Lab 2: FPFA-based Mental Binary Math Game

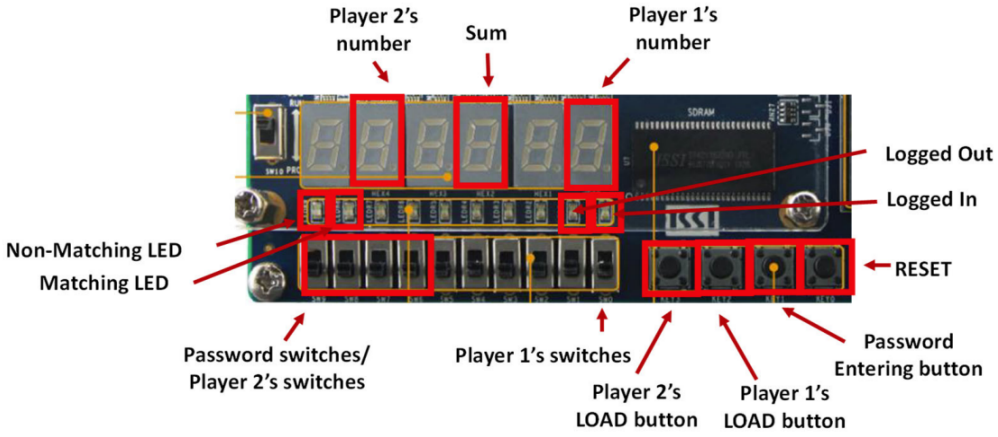
02.24.2023

—

Smithson Arrey
Electrical Engineering, Computer and Embedded Systems
Dr. Yuhua Chen
ECE 5440, Advanced Digital Design, Spring 2023

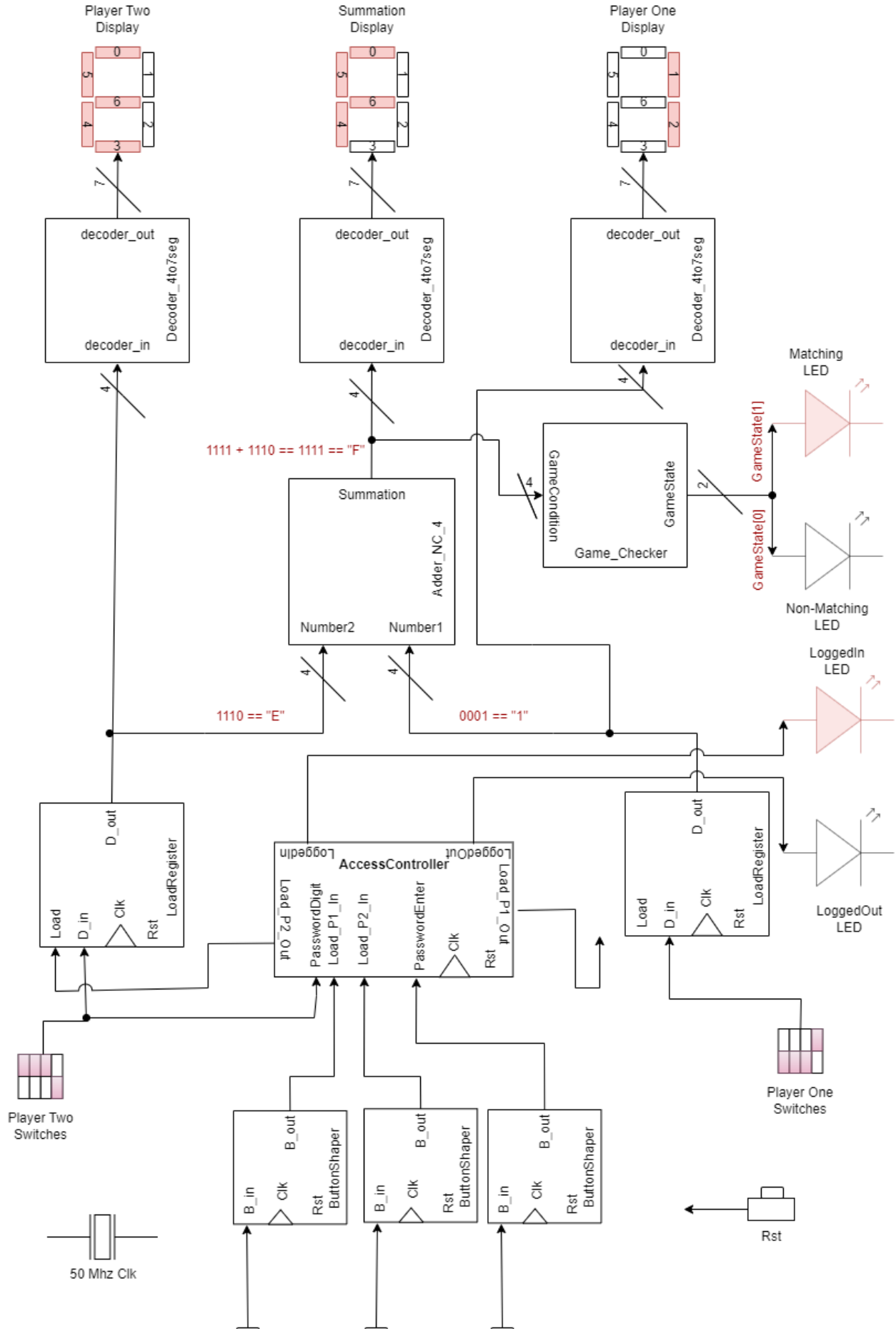
Introduction

This FPGA-based Mental Binary Math Game is a game that tests the two player's knowledge of binary. Each player is given a set of four switches that represent the bits of a hexadecimal number. By switching a switch to its active position, the players can indicate that a bit in their series of bits is active. For example, a player that has all but their 1st and 3rd switches in the inactive state will be representing the binary sequence, 1010, which is the hexadecimal number A. The goal of the game is for both players to add their numbers together to reach a total of 1111 or "F". Each player will be able to see the numbers that their inputs are creating on one of the 7 segment displays on either end of the board. In between the summation of their numbers will be shown. If the players are successful, they should see "F" appear on the summation screen and the leftmost red LED will turn on to confirm their victory. The rightmost LED, signaling an incorrect match, will turn off in that case. Otherwise, the summation screen will show another number and the rightmost LED will be active and the leftmost will be inactive. Note: Summation is limited to four bits. See the image below for visual details. Before the players begin playing, they must log in using a unique pin by the game's creator. After they are done playing, they may log out using the same password entry button. Their progress will not be erased unless the reset button is pressed.



