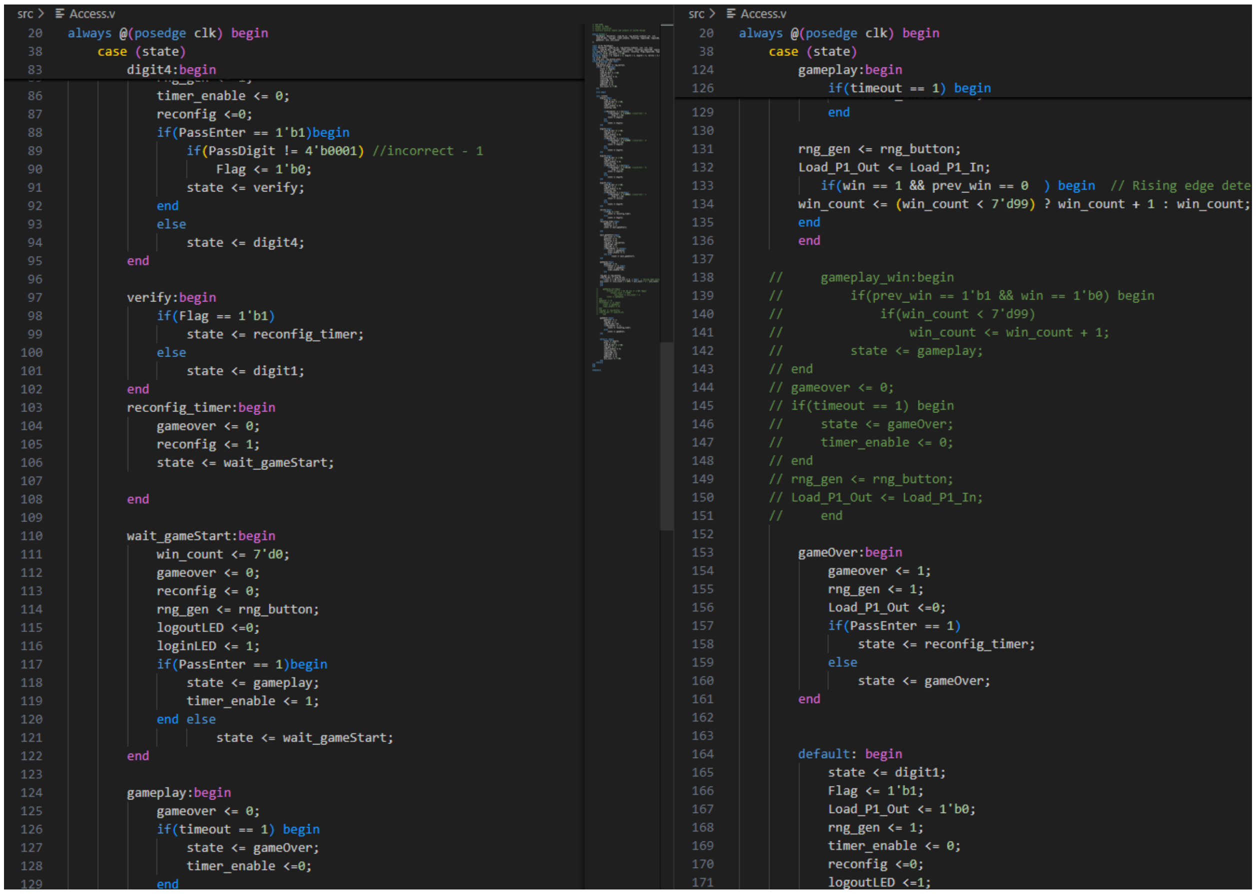
src > Access.v

```
// ECE 5440
// Thomas Vo 9861
// Access Controller
// controls several inputs and outputs of system design

module Access (
    PassDigit, PassEnter, Load_P1_In, rng_button,timeout,rst, clk,
    Load_P1_Out, rng_gen, timer_enable, reconfig, logoutLED, loginLED,
    gameover, win, win_count
);

input [3:0] PassDigit;
input PassEnter, Load_P1_In, rng_button,timeout, clk, rst, win;
output  Load_P1_Out, rng_gen, timer_enable, reconfig,logoutLED, loginL
reg Load_P1_Out, rng_gen, timer_enable, reconfig, Flag,logoutLED, logi
output reg [6:0] win_count = 7'd0;
parameter digit1 = 0, digit2 = 1, digit3 = 2, digit4 = 3, verify = 4,r
reg [3:0] state;
reg prev_win, rng_button_prev;
always @(posedge clk) begin
    prev_win <= win;
    rng_button_prev <= rng_button;
    if(rst == 1'b0)begin
        state <= digit1;
        Flag <= 1'b1;
        Load_P1_Out <= 1'b0;
        rng_gen <= 1;
        timer_enable <= 0;
        reconfig <=0;
        logoutLED <=1;
        loginLED <= 0;
        gameover <= 0;
        win_count <= 7'd0;
    end

    else begin

    case (state)
        digit1:begin
            Flag <= 1'b1;
            Load_P1_Out <= 1'b0;
            rng_gen <= 1;
            timer_enable <= 0;
            reconfig <=0;

            if(PassEnter == 1'b1)begin
                if(PassDigit != 4'b1001) //incorrect - 9
                    Flag <= 1'b0;
```

src > Access.v

```
always @(posedge clk) begin
    case (state)
        digit1:begin
            reconfig <=0;

            if(PassEnter == 1'b1)begin
                if(PassDigit != 4'b1001) //incorrect - 9
                    Flag <= 1'b0;
                state <= digit2;
            end
            else
                state <= digit1;
        end

        digit2:begin
            Load_P1_Out <= 1'b0;
            rng_gen <= 1;
            timer_enable <= 0;
            reconfig <=0;
            if(PassEnter == 1'b1)begin
                if(PassDigit != 4'b1000) //incorrect - 8
                    Flag <= 1'b0;
                state <= digit3;
            end
            else
                state <= digit2;
        end

        digit3:begin
            Load_P1_Out <= 1'b0;
            rng_gen <= 1;
            timer_enable <= 0;
            reconfig <=0;
            if(PassEnter == 1'b1)begin
                if(PassDigit != 4'b0110) //incorrect - 6
                    Flag <= 1'b0;
                state <= digit4;
            end
            else
                state <= digit3;
        end

        digit4:begin
            Load_P1_Out <= 1'b0;
            rng_gen <= 1;
            timer_enable <= 0;
            reconfig <=0;
            if(PassEnter == 1'b1)begin
```

