Lab 2: Game Access Control on FPGA

Thomas Vo
2179861

ECE 5440

Introduction

This game is a binary arithmetic game, where players toggle switches to reach the target sum of 1111 (binary 15).

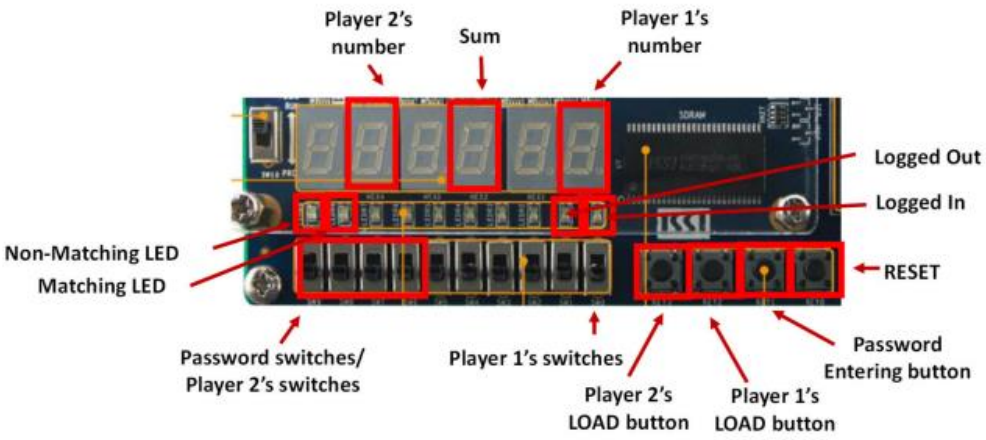


Figure 1. Diagram for Controls and Displays

As shown above in Figure 1, the 7-segment displays show the number entered by player 1, player 2, and the sum of the two numbers entered. The LEDs on the left show if the numbers entered match to 1111 and if the numbers entered do not match to 1111. The LEDs on the right show if the user is logged out or logged in. Player 1's switches are the 4 left-most switches and player 2's switches are the 4 right-most switches. The switches used to enter the password are also player 2's switches. The 4 buttons are used to load player 2's switches to the display, load player 1's switches to the display, load the (4) four password digits to the display , and to reset the status of the game.

The game starts with the player entering the password, which requires entering a password. If it is correct , the game is logged in. If not, the game remains logged in and will not be able to play the game until a successful password attempt. Then the game starts with player 1 choosing a number, and without player 2 seeing player 1's switches, player 2 must correctly input the number required to add up to 1111. If player 2 inputs correctly, they get a point, and if not, player 1 gets a point. This process is repeated and player 2 will start the next round. The winner will have the most points.

If the user wants to log out, they can do so by pressing the password entering button again. This will log them out, as shown on the LED. If they want to log back in, they should press the password entering button once more, and then enter the password to log back in.

If the user wishes to reset the game at any time, they can press the reset button which will reset all progress and take the user back into the log in phase.