Game User Interface

System Architecture Design


Player Two Display
Summation Display
Player One Display
decoder_out
decoder_in
Decoder_4to7seg
1111 + 1110 == 1111 == "F"
Summation
Adder_NC_4
Number2
Number1
GameCondition
GameState
Game_Checker
Matching LED
Non-Matching LED
GameState[1]
GameState[0]
LoggedIn LED
LoggedOut LED
1110 == "E"
0001 == "1"
D_out
Load
D_in
Clk
Rst
LoadRegister
AccessController
LoggedIn
LoggedOut
Load_P2_Out
PasswordDigit
Load_P1_In
Load_P2_In
PasswordEnter
Load_P1_Out
Player Two Switches
Player One Switches
B_out
B_in
ButtonShaper
50 Mhz Clk
Rst

Lab 2 System Architecture

The figure above shows the system architecture for this game. There are three types of modules in this design:

1. Decoder_4to7_seg - This modules decodes a 4-bit binary input, decoder_in, representing a hexadecimal number to a 7-bit binary code, decoder_out, that maps to the pins of a 7 segment display such that the display visualizes the alphanumeric representation of the hexadecimal. The 7 seg display has active low logic.
2. Adder_NC_4 - This module is a generic 2 input, 4-bit adder without a carry bit in its output. In this case, the two inputs are defined as the four bit binary input signal from either player's switches.
3. Game_Checker - This module is a generic game win checker that can interpret a 4-bit game condition and output a 2-bit response indicating the game's state. Bit 0 of the response indicates a non-valid condition. Bit 1 of the response indicates a valid condition. Only one bit will be activated at a time. The use of this module in Lab 1 will be to use the 4-bit sum of both player's numbers as this module's input. The output will be wired to the verification LEDs.
4. ButtonShaper - This module is a generic button shaper. It transforms multi clock cycle input pulses into a single cycle pulse
5. LoadRegister - This module is a generic load register. It shifts the input to the output on command. Acts as a simple memory device.
6. AccessController - Prevents the user from accessing the game without first logging in. Uses sequential logic to input a predetermined password. Can be used to log in and out of the game.

7 Segment Decoder Simulation

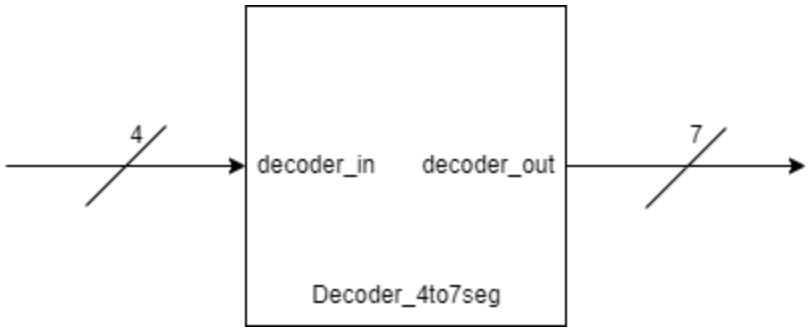
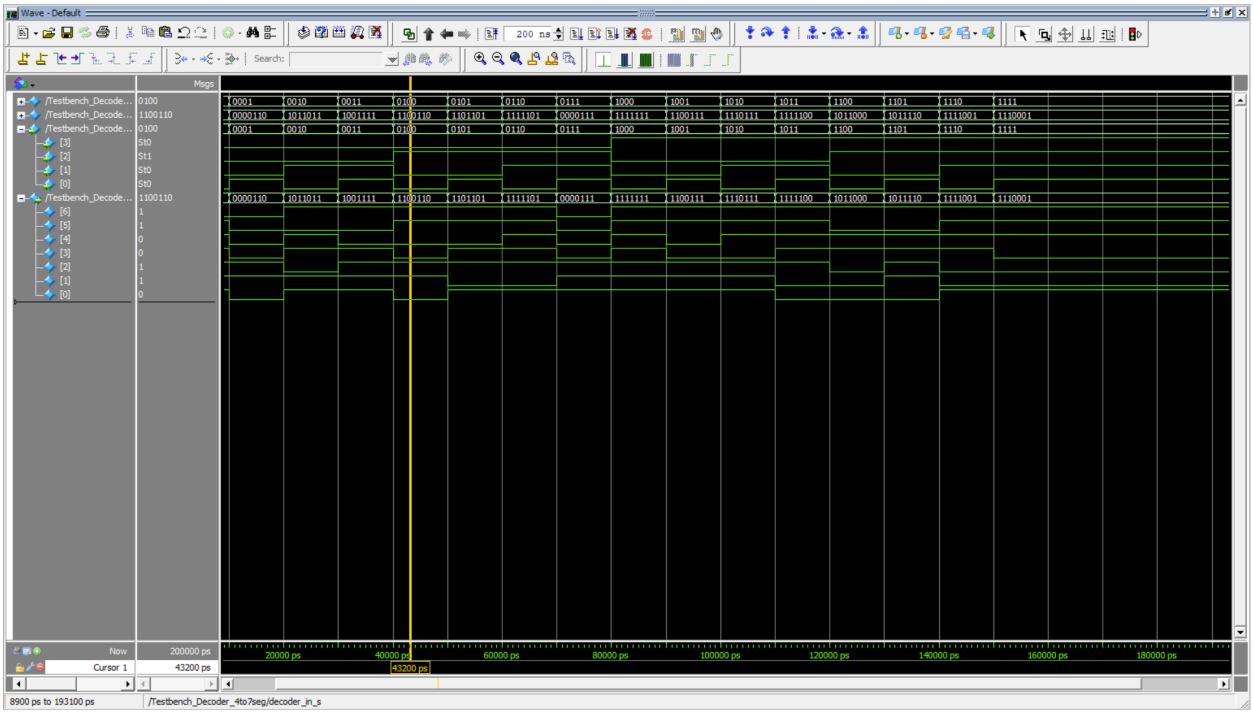


Diagram of 7 Segment Decoder Module

The 7 Segment Decoder module necessary for Lab 1 takes 1 four bit input and outputs a 7 bit output. In this instance, the decoder will generate an output, decoder_out,

that correlates to the visual representation of the hexadecimal equivalent of the four bit input, decoder_in, rather than its direct representation in binary. The mapping is defined by the pin assignments driving the 7 Segment Display. The input signal will be driven by a player's slide switch used to indicate a 4 bit binary number or the summation of two player's binary numbers. The output signal will connect to the 7 segment display.