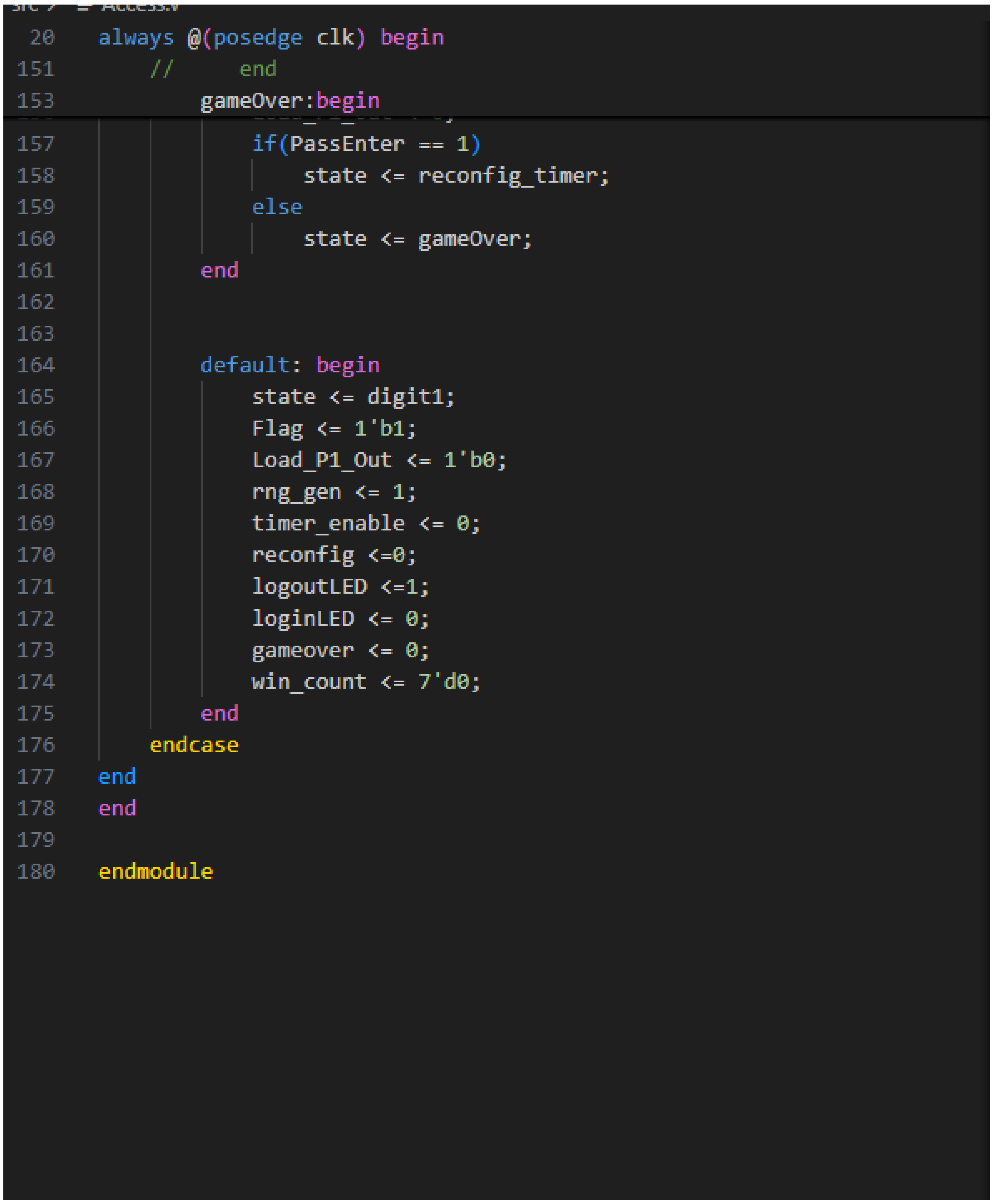
Figure 31. Access Controller Code 1/3



```
src > Access.v
always @(posedge clk) begin
    case (state)
        digit4:begin
            timer_enable <= 0;
            reconfig <=0;
            if(PassEnter == 1'b1)begin
                if(PassDigit != 4'b0001) //incorrect - 1
                    Flag <= 1'b0;
                state <= verify;
            end
            else
                state <= digit4;
        end

        verify:begin
            if(Flag == 1'b1)
                state <= reconfig_timer;
            else
                state <= digit1;
        end
        reconfig_timer:begin
            gameover <= 0;
            reconfig <= 1;
            state <= wait_gameStart;

        end

        wait_gameStart:begin
            win_count <= 7'd0;
            gameover <= 0;
            reconfig <= 0;
            rng_gen <= rng_button;
            logoutLED <=0;
            loginLED <= 1;
            if(PassEnter == 1)begin
                state <= gameplay;
                timer_enable <= 1;
            end else
                    state <= wait_gameStart;
        end

        gameplay:begin
            gameover <= 0;
            if(timeout == 1) begin
                state <= gameOver;
                timer_enable <=0;
            end
```

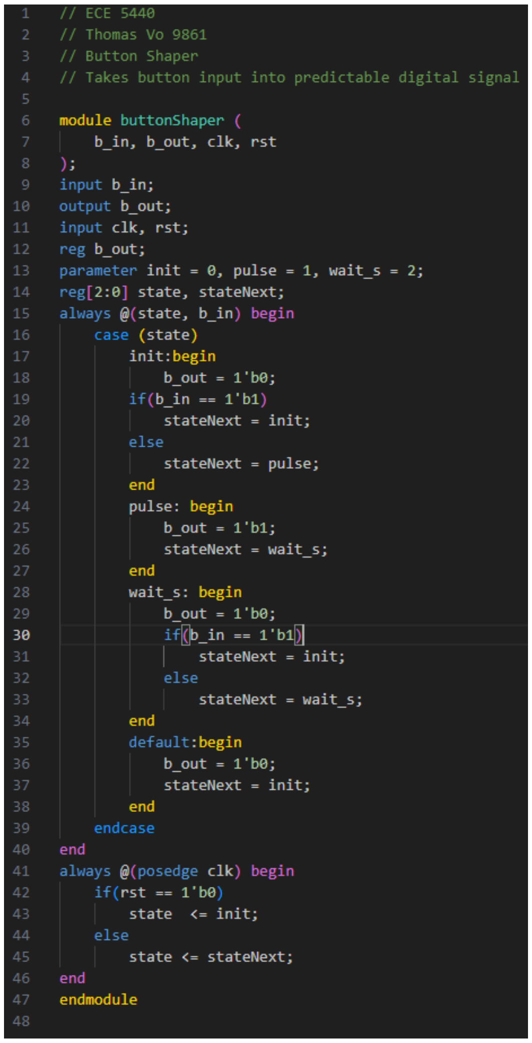
```
src > Access.v
always @(posedge clk) begin
    case (state)
        gameplay:begin
            if(timeout == 1) begin
            end

        rng_gen <= rng_button;
        Load_P1_Out <= Load_P1_In;
           if(win == 1 && prev_win == 0  ) begin  // Rising edge dete
        win_count <= (win_count < 7'd99) ? win_count + 1 : win_count;
        end
        end

    //     gameplay_win:begin
    //         if(prev_win == 1'b1 && win == 1'b0) begin
    //             if(win_count < 7'd99)
    //                 win_count <= win_count + 1;
    //         state <= gameplay;
    // end
    // gameover <= 0;
    // if(timeout == 1) begin
    //     state <= gameOver;
    //     timer_enable <= 0;
    // end
    // rng_gen <= rng_button;
    // Load_P1_Out <= Load_P1_In;
    //     end

        gameOver:begin
            gameover <= 1;
            rng_gen <= 1;
            Load_P1_Out <=0;
            if(PassEnter == 1)
                state <= reconfig_timer;
            else
                state <= gameOver;
        end


        default: begin
            state <= digit1;
            Flag <= 1'b1;
            Load_P1_Out <= 1'b0;
            rng_gen <= 1;
            timer_enable <= 0;
            reconfig <=0;
            logoutLED <=1;
```