System Architecture Design

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

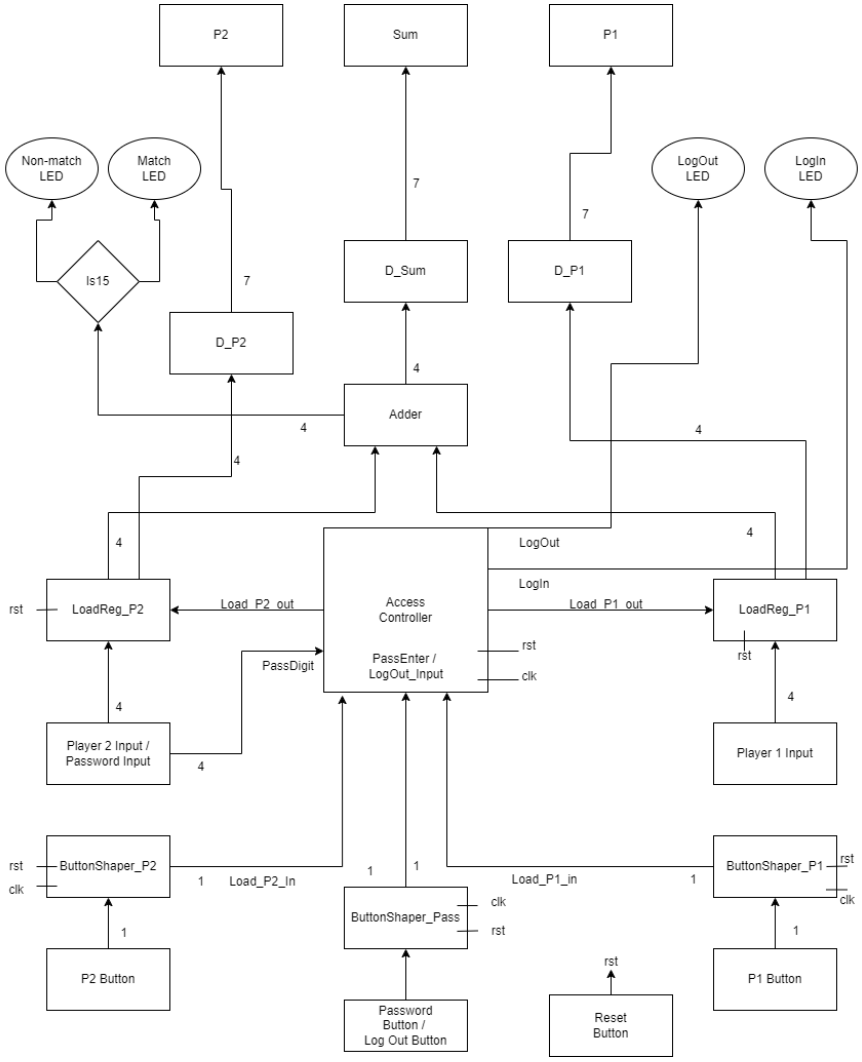


Figure 2. Top-level System Architectural Design

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.

Access Controller

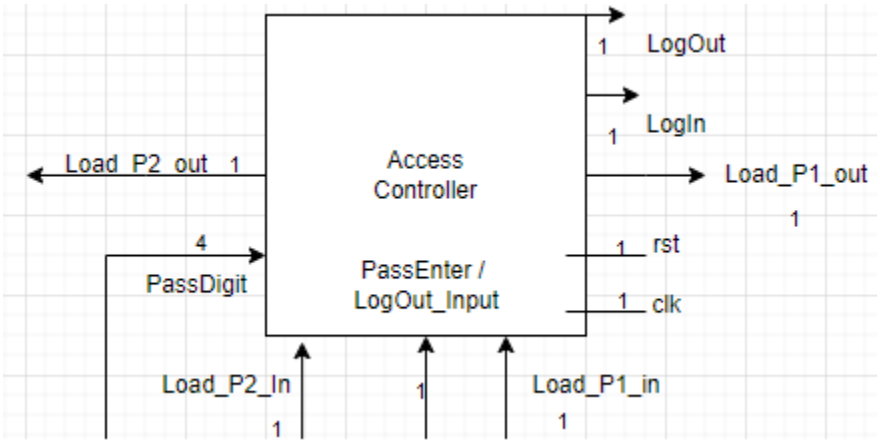


Figure 3. Access Controller

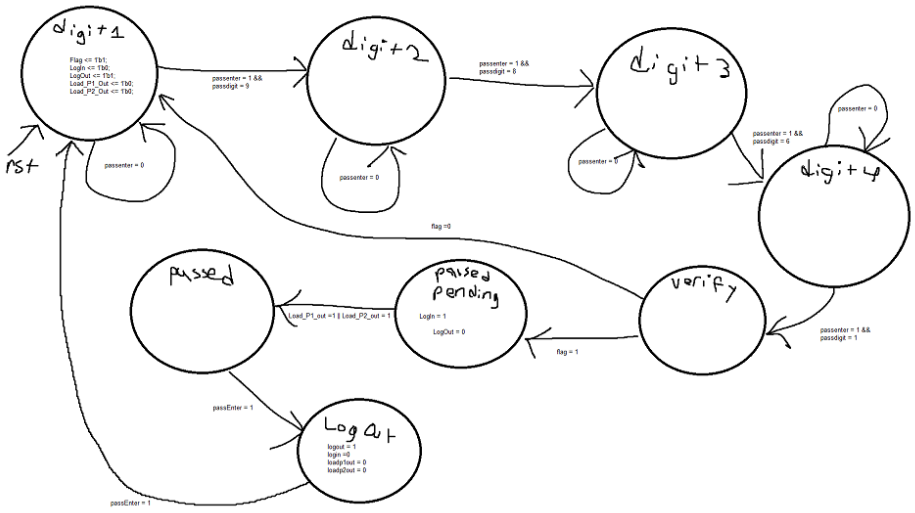


Figure 4. Finite State Machine for Access Controller

The Access module manages user authentication by processing a four-digit passcode and controlling system access. It transitions through multiple states to verify the entered passcode, granting access if the correct sequence (9-8-6-1) is entered. The module outputs LogIn and LogOut signals to indicate access status and controls Load_P1_Out and Load_P2_Out based on input signals. If an incorrect passcode is entered, the module resets to the initial state. The input signals are PassDigit, PassEnter, Load_P1_In, Load_P2_In, clk, rst, and LogOut_Input, while the output signals are LogIn, LogOut, Load_P1_Out, and Load_P2_Out.