7 Segment Decoder Test Waveforms

For my test of the decoder, I ran a simulation of every input case. The source code for this test can be found in the appendix of this document. For example, on the fourth test run, indicated by the yellow line, the number four is inputed as a binary signal and the output is given as 7'b1100110. That output would correctly draw the hexadecimal number 4 on the display. See the appendix, for a visualization of which pins are connected to which LED on the display. Likewise, the binary signal of number three is decoded as 7'b1001111 and the binary signal of 5 is decoded as 7'b1101101. However, it should be noted that the bits of the output will be inverted in the production code because the 7 Segment Display has an active low logic configuration.

Adder Simulation

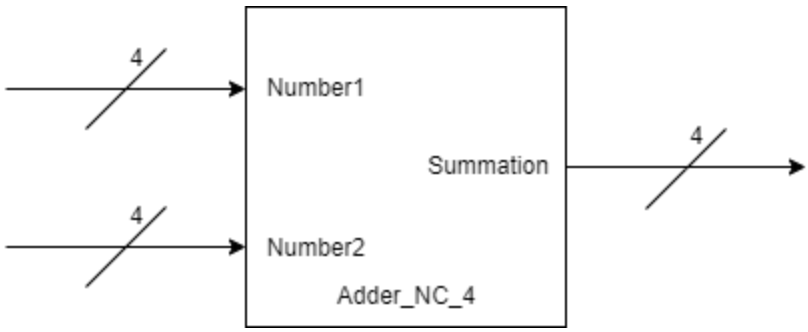
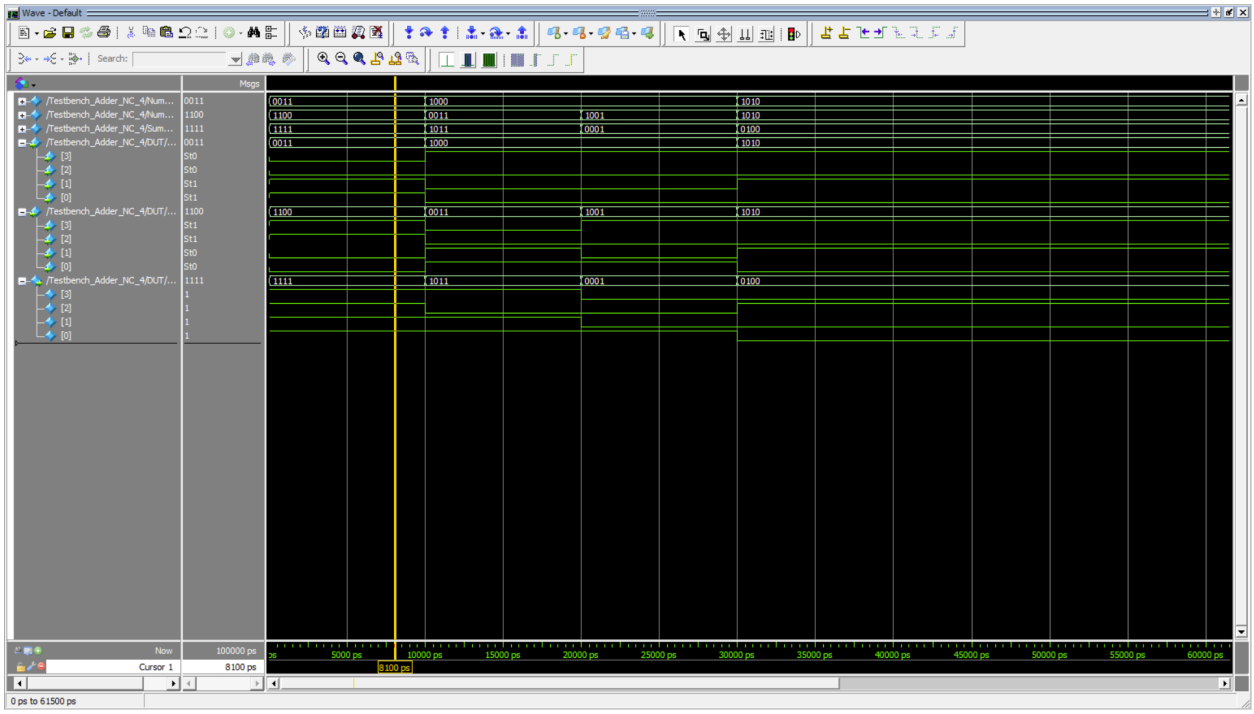


Diagram of Adder Module

The Adder module necessary for Lab 1 takes 2 four bit inputs and outputs a four bit summation of the two inputs. No carry bit is necessary for this lab so it will be excluded from the design. The two inputs are generically defined as Number1 and Number2 can therefore be correlated to the corresponding player number in the lab. These inputs are generated from the player's use of the slide switches to indicate a 4 bit binary number. The 4 bit Summation output is the sum of Number1 and Number2. This output will then be used as an input signal for the 7 segment decoder linked to the summation display.



Adder Test Waveforms

For my test of the adder, I ran a simulation of a matching number, non-matching number, and two overflow cases. The source code for this test can be found in the appendix of this document. For example, a matching case should have an output of 4'b1111. The first test shown correctly displays this output. Likewise, a non matching case will be any number that is not equivalent to 4'b1111. That means that the second test correctly displays a non-matching case. Thirdly, the remaining cases show that a summation that overflows will not produce a correct matching case. However it would be hard to know that this was an overflow from looking at the output alone.