Figure 32. Access Controller Code 2/3



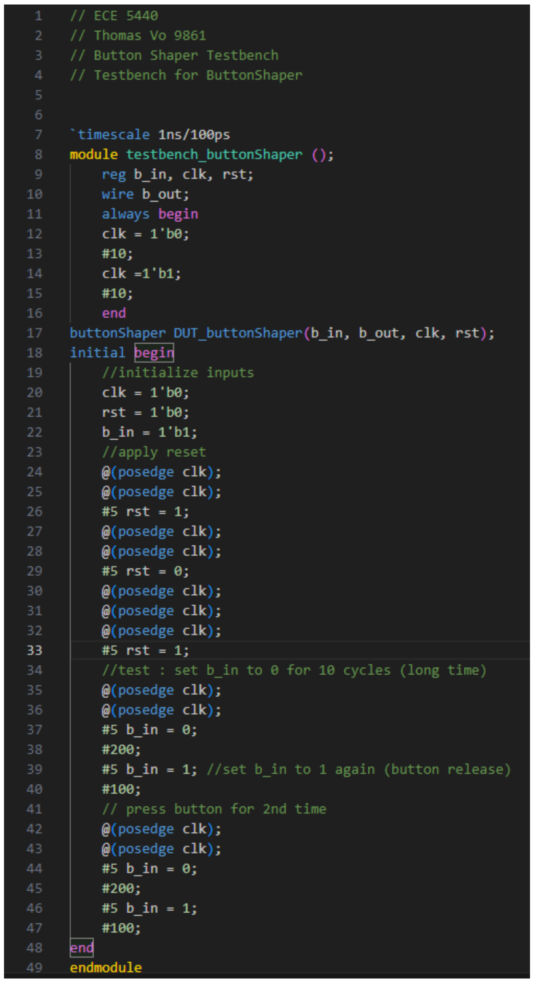
```
always @(posedge clk) begin
    //     end
        gameOver:begin
            if(PassEnter == 1)
                state <= reconfig_timer;
            else
                state <= gameOver;
        end


        default: begin
            state <= digit1;
            Flag <= 1'b1;
            Load_P1_Out <= 1'b0;
            rng_gen <= 1;
            timer_enable <= 0;
            reconfig <=0;
            logoutLED <=1;
            loginLED <= 0;
            gameover <= 0;
            win_count <= 7'd0;
        end
    endcase
end
end

endmodule
```

Figure 33. Access Controller Code 3/3



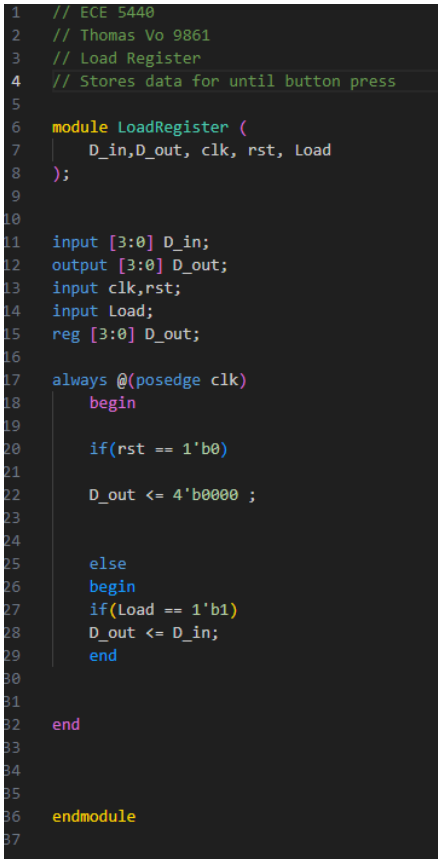
```
// ECE 5440
// Thomas Vo 9861
// Button Shaper
// Takes button input into predictable digital signal

module buttonShaper (
    b_in, b_out, clk, rst
);
input b_in;
output b_out;
input clk, rst;
reg b_out;
parameter init = 0, pulse = 1, wait_s = 2;
reg[2:0] state, stateNext;
always @(state, b_in) begin
    case (state)
        init:begin
            b_out = 1'b0;
        if(b_in == 1'b1)
            stateNext = init;
        else
            stateNext = pulse;
        end
        pulse: begin
            b_out = 1'b1;
            stateNext = wait_s;
        end
        wait_s: begin
            b_out = 1'b0;
            if(b_in == 1'b1)
                stateNext = init;
            else
                stateNext = wait_s;
        end
        default:begin
            b_out = 1'b0;
            stateNext = init;
        end
    endcase
end
always @(posedge clk) begin
    if(rst == 1'b0)
        state  <= init;
    else
        state <= stateNext;
end
endmodule
```

Figure 34. Button Shaper Code



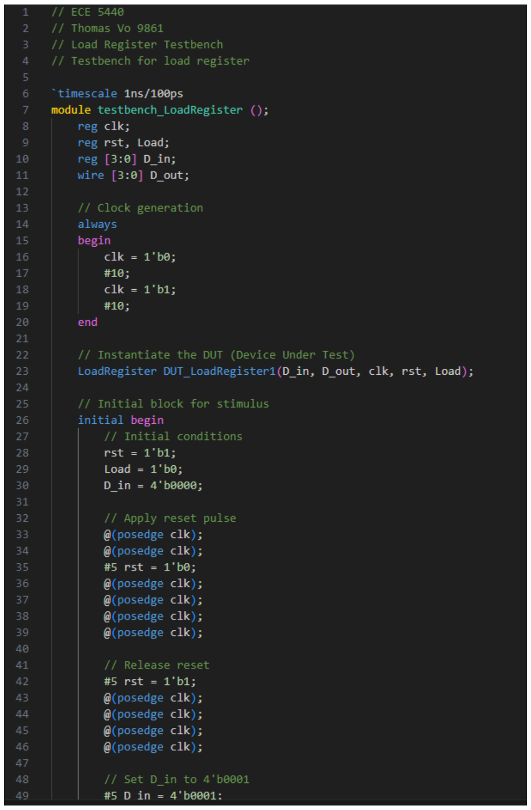
```verilog
// ECE 5440
// Thomas Vo 9861
// Button Shaper Testbench
// Testbench for ButtonShaper


`timescale 1ns/100ps
module testbench_buttonShaper ();
    reg b_in, clk, rst;
    wire b_out;
    always begin
    clk = 1'b0;
    #10;
    clk =1'b1;
    #10;
    end
buttonShaper DUT_buttonShaper(b_in, b_out, clk, rst);
initial begin
    //initialize inputs
    clk = 1'b0;
    rst = 1'b0;
    b_in = 1'b1;
    //apply reset
    @(posedge clk);
    @(posedge clk);
    #5 rst = 1;
    @(posedge clk);
    @(posedge clk);
    #5 rst = 0;
    @(posedge clk);
    @(posedge clk);
    @(posedge clk);
    #5 rst = 1;
    //test : set b_in to 0 for 10 cycles (long time)
    @(posedge clk);
    @(posedge clk);
    #5 b_in = 0;
    #200;
    #5 b_in = 1; //set b_in to 1 again (button release)
    #100;
    // press button for 2nd time
    @(posedge clk);
    @(posedge clk);
    #5 b_in = 0;
    #200;
    #5 b_in = 1;
    #100;
end
endmodule
```