Button Shaper

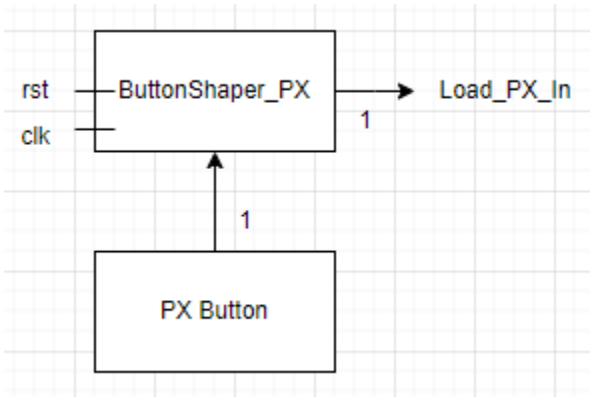


Figure 5. Button Shaper

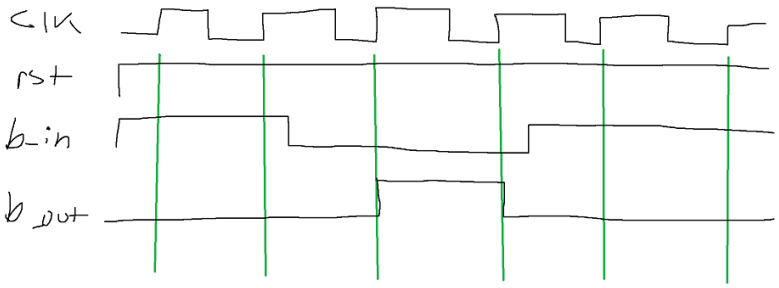


Figure 6. Expected Waveform for Button Shaper

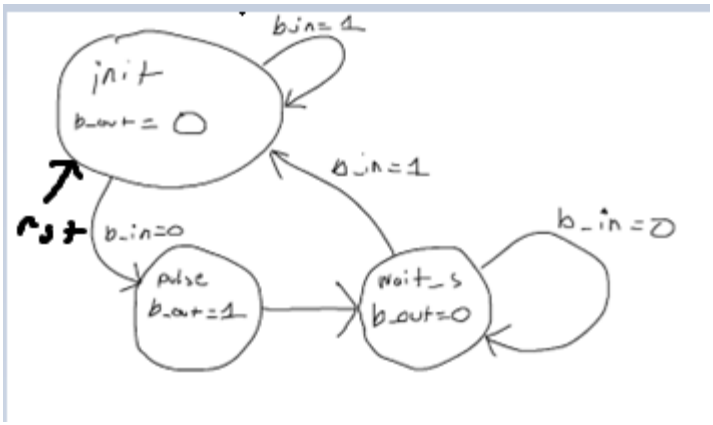


Figure 7. Finite State Machine drawing for Button Shaper

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

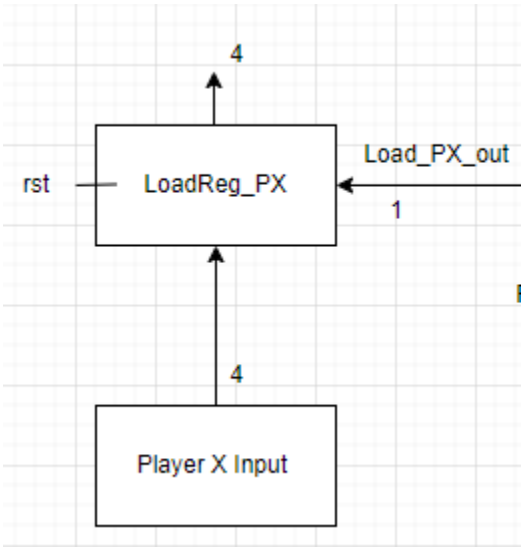


Figure 8. Load Register

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

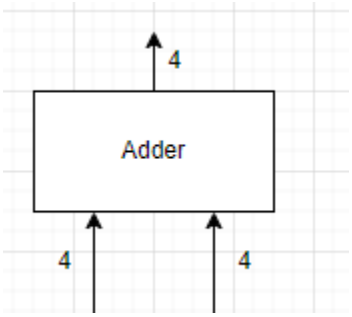


Figure 9. Adder Module

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

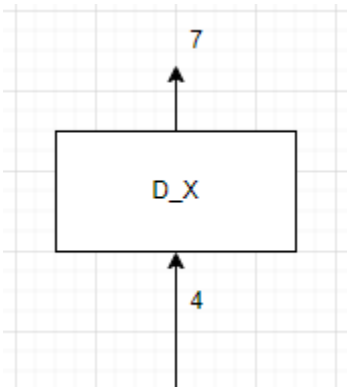