Button Shaper Simulation

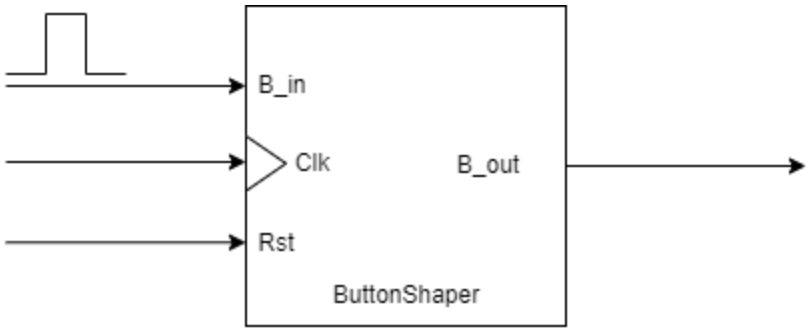
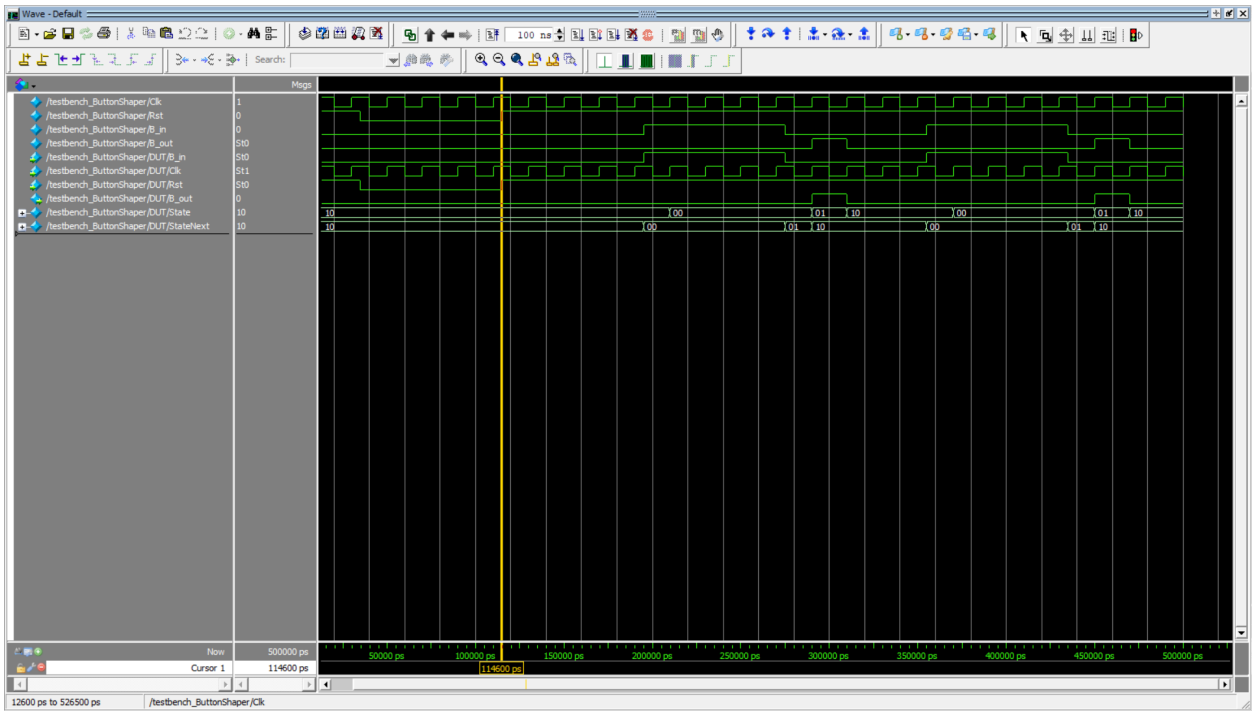


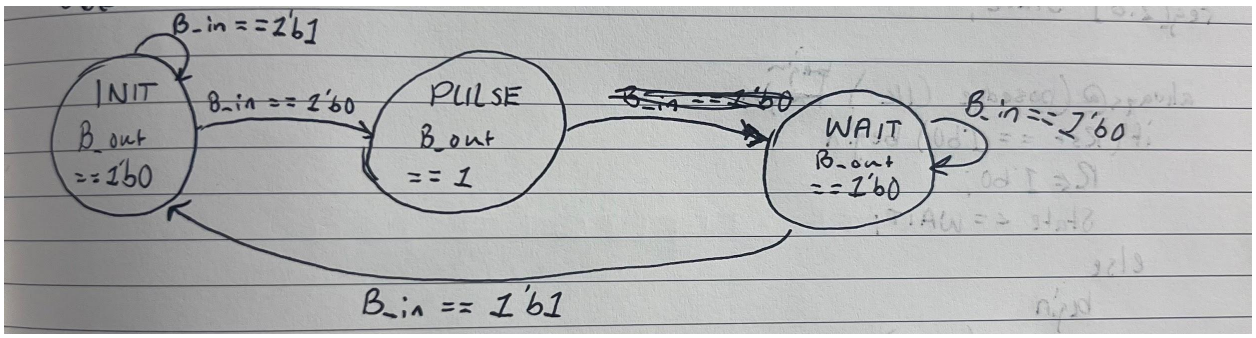
Diagram of Button Shaper Module

The Button Shaper module necessary for Lab 2 takes one single bit button press signal, B_in, 1 clock signal, Clk, and 1 Reset signal, Rst, as inputs. The output is a single cycle pulse signal, B_out. This module uses both combination and sequential logic to transform the multi-clock cycle input of B_in into a single cycle pulse. This is accomplished using a two-procedure finite state machine implementation.



Button Shaper Test Waveforms

For my test of the Button Shaper, I ran a simulation of two multicycle button presses to ensure the system is functional. The image above shows the waveforms generated during my simulation. The period before the vertical yellow line indicates the "start-up" phase where Rst == 1'b0. The system is intentionally unresponsive during this stage. When Rst == 1'b1, the system we are interested in is active. For this system, we expect a single cycle pulse to appear in the output after a button is pressed and released no matter how long the button is pressed. The period after the vertical yellow line shows that this requirement has been met. Initially, the button isn't pressed so no output can be given. Then the button is pressed, held for a while, and then released. Resulting in a single cycle pulse shortly afterwards. The second occurrence of this scenario shows that this system is repeatable and doesn't get stuck in any state.