Figure 35. Button Shaper Testbench Code



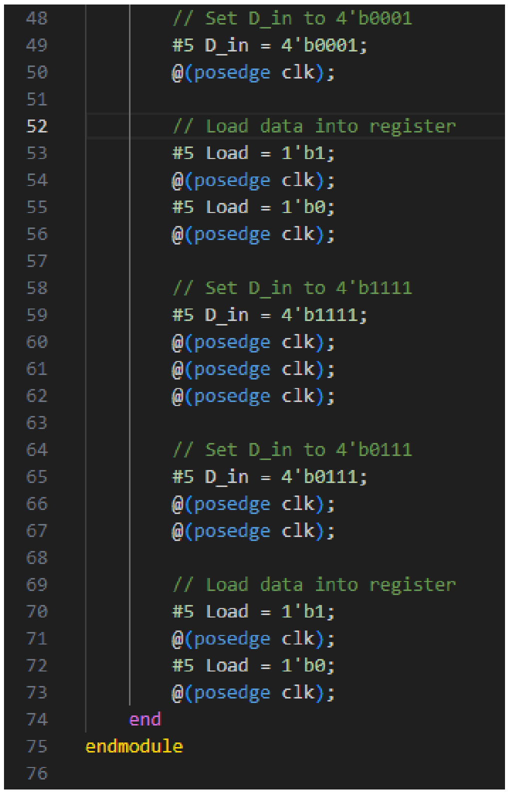
```
// ECE 5440
// Thomas Vo 9861
// Load Register
// Stores data for until button press

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)

    D_out <= 4'b0000 ;


    else
    begin
    if(Load == 1'b1)
    D_out <= D_in;
    end


end



endmodule

```

Figure 36. Load Register Code



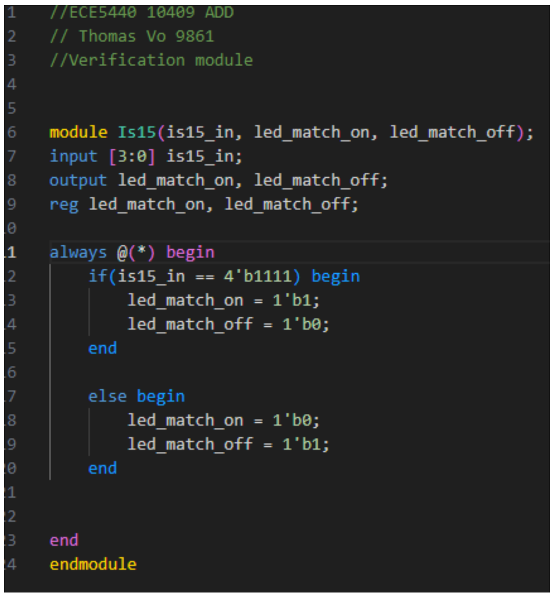
```verilog
// ECE 5440
// Thomas Vo 9861
// Load Register Testbench
// Testbench for load register

`timescale 1ns/100ps
module testbench_LoadRegister ();
    reg clk;
    reg rst, Load;
    reg [3:0] D_in;
    wire [3:0] D_out;

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

    // Instantiate the DUT (Device Under Test)
    LoadRegister DUT_LoadRegister1(D_in, D_out, clk, rst, Load);

    // Initial block for stimulus
    initial begin
        // Initial conditions
        rst = 1'b1;
        Load = 1'b0;
        D_in = 4'b0000;

        // Apply reset pulse
        @(posedge clk);
        @(posedge clk);
        #5 rst = 1'b0;
        @(posedge clk);
        @(posedge clk);
        @(posedge clk);
        @(posedge clk);

        // Release reset
        #5 rst = 1'b1;
        @(posedge clk);
        @(posedge clk);
        @(posedge clk);
        @(posedge clk);

        // Set D_in to 4'b0001
        #5 D in = 4'b0001:
```

Figure 37. Load Register Testbench Code 1/2



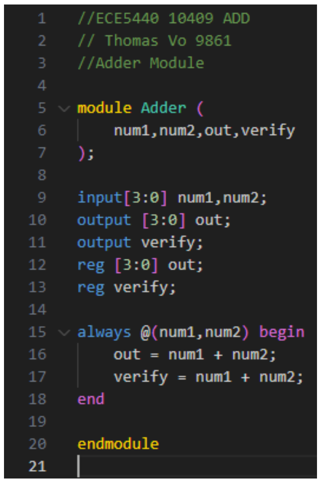
```verilog
        // Set D_in to 4'b0001
        #5 D_in = 4'b0001;
        @(posedge clk);

        // Load data into register
        #5 Load = 1'b1;
        @(posedge clk);
        #5 Load = 1'b0;
        @(posedge clk);

        // Set D_in to 4'b1111
        #5 D_in = 4'b1111;
        @(posedge clk);
        @(posedge clk);
        @(posedge clk);

        // Set D_in to 4'b0111
        #5 D_in = 4'b0111;
        @(posedge clk);
        @(posedge clk);

        // Load data into register
        #5 Load = 1'b1;
        @(posedge clk);
        #5 Load = 1'b0;
        @(posedge clk);
    end
endmodule
```

Figure 38. Load Register Testbench Code 2/2



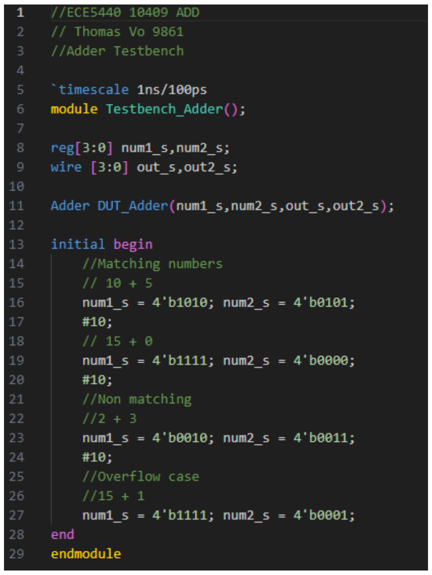
```verilog
//ECE5440 10409 ADD
// Thomas Vo 9861
//Verification module


module Is15(is15_in, led_match_on, led_match_off);
input [3:0] is15_in;
output led_match_on, led_match_off;
reg led_match_on, led_match_off;

always @(*) begin
    if(is15_in == 4'b1111) begin
        led_match_on = 1'b1;
        led_match_off = 1'b0;
    end

    else begin
        led_match_on = 1'b0;
        led_match_off = 1'b1;
    end


end
endmodule
```