Figure 10. Seven Segment Decoder

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

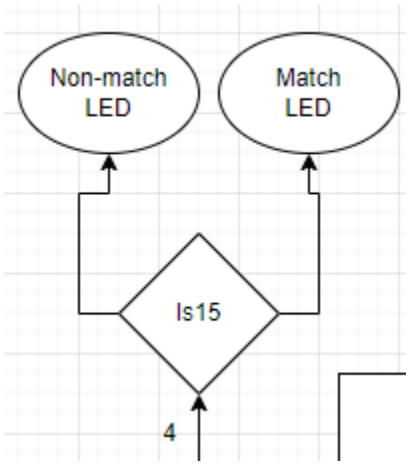


Figure 11. Is15 Verification Module

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

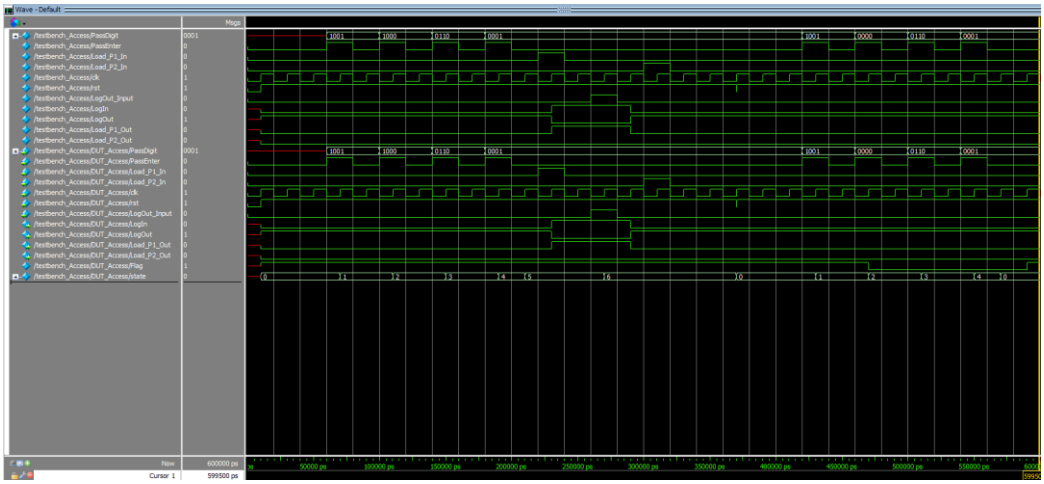


Figure 12. Simulation for Access Controller

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.

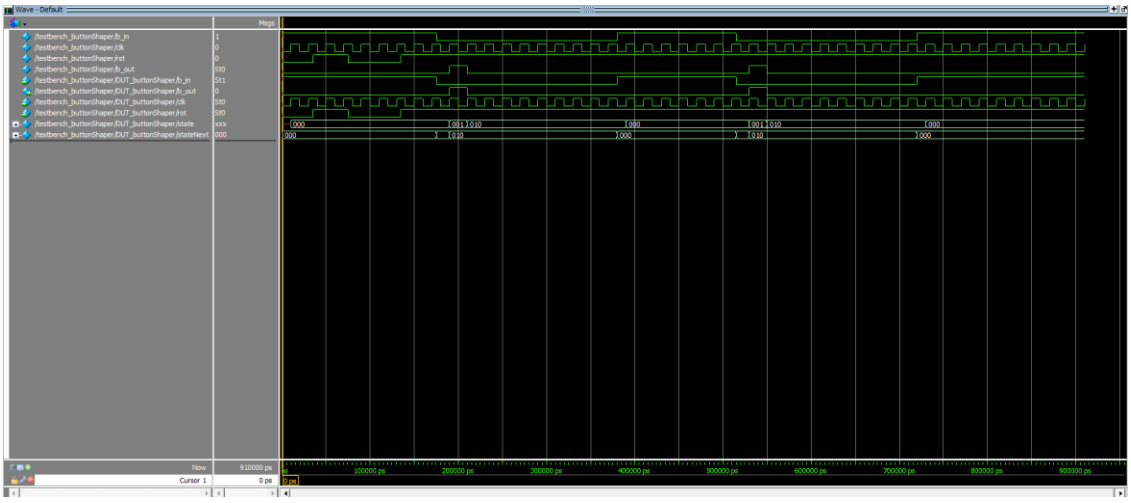


Figure 13. Simulation for Button Shaper

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.

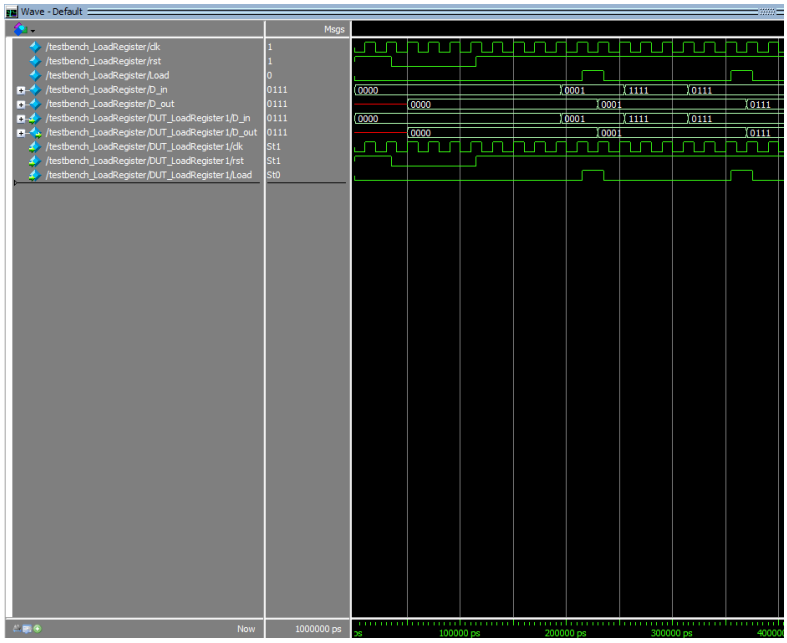


Figure 14. Load Register Simulation

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.