Finite State Machine for Button Shaper

Access Control Simulation

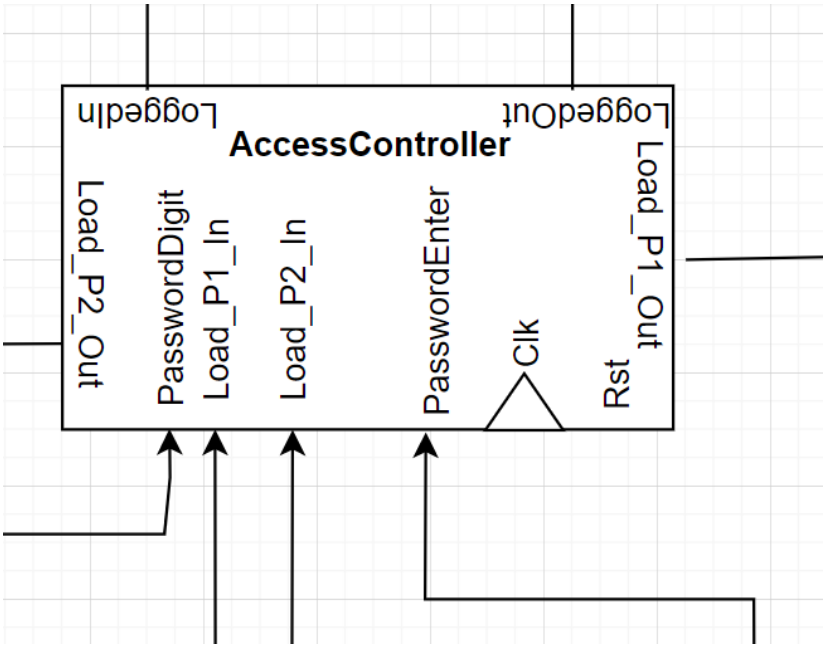
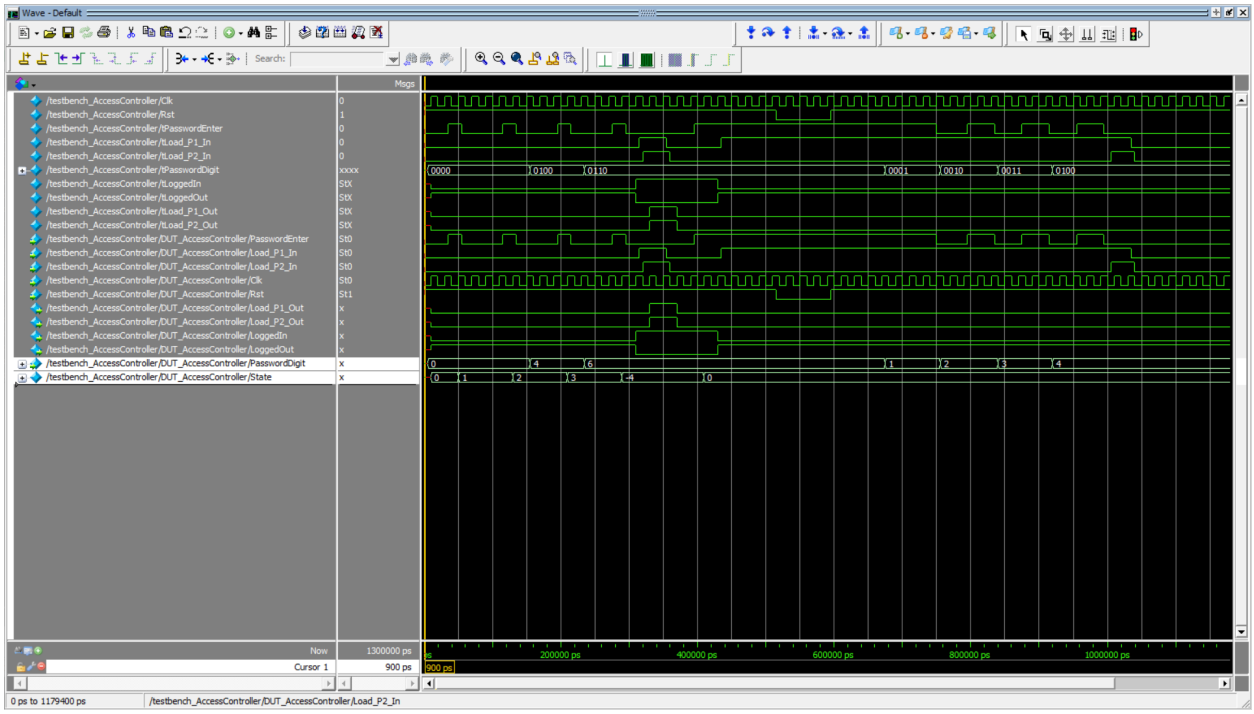