Figure 39. Is15 Verification Module Code



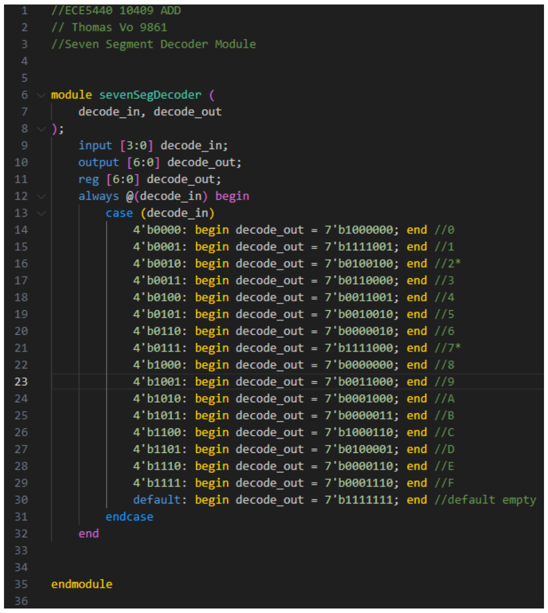
```verilog
//ECE5440 10409 ADD
// Thomas Vo 9861
//Adder Module

module Adder (
    num1,num2,out,verify
);

input[3:0] num1,num2;
output [3:0] out;
output verify;
reg [3:0] out;
reg verify;

always @(num1,num2) begin
    out = num1 + num2;
    verify = num1 + num2;
end

endmodule

```

Figure 40. Adder Module Code



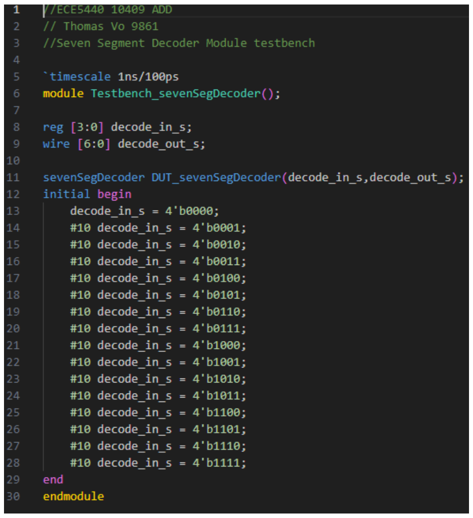
```verilog
//ECE5440 10409 ADD
// Thomas Vo 9861
//Adder Testbench

`timescale 1ns/100ps
module Testbench_Adder();

reg[3:0] num1_s,num2_s;
wire [3:0] out_s,out2_s;

Adder DUT_Adder(num1_s,num2_s,out_s,out2_s);

initial begin
    //Matching numbers
    // 10 + 5
    num1_s = 4'b1010; num2_s = 4'b0101;
    #10;
    // 15 + 0
    num1_s = 4'b1111; num2_s = 4'b0000;
    #10;
    //Non matching
    //2 + 3
    num1_s = 4'b0010; num2_s = 4'b0011;
    #10;
    //Overflow case
    //15 + 1
    num1_s = 4'b1111; num2_s = 4'b0001;
end
endmodule
```

Figure 41. Adder Testbench Code



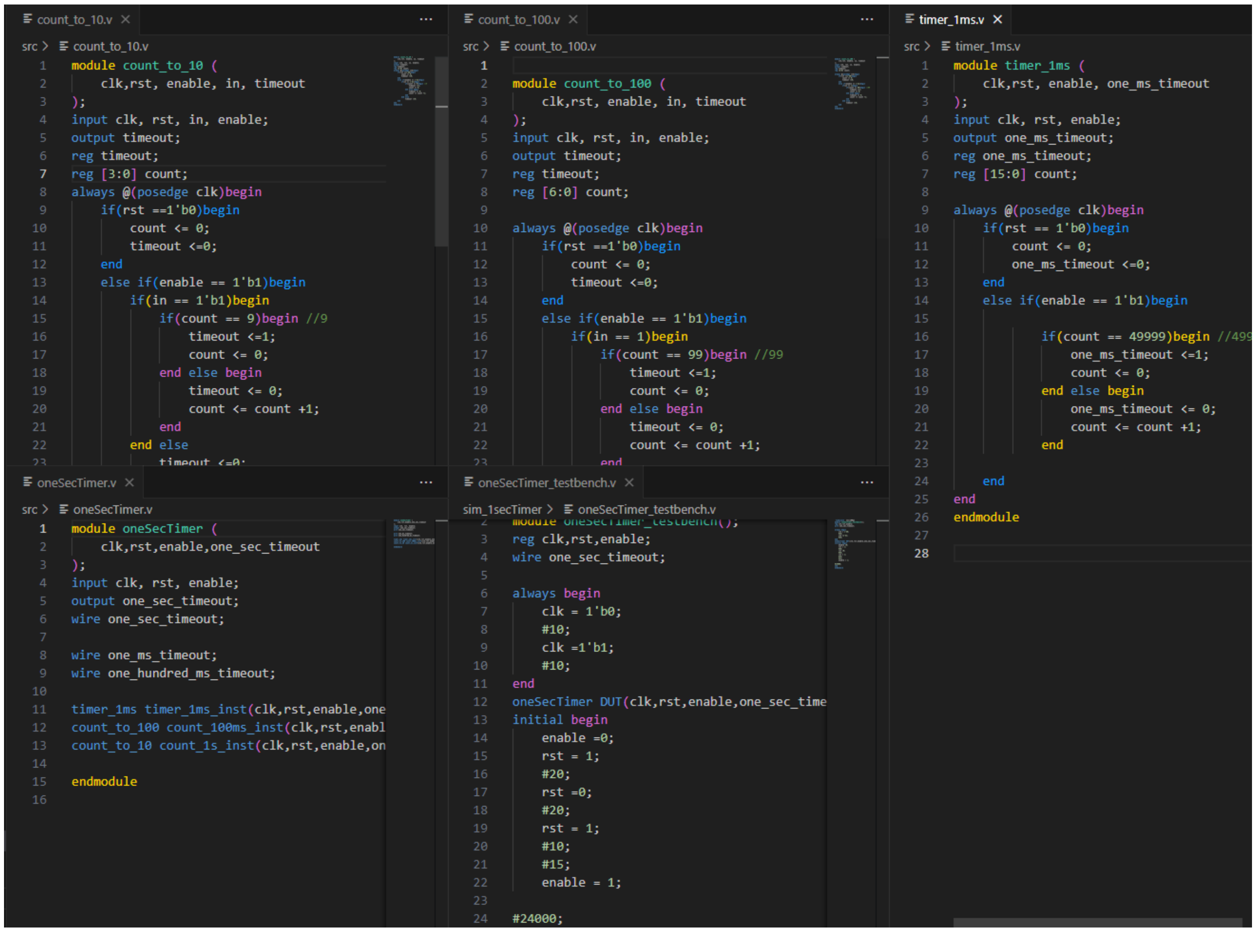
```verilog
//ECE5440 10409 ADD
// Thomas Vo 9861
//Seven Segment Decoder Module


module sevenSegDecoder (
    decode_in, decode_out
);
    input [3:0] decode_in;
    output [6:0] decode_out;
    reg [6:0] decode_out;
    always @(decode_in) begin
        case (decode_in)
            4'b0000: begin decode_out = 7'b1000000; end //0
            4'b0001: begin decode_out = 7'b1111001; end //1
            4'b0010: begin decode_out = 7'b0100100; end //2*
            4'b0011: begin decode_out = 7'b0110000; end //3
            4'b0100: begin decode_out = 7'b0011001; end //4
            4'b0101: begin decode_out = 7'b0010010; end //5
            4'b0110: begin decode_out = 7'b0000010; end //6
            4'b0111: begin decode_out = 7'b1111000; end //7*
            4'b1000: begin decode_out = 7'b0000000; end //8
            4'b1001: begin decode_out = 7'b0011000; end //9
            4'b1010: begin decode_out = 7'b0001000; end //A
            4'b1011: begin decode_out = 7'b0000011; end //B
            4'b1100: begin decode_out = 7'b1000110; end //C
            4'b1101: begin decode_out = 7'b0100001; end //D
            4'b1110: begin decode_out = 7'b0000110; end //E
            4'b1111: begin decode_out = 7'b0001110; end //F
            default: begin decode_out = 7'b1111111; end //default empty
        endcase
    end


endmodule
```