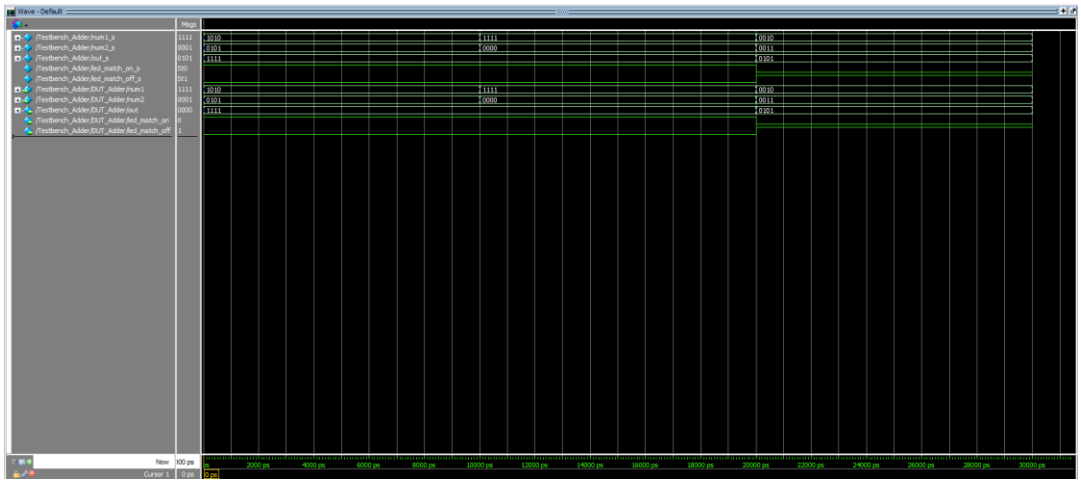


Figure 15. Adder Simulation

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.

Figure 16. 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.

FPGA Board Testing Results

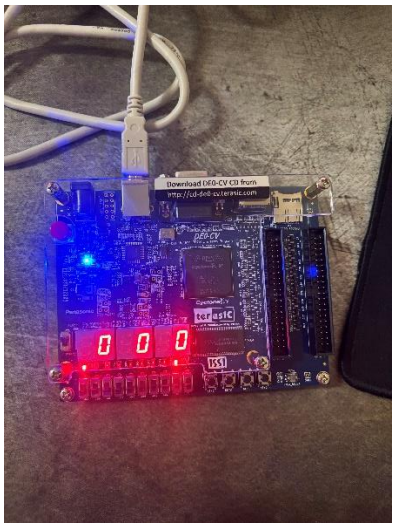


Figure 17. Default State / Reset State

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

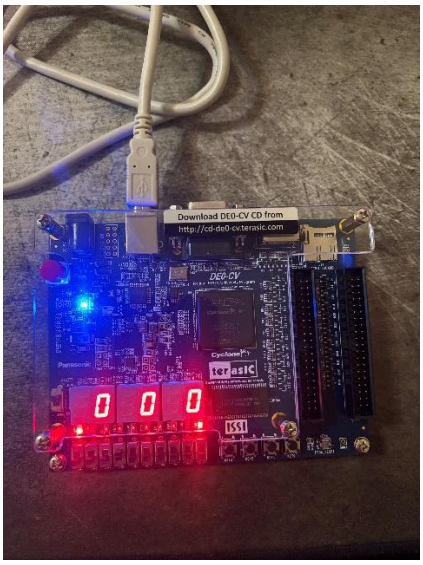


Figure 18. Logged In State

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

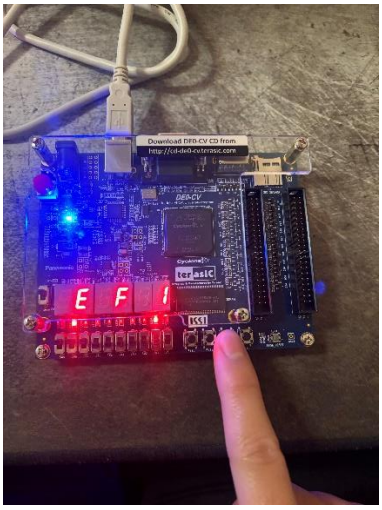


Figure 19. Log Out After Playing

After a game has been played, the user logs out by pressing the password entry button. This logs the user out, as indicated by the LED on the right.

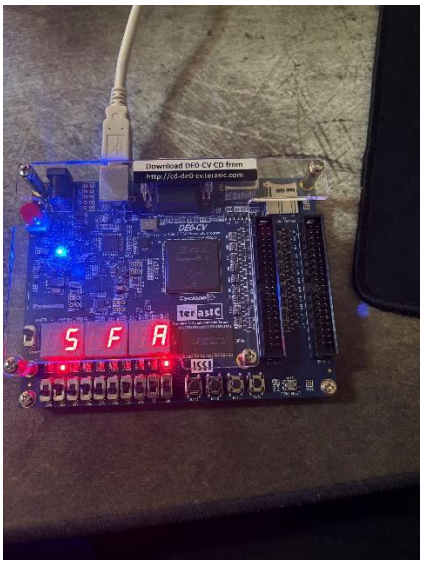


Figure 20. Matching Case

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.