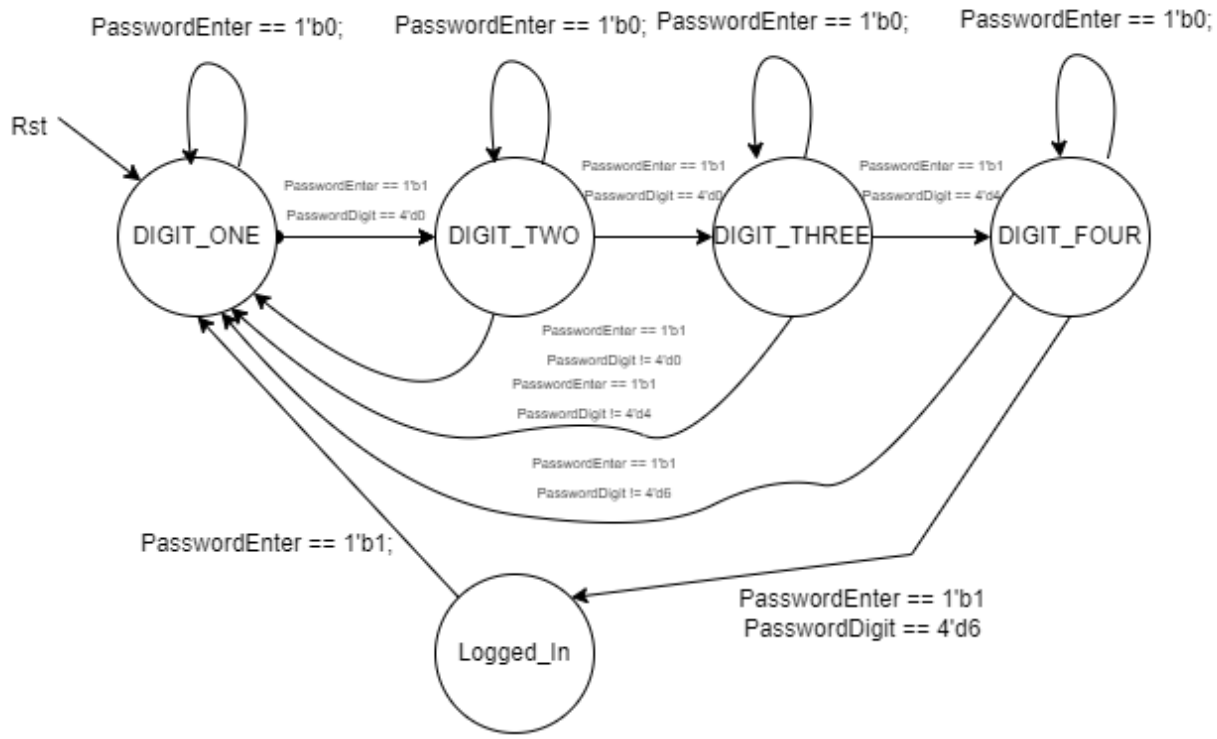
Diagram of Access Control Module

The Access Control module necessary for Lab 2 takes a 4 bit PasswordDigit input, one Load signal input per player, 1 clock signal, Clk, and 1 Reset signal, Rst, as inputs. The outputs are the two replicated load signals, and the two signals indicating whether the players have logged in or not. Those signals will be connected directly to internal LEDs. This module uses both combination and sequential logic. This is accomplished using a one-procedure finite state machine implementation.



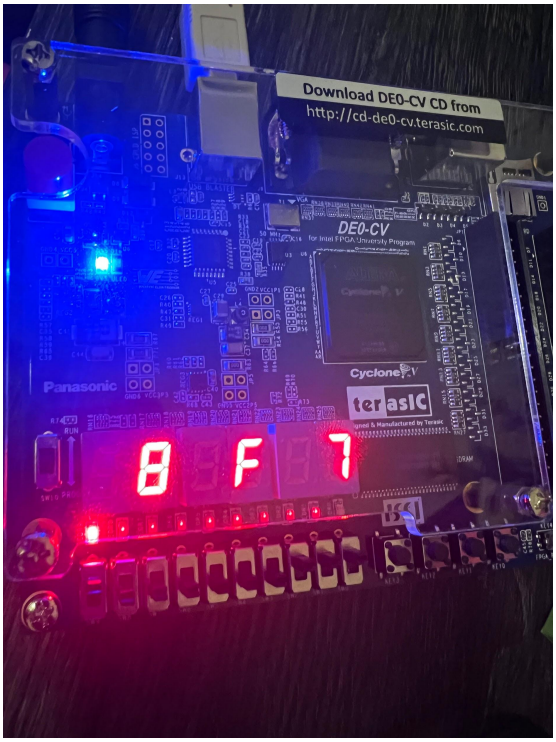
Access Control Module Waveforms

For my test of the Access Control Module I checked the state machine by entering in the correct password, clearing that, and then entering an incorrect password. In the waveform above, it can be seen that the FPGA successfully travels the states of the state machine and logs in. Likewise, when entering any wrong input, the FPGA returns back to the original state to wait for a correct input.



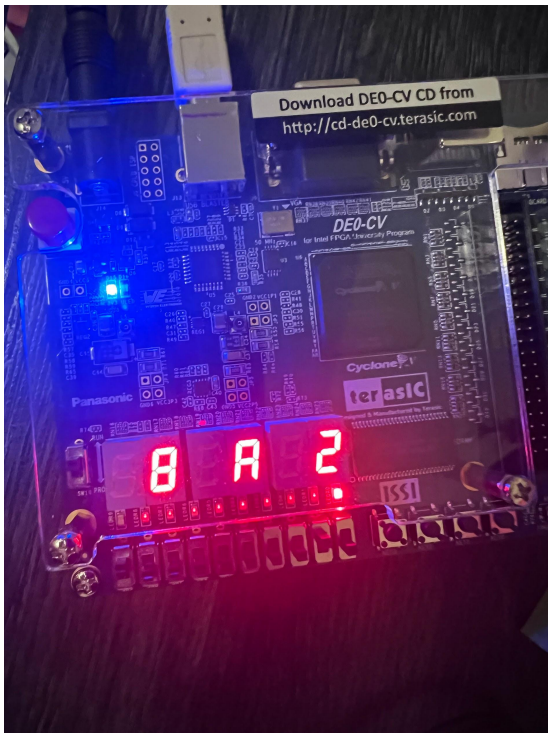
Finite State Machine for Access Control Module