Figure 42. Code for sevenSegDecoder Module



```verilog
//ECE5440 10409 ADD
// Thomas Vo 9861
//Seven Segment Decoder Module testbench

`timescale 1ns/100ps
module Testbench_sevenSegDecoder();

reg [3:0] decode_in_s;
wire [6:0] decode_out_s;

sevenSegDecoder DUT_sevenSegDecoder(decode_in_s,decode_out_s);
initial begin
    decode_in_s = 4'b0000;
    #10 decode_in_s = 4'b0001;
    #10 decode_in_s = 4'b0010;
    #10 decode_in_s = 4'b0011;
    #10 decode_in_s = 4'b0100;
    #10 decode_in_s = 4'b0101;
    #10 decode_in_s = 4'b0110;
    #10 decode_in_s = 4'b0111;
    #10 decode_in_s = 4'b1000;
    #10 decode_in_s = 4'b1001;
    #10 decode_in_s = 4'b1010;
    #10 decode_in_s = 4'b1011;
    #10 decode_in_s = 4'b1100;
    #10 decode_in_s = 4'b1101;
    #10 decode_in_s = 4'b1110;
    #10 decode_in_s = 4'b1111;
end
endmodule
```

Figure 43. Seven Segment Decoder Testbench Code



count_to_10.v

src > count_to_10.v

```verilog
module count_to_10 (
    clk,rst, enable, in, timeout
);
input clk, rst, in, enable;
output timeout;
reg timeout;
reg [3:0] count;
always @(posedge clk)begin
    if(rst ==1'b0)begin
        count <= 0;
        timeout <=0;
    end
    else if(enable == 1'b1)begin
        if(in == 1'b1)begin
            if(count == 9)begin //9
                timeout <=1;
                count <= 0;
            end else begin
                timeout <= 0;
                count <= count +1;
            end
        end else
            timeout <=0;
```

count_to_100.v

src > count_to_100.v

```verilog

module count_to_100 (
    clk,rst, enable, in, timeout
);
input clk, rst, in, enable;
output timeout;
reg timeout;
reg [6:0] count;

always @(posedge clk)begin
    if(rst ==1'b0)begin
        count <= 0;
        timeout <=0;
    end
    else if(enable == 1'b1)begin
        if(in == 1)begin
            if(count == 99)begin //99
                timeout <=1;
                count <= 0;
            end else begin
                timeout <= 0;
                count <= count +1;
            end
```

timer_1ms.v

src > timer_1ms.v

```verilog
module timer_1ms (
    clk,rst, enable, one_ms_timeout
);
input clk, rst, enable;
output one_ms_timeout;
reg one_ms_timeout;
reg [15:0] count;

always @(posedge clk)begin
    if(rst == 1'b0)begin
        count <= 0;
        one_ms_timeout <=0;
    end
    else if(enable == 1'b1)begin

            if(count == 49999)begin //499
                one_ms_timeout <=1;
                count <= 0;
            end else begin
                one_ms_timeout <= 0;
                count <= count +1;
            end

    end
end
endmodule

```

oneSecTimer.v

src > oneSecTimer.v

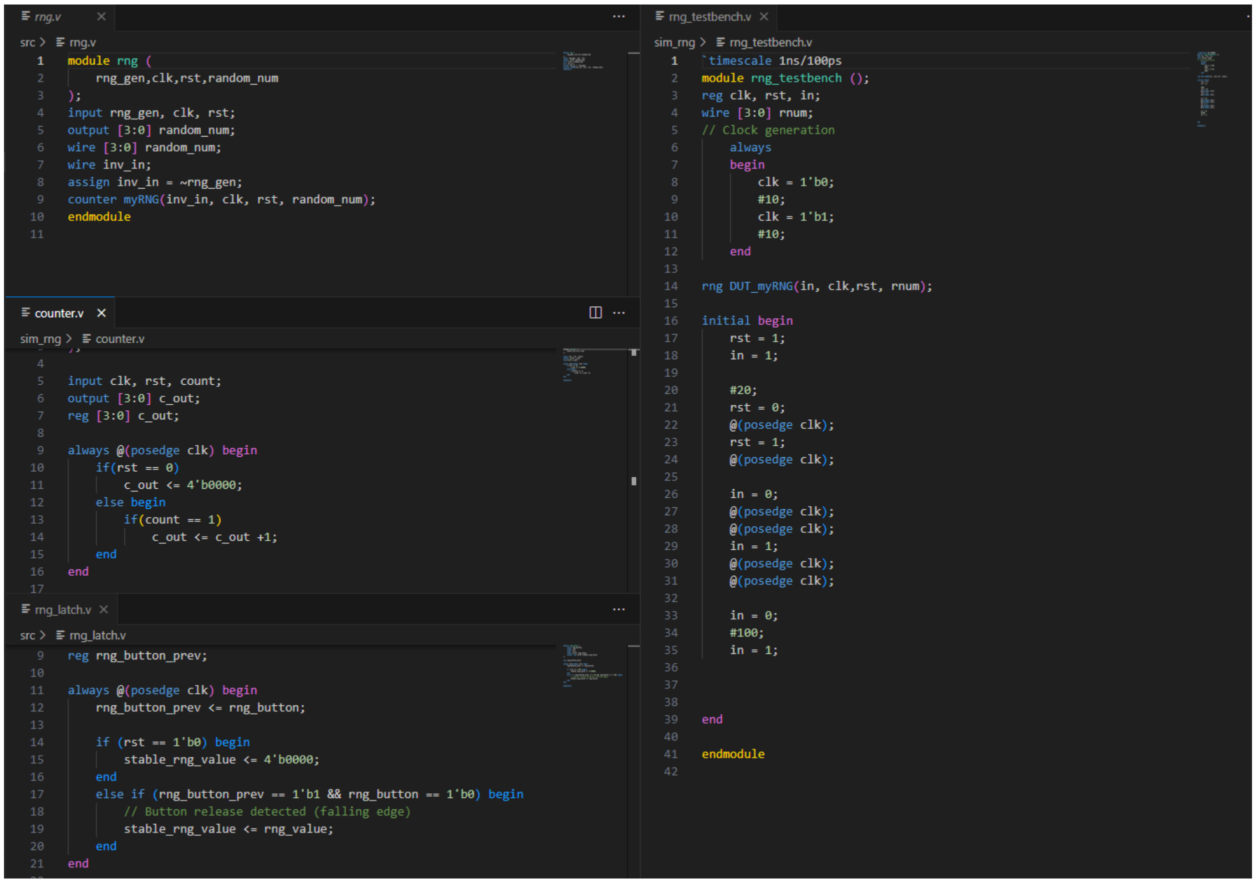
```verilog
module oneSecTimer (
    clk,rst,enable,one_sec_timeout
);
input clk, rst, enable;
output one_sec_timeout;
wire one_sec_timeout;

wire one_ms_timeout;
wire one_hundred_ms_timeout;

timer_1ms timer_1ms_inst(clk,rst,enable,one
count_to_100 count_100ms_inst(clk,rst,enabl
count_to_10 count_1s_inst(clk,rst,enable,on

endmodule