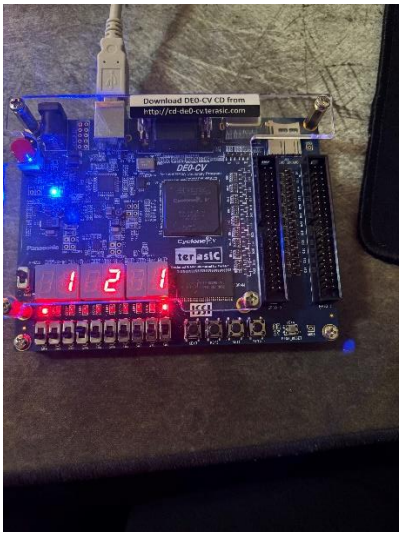


Figure 21. Non-matching Case

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.

Video Demo

gameplay and load reg.MOV

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

reset.MOV
This shows how the reset button operates.

log out and re login.MOV

This shows the log out feature and logging back in.

password entry to log in.MOV

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

Conclusion

I have successfully built and completed lab 2, 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 log-out feature.

Appendix

```
// ECE 5440
// Thomas Vo 9861
// Top Level module for Lab 2

module Lab2_VO_Thomas (
    p1_in, p2_in, clk, rst,
    p1_button, p2_button,  pass_LogOut_button,
    p1_out, p2_out, sum_out,
    sum_led_nonMatch, sum_led_Match,
    LogOut_LED, LogIn_LED
);

input [3:0] p1_in, p2_in;
input p1_button, p2_button,  pass_LogOut_button, clk, rst;
output [6:0] p1_out, p2_out, sum_out;
output sum_led_nonMatch, sum_led_Match,LogOut_LED, LogIn_LED;

//configure buttons first **NOT reset
wire p1_To_Access, p2_To_Access, pass_To_Access, LogOut_To_Access;
buttonShaper p1buttonShaper(p1_button, p1_To_Access, clk, rst);
buttonShaper p2buttonShaper(p2_button, p2_To_Access, clk, rst);
buttonShaper pass(pass_LogOut_button, pass_To_Access, clk, rst);
buttonShaper LogOut(pass_LogOut_button, LogOut_To_Access, clk, rst);
//configure player inputs to Load Registers
wire [3:0] loadReg_p1_out, loadReg_p2_out;
wire Load_P1_out, Load_P2_out;
LoadRegister p1loadReg(p1_in, loadReg_p1_out, clk, rst, Load_P1_out);
LoadRegister p2loadReg(p2_in, loadReg_p2_out, clk, rst, Load_P2_out);
//connect loadRegs to Adder and Decoders
wire [3:0] AddertoIs15, AddertoDecoder;
Adder myAdder(loadReg_p1_out, loadReg_p2_out, AddertoDecoder, AddertoIs15);
sevenSegDecoder p1SSD(loadReg_p1_out, p1_out);
sevenSegDecoder p2SSD(loadReg_p2_out, p2_out);
sevenSegDecoder adder(AddertoDecoder, sum_out);
// Is15 module
Is15 verify(AddertoDecoder, sum_led_Match, sum_led_nonMatch);
//Access Controller
Access myAC(p2_in, pass_To_Access, p1_To_Access, p2_To_Access,
LogIn_LED, LogOut_LED, Load_P1_out, Load_P2_out, rst, clk, pass_LogOut_button);

endmodule
```

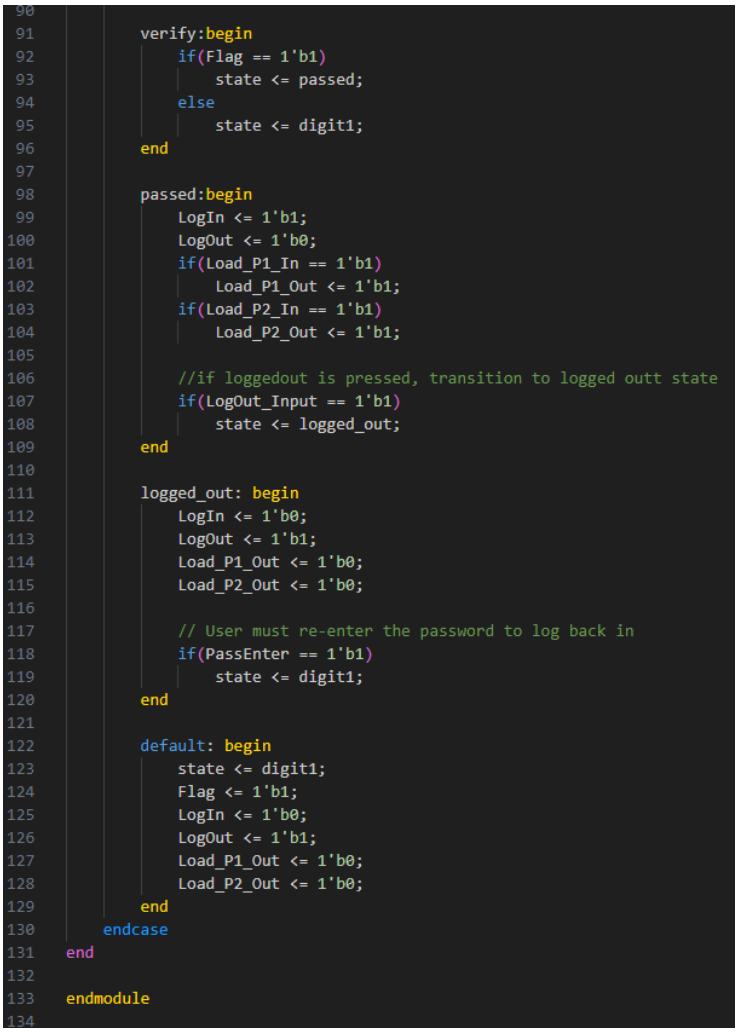
Figure 22. Top Level Module Code

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

module Access (
    PassDigit, PassEnter, Load_P1_In,
    Load_P2_In, LogIn, LogOut,
    Load_P1_Out, Load_P2_Out,
    rst, clk, LogOut_Input
);