FPGA Board Testing Results



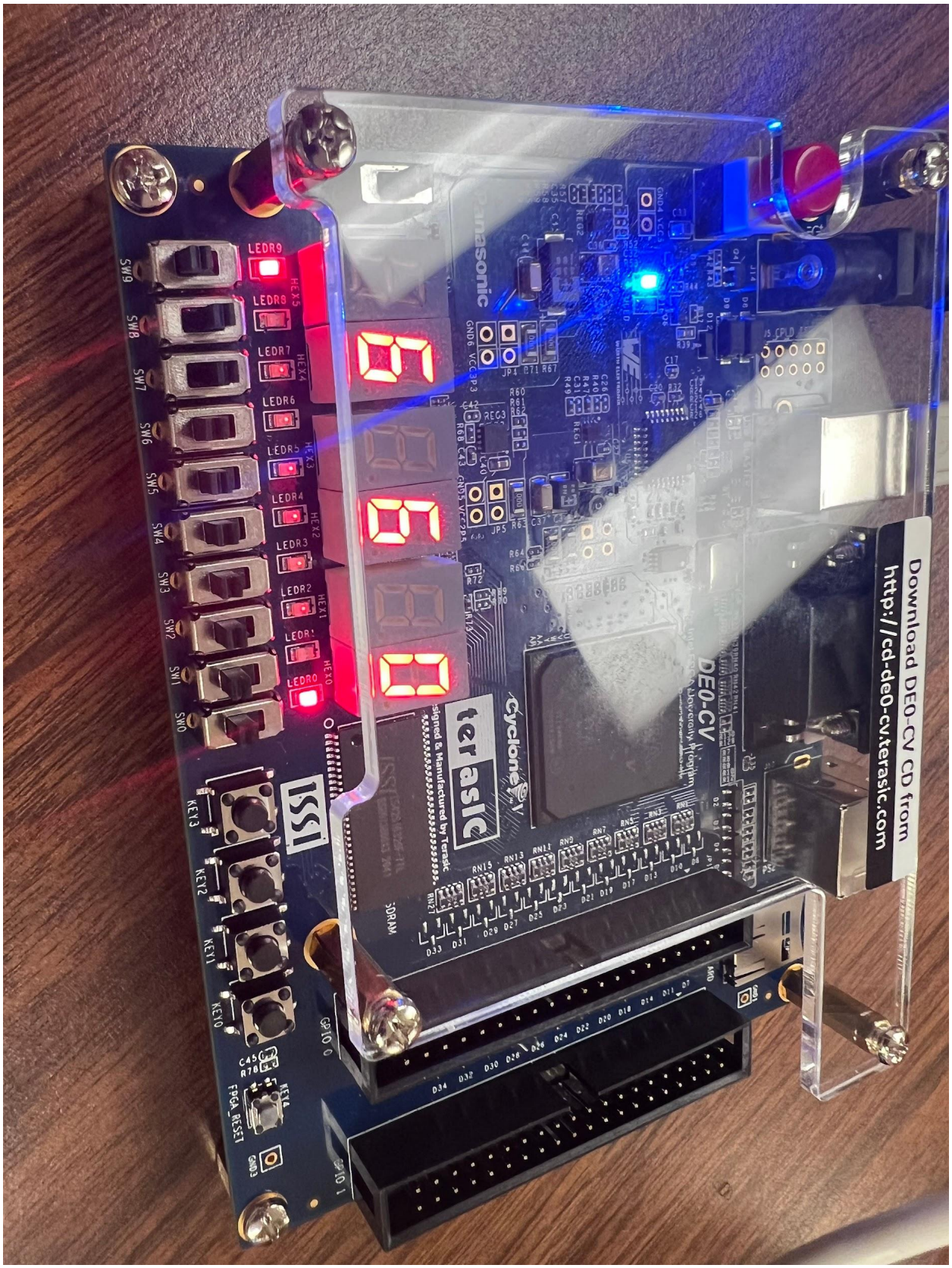
Matching Case Demonstrated on FPGA

The demonstration above showcases a matching or “game-winning” case because the middle 7 segment display showcases the hexadecimal “F” and the matching LED is active while the non-matching LED is inactive. It can be seen that player two activated their most significant bit to represent the number 8. Likewise, player one activated every bit but their most significant bit to represent the number 7. And 8 + 7 == “F” in hexadecimal so this confirms the validity of the demonstration.



Non-Matching Case Demonstrated on FPGA

The demonstration above showcases a non-matching or “game-losing” case because the middle 7 segment display showcases the hexadecimal “A” rather than “F”. Additionally, the matching LED is inactive while the non-matching LED is active. It can be seen that player two activated their most significant bit to represent the number 8. Likewise, player one only activated their second least significant bit to represent the number 2. And 8 + 2 == “A” in hexadecimal so this confirms the validity of the demonstration.


Download DE0-CV CD from
http://cd-de0-cv.terasic.com
DE0-CV
terasic
Cyclone V
LEDR9
LEDR8
LEDR7
LEDR6
LEDR5
LEDR4
LEDR3
LEDR2
LEDR1
LEDR0
SW9
SW8
SW7
SW6
SW5
SW4
SW3
SW2
SW1
SW0
KEY3
KEY2
KEY1
KEY0
KEY4
FPGA_RESET
HEX5
HEX4
HEX3
HEX2
HEX1
HEX0
GPIO 0
GPIO 1

User Interface Update 2-24-23

Video Demo

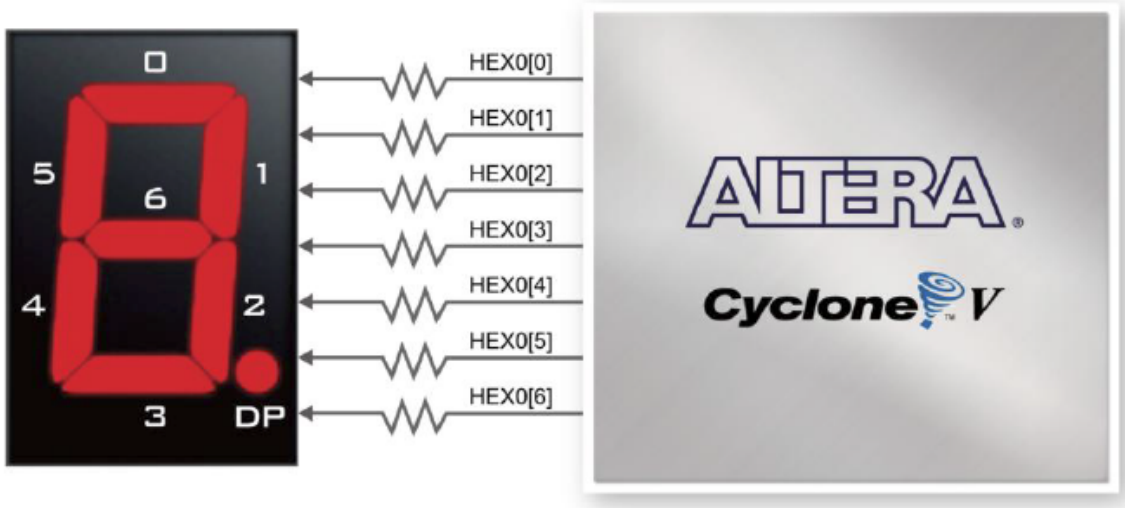
Link to Game Demo:
https://drive.google.com/file/d/1kf470GGztmG8lyOS5GGdO289C3EkZ-QW/view?usp=sharing

Link to Access Control and Loading Update Demo:
https://drive.google.com/file/d/1VgKYtuTyxryIWglkSf5pn5ktK65yZ_K_/view?usp=sharing

Conclusion

This week I was able to fully complete Lab 2 plus the bonus features. It was great to learn about how to build a fully functional system using FPGAs!