```

oneSecTimer_testbench.v

sim_1secTimer > oneSecTimer_testbench.v

```verilog
module oneSecTimer_testbench();
reg clk,rst,enable;
wire one_sec_timeout;

always begin
    clk = 1'b0;
    #10;
    clk =1'b1;
    #10;
end
oneSecTimer DUT(clk,rst,enable,one_sec_time
initial begin
    enable =0;
    rst = 1;
    #20;
    rst =0;
    #20;
    rst = 1;
    #10;
    #15;
    enable = 1;

#24000;
```

Figure 44. oneSecTimer, sub Module Code, and testbench code



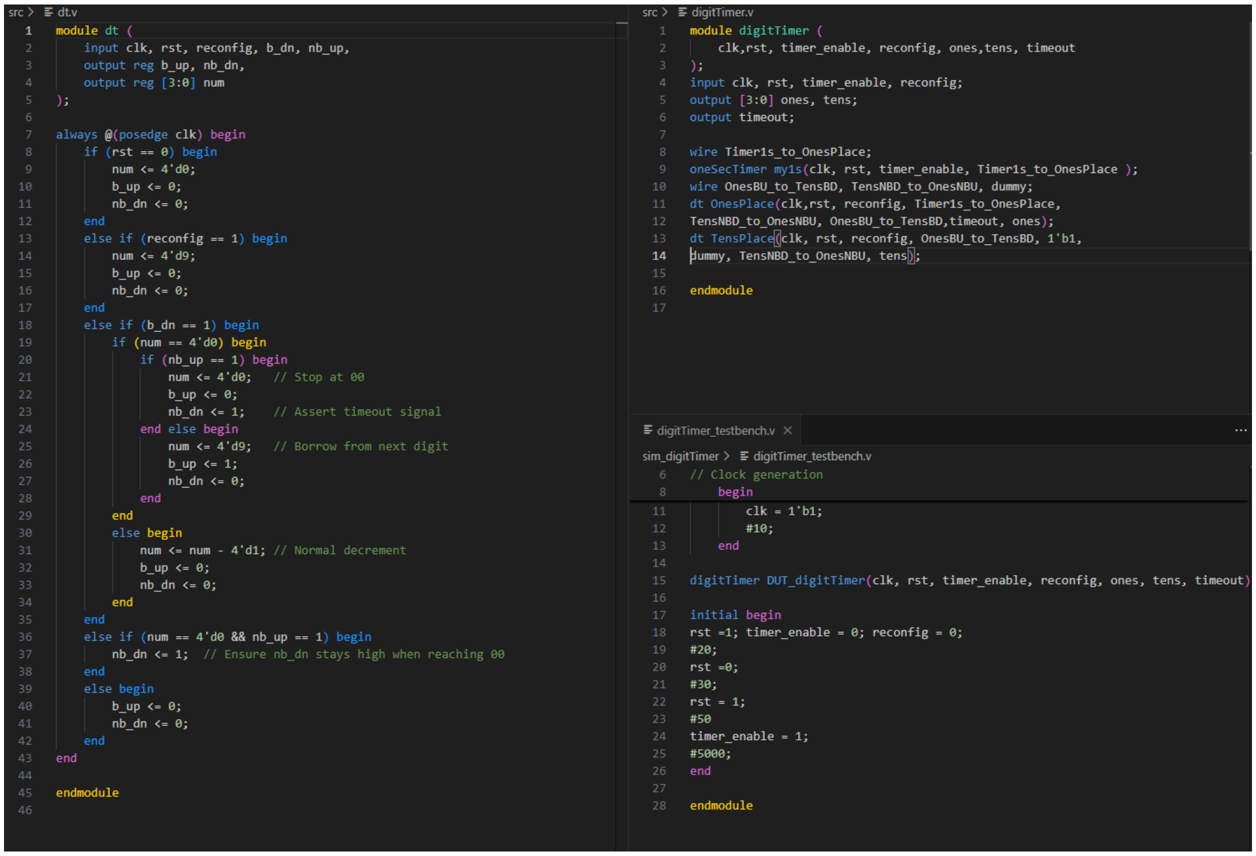
```verilog
module rng (
    rng_gen,clk,rst,random_num
);
input rng_gen, clk, rst;
output [3:0] random_num;
wire [3:0] random_num;
wire inv_in;
assign inv_in = ~rng_gen;
counter myRNG(inv_in, clk, rst, random_num);
endmodule
```

```verilog

input clk, rst, count;
output [3:0] c_out;
reg [3:0] c_out;

always @(posedge clk) begin
    if(rst == 0)
        c_out <= 4'b0000;
    else begin
        if(count == 1)
            c_out <= c_out +1;
    end
end

```

```verilog
reg rng_button_prev;

always @(posedge clk) begin
    rng_button_prev <= rng_button;

    if (rst == 1'b0) begin
        stable_rng_value <= 4'b0000;
    end
    else if (rng_button_prev == 1'b1 && rng_button == 1'b0) begin
        // Button release detected (falling edge)
        stable_rng_value <= rng_value;
    end
end
```

```verilog
`timescale 1ns/100ps
module rng_testbench ();
reg clk, rst, in;
wire [3:0] rnum;
// Clock generation
    always
    begin
        clk = 1'b0;
        #10;
        clk = 1'b1;
        #10;
    end

rng DUT_myRNG(in, clk,rst, rnum);

initial begin
    rst = 1;
    in = 1;

    #20;
    rst = 0;
    @(posedge clk);
    rst = 1;
    @(posedge clk);

    in = 0;
    @(posedge clk);
    @(posedge clk);
    in = 1;
    @(posedge clk);
    @(posedge clk);

    in = 0;
    #100;
    in = 1;



end

endmodule
```