input [3:0] PassDigit;
input PassEnter, Load_P1_In, Load_P2_In, clk, rst, LogOut_Input;
output LogIn, LogOut, Load_P1_Out, Load_P2_Out;
reg LogIn, LogOut, Load_P1_Out, Load_P2_Out, Flag;

parameter digit1 = 0, digit2 = 1, digit3 = 2, digit4 = 3, verify = 4, passed = 5, logged_out = 6;
reg [2:0] state;

always @(posedge clk) begin
    if(rst == 1'b0)begin
        state <= digit1;
        Flag <= 1'b1;
        LogIn <= 1'b0;
        LogOut <= 1'b1;
        Load_P1_Out <= 1'b0;
        Load_P2_Out <= 1'b0;
    end

    else
    case (state)
        digit1:begin
            Flag <= 1'b1;
            LogIn <= 1'b0;
            LogOut <= 1'b1;
            Load_P1_Out <= 1'b0;
            Load_P2_Out <= 1'b0;

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

        digit2:begin
```

Figure 23. Access Controller Code 1/3

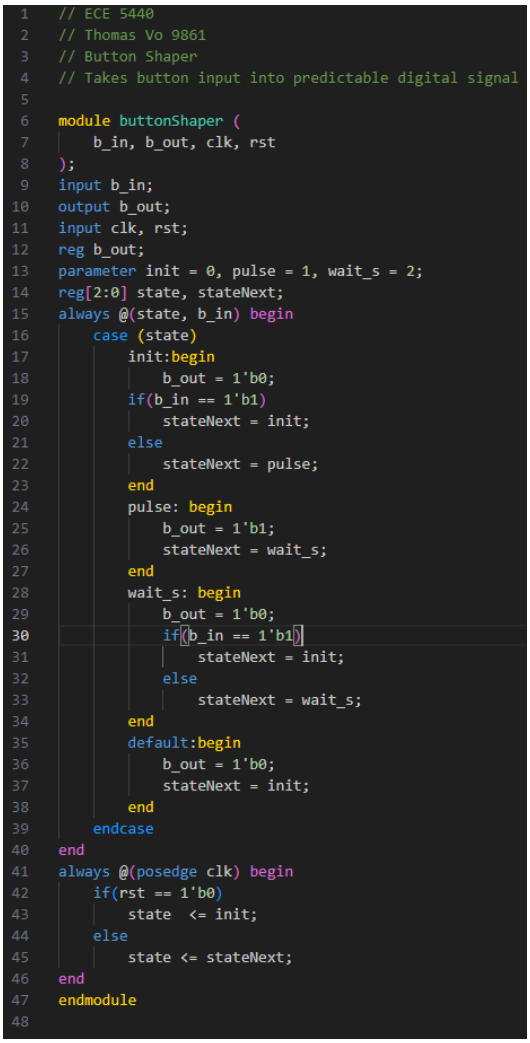
```
        digit2:begin
            LogIn <= 1'b0;
            LogOut <= 1'b1;
            Load_P1_Out <= 1'b0;
            Load_P2_Out <= 1'b0;
            if(PassEnter == 1'b1)begin
                if(PassDigit != 4'b1000) //incorrect - 8
                    Flag <= 1'b0;
                state <= digit3;
            end
            else
                state <= digit2;
        end

        digit3:begin
            LogIn <= 1'b0;
            LogOut <= 1'b1;
            Load_P1_Out <= 1'b0;
            Load_P2_Out <= 1'b0;
            if(PassEnter == 1'b1)begin
                if(PassDigit != 4'b0110) //incorrect - 6
                    Flag <= 1'b0;
                state <= digit4;
            end
            else
                state <= digit3;
        end