Figure 45. RNG, sub module, and testbench code


src > dt.v

```verilog
module dt (
    input clk, rst, reconfig, b_dn, nb_up,
    output reg b_up, nb_dn,
    output reg [3:0] num
);

always @(posedge clk) begin
    if (rst == 0) begin
        num <= 4'd0;
        b_up <= 0;
        nb_dn <= 0;
    end
    else if (reconfig == 1) begin
        num <= 4'd9;
        b_up <= 0;
        nb_dn <= 0;
    end
    else if (b_dn == 1) begin
        if (num == 4'd0) begin
            if (nb_up == 1) begin
                num <= 4'd0;   // Stop at 00
                b_up <= 0;
                nb_dn <= 1;    // Assert timeout signal
            end else begin
                num <= 4'd9;   // Borrow from next digit
                b_up <= 1;
                nb_dn <= 0;
            end
        end
        else begin
            num <= num - 4'd1; // Normal decrement
            b_up <= 0;
            nb_dn <= 0;
        end
    end
    else if (num == 4'd0 && nb_up == 1) begin
        nb_dn <= 1;  // Ensure nb_dn stays high when reaching 00
    end
    else begin
        b_up <= 0;
        nb_dn <= 0;
    end
end

endmodule
```

src > digitTimer.v

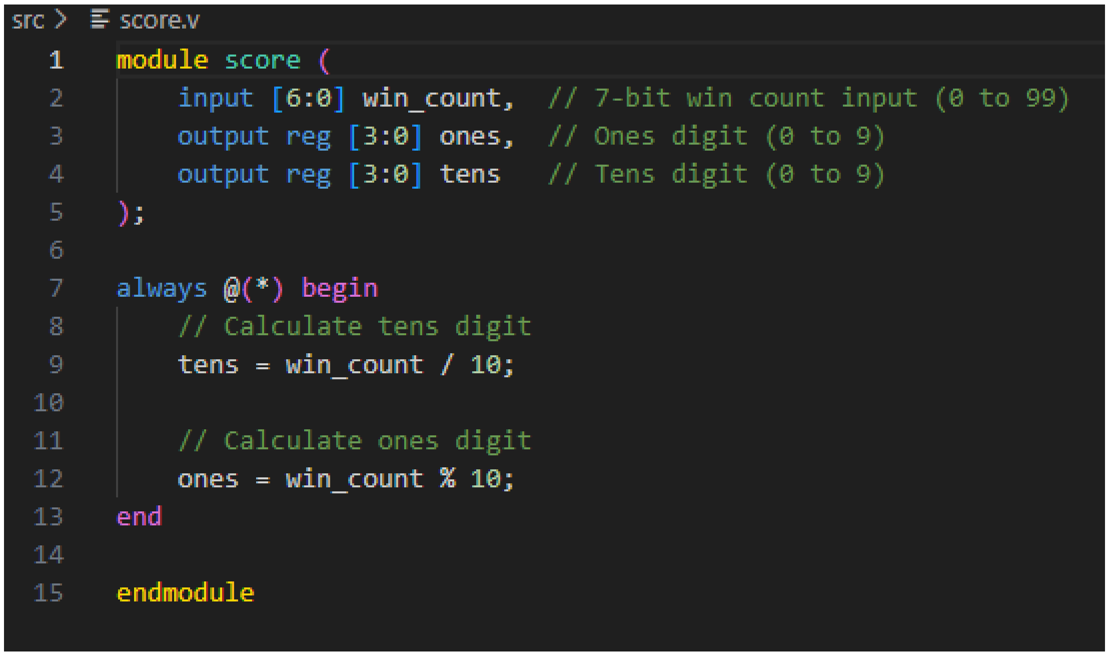
```verilog
module digitTimer (
    clk,rst, timer_enable, reconfig, ones,tens, timeout
);
input clk, rst, timer_enable, reconfig;
output [3:0] ones, tens;
output timeout;

wire Timer1s_to_OnesPlace;
oneSecTimer my1s(clk, rst, timer_enable, Timer1s_to_OnesPlace );
wire OnesBU_to_TensBD, TensNBD_to_OnesNBU, dummy;
dt OnesPlace(clk,rst, reconfig, Timer1s_to_OnesPlace,
TensNBD_to_OnesNBU, OnesBU_to_TensBD,timeout, ones);
dt TensPlace(clk, rst, reconfig, OnesBU_to_TensBD, 1'b1,
dummy, TensNBD_to_OnesNBU, tens);

endmodule
```

digitTimer_testbench.v

sim_digitTimer > digitTimer_testbench.v

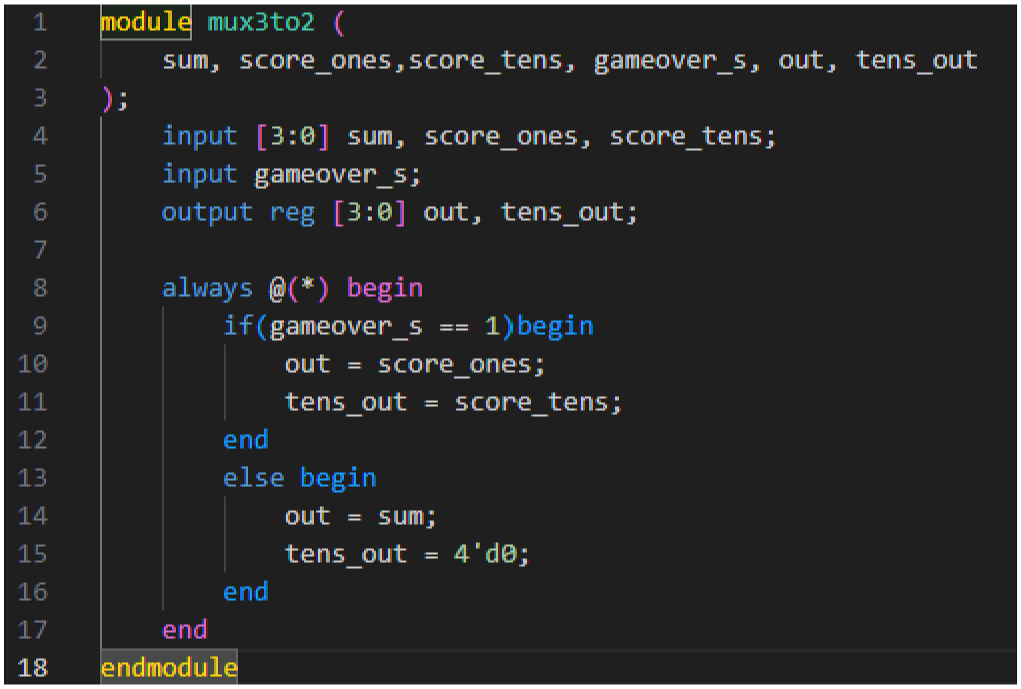
```verilog
// Clock generation
    begin
        clk = 1'b1;
        #10;
    end

digitTimer DUT_digitTimer(clk, rst, timer_enable, reconfig, ones, tens, timeout)

initial begin
rst =1; timer_enable = 0; reconfig = 0;
#20;
rst =0;
#30;
rst = 1;
#50
timer_enable = 1;
#5000;
end

endmodule
```

Figure 46. digitTimer, sub Module, and testbench code


src > score.v

```verilog
module score (
    input [6:0] win_count,   // 7-bit win count input (0 to 99)
    output reg [3:0] ones,   // Ones digit (0 to 9)
    output reg [3:0] tens    // Tens digit (0 to 9)
);

always @(*) begin
    // Calculate tens digit
    tens = win_count / 10;

    // Calculate ones digit
    ones = win_count % 10;
end

endmodule
```

Figure 47. Score Module Code



```verilog
module mux3to2 (
    sum, score_ones,score_tens, gameover_s, out, tens_out
);
    input [3:0] sum, score_ones, score_tens;
    input gameover_s;
    output reg [3:0] out, tens_out;

    always @(*) begin
        if(gameover_s == 1)begin
            out = score_ones;
            tens_out = score_tens;
        end
        else begin
            out = sum;
            tens_out = 4'd0;
        end
    end
endmodule
```

Figure 48. Mux 3 to 2 Code