        digit4:begin
            LogIn <= 1'b0;
            LogOut <= 1'b1;
            Load_P1_Out <= 1'b0;
            Load_P2_Out <= 1'b0;
            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 <= passed;
```

Figure 24. Access Controller Code 2/3



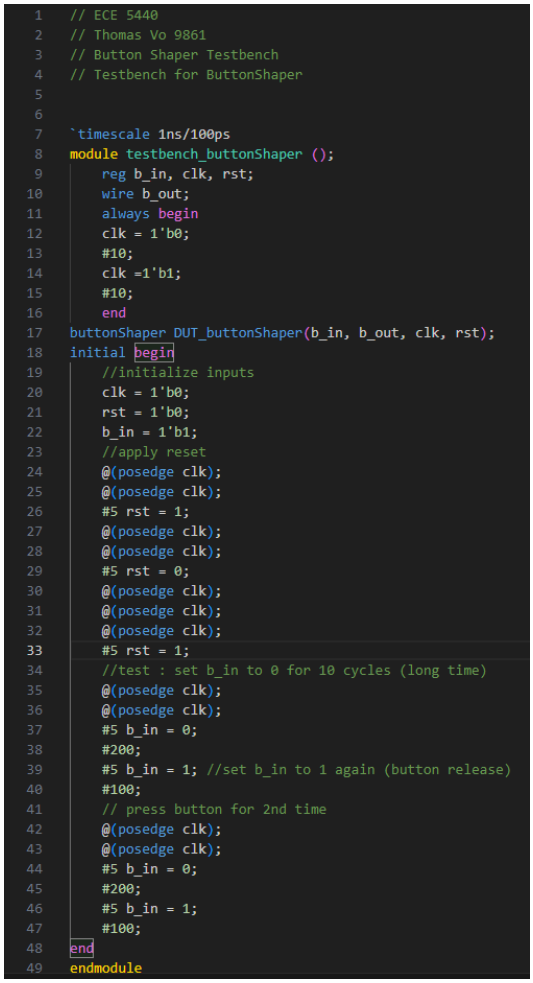
```
90
91          verify:begin
92              if(Flag == 1'b1)
93                  state <= passed;
94              else
95                  state <= digit1;
96          end
97
98          passed:begin
99              LogIn <= 1'b1;
100             LogOut <= 1'b0;
101             if(Load_P1_In == 1'b1)
102                 Load_P1_Out <= 1'b1;
103             if(Load_P2_In == 1'b1)
104                 Load_P2_Out <= 1'b1;
105
106             //if loggedout is pressed, transition to logged outt state
107             if(LogOut_Input == 1'b1)
108                 state <= logged_out;
109         end
110
111         logged_out: begin
112             LogIn <= 1'b0;
113             LogOut <= 1'b1;
114             Load_P1_Out <= 1'b0;
115             Load_P2_Out <= 1'b0;
116
117             // User must re-enter the password to log back in
118             if(PassEnter == 1'b1)
119                 state <= digit1;
120         end
121
122         default: begin
123             state <= digit1;
124             Flag <= 1'b1;
125             LogIn <= 1'b0;
126             LogOut <= 1'b1;
127             Load_P1_Out <= 1'b0;
128             Load_P2_Out <= 1'b0;
129         end
130     endcase
131 end
132
133 endmodule
134
```

Figure 25. Access Controller Code 3/3



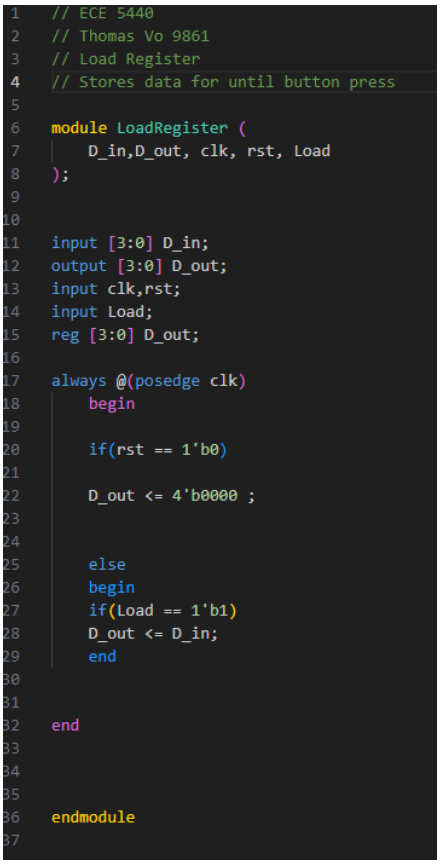
```
// ECE 5440
// Thomas Vo 9861
// Button Shaper
// Takes button input into predictable digital signal

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 26. Button Shaper Code



```
// 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 27. 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 28. Load Register Code

```
// 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 29. Load Register Testbench Code 1/2

```
        // 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 30. Load Register Testbench Code 2/2

```
//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 31. Is15 Verification Module Code

```
//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
```

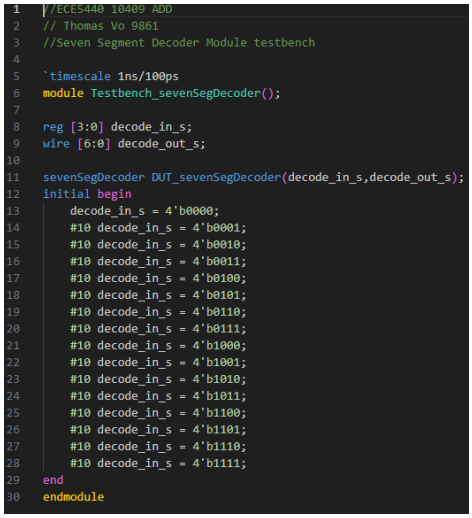
Figure 32. Adder Module Code

```
//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 33. Adder Testbench Code

```
//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 34. Code for sevenSegDecoder Module



```
//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 35. Seven Segment Decoder Testbench Code