Appendix



7 Segment Display Pin Assignments

```
module Adder_NC_4(Number1, Number2, Summation);
    input [3:0] Number1, Number2;
    output [3:0] Summation;
    reg [3:0] Summation;

    always@(Number1, Number2)
        begin Summation = Number1 + Number2; end
endmodule
```

Adder Module Source Code

```
`timescale 1 ns/100 ps
module Testbench_Adder_NC_4();

    reg [3:0] Number1_s, Number2_s;
    wire [3:0] Summation_s;

    Adder_NC_4 DUT(Number1_s, Number2_s, Summation_s);

    initial
        begin
            // matching numbers
            Number1_s = 4'b0011; Number2_s = 4'b1100;
            #10;
            // non-matching numbers
            Number1_s = 4'b1000; Number2_s = 4'b0011;
            #10;
            Number1_s = 4'd8; // Equivalent to 4'b1000
            Number2_s = 4'd9; // Equivalent to 4'b1001
            #10;
            // Should overflow
            Number1_s = 4'd10; Number2_s = 4'd10;
        end
endmodule
```

Adder Testbench Source Code

```
// Course Number: ECE 5440
// Author: Smithson Arrey, 0046
// Module Name: Decoder_4to7seg
// Description: This modules decodes a 4-bit binary input representing a hexadecimal number to
//              an 7-bit binary code that maps to the pins of a 7 segment display
//              such that the display visualizes the alphanumeric
//              representation of the hexadecimal
//              The 7 seg display has active low logic
// Comments: N/A
module Decoder_4to7seg(decoder_in, decoder_out);
    input [3:0] decoder_in;
    output [6:0] decoder_out;
    reg [6:0] decoder_out;

    always@(decoder_in)
        begin
            case(decoder_in)
                4'b0000://
                    begin decoder_out = 7'b1000000; end
                4'b0001://
                    begin decoder_out = 7'b1111001; end
                4'b0010://
                    begin decoder_out = 7'b0100100; end
                4'b0011://
                    begin decoder_out = 7'b0110000; end
                4'b0100://
                    begin decoder_out = 7'b0011001; end
                4'b0101://
                    begin decoder_out = 7'b0010010; end
                4'b0110://
                    begin decoder_out = 7'b0000010; end
                4'b0111://
                    begin decoder_out = 7'b1111000; end
                4'b1000://
                    begin decoder_out = 7'b0000000; end
                4'b1001://
                    begin decoder_out = 7'b0011000; end
                4'b1010://
                    begin decoder_out = 7'b0001000; end
                4'b1011://
                    begin decoder_out = 7'b0000011; end
                4'b1100://
                    begin decoder_out = 7'b0100111; end
```

7 Segment Decoder Source Code (Active Low Logic) Part 1

```
    always@(decoder_in)
        begin
            case(decoder_in)
                4'b0000://
                    begin decoder_out = 7'b1000000; end
                4'b0001://
                    begin decoder_out = 7'b1111001; end
                4'b0010://
                    begin decoder_out = 7'b0100100; end
                4'b0011://
                    begin decoder_out = 7'b0110000; end
                4'b0100://
                    begin decoder_out = 7'b0011001; end
                4'b0101://
                    begin decoder_out = 7'b0010010; end
                4'b0110://
                    begin decoder_out = 7'b0000010; end
                4'b0111://
                    begin decoder_out = 7'b1111000; end
                4'b1000://
                    begin decoder_out = 7'b0000000; end
                4'b1001://
                    begin decoder_out = 7'b0011000; end
                4'b1010://
                    begin decoder_out = 7'b0001000; end
                4'b1011://
                    begin decoder_out = 7'b0000011; end
                4'b1100://
                    begin decoder_out = 7'b0100111; end
                4'b1101://
                    begin decoder_out = 7'b0100001; end
                4'b1110://
                    begin decoder_out = 7'b0000110; end
                4'b1111:
                    begin decoder_out = 7'b0001110; end
                default:
                    begin decoder_out = 7'b0000000; end
            endcase

        end

endmodule
```

7 Segment Decoder Source Code (Active Low Logic) Part 2

```
`timescale 1 ns/100 ps
module Testbench_Decoder_4to7seg();

    reg [3:0] decoder_in_s;
    wire [6:0] decoder_out_s;

    Decoder_4to7seg DUT(decoder_in_s, decoder_out_s);

    initial
        begin
            // Testing 0
            decoder_in_s = 4'd0;
            #10;
            // Testing 1
            decoder_in_s = 4'd1;
            #10;
            // Testing 2
            decoder_in_s = 4'd2;
            #10;
            // Testing 3
            decoder_in_s = 4'd3;
            #10;
            // Testing 4
            decoder_in_s = 4'd4;
            #10;
            // Testing 5
            decoder_in_s = 4'd5;
            #10;
            // Testing 6
            decoder_in_s = 4'd6;
            #10;
            // Testing 7
            decoder_in_s = 4'd7;
            #10;
            // Testing 8
            decoder_in_s = 4'd8;
            #10;
            // Testing 9
            decoder_in_s = 4'd9;
```

7 Segment Decoder Testbench Source Code Part 1

```
            #10;
            // Testing 4
            decoder_in_s = 4'd4;
            #10;
            // Testing 5
            decoder_in_s = 4'd5;
            #10;
            // Testing 6
            decoder_in_s = 4'd6;
            #10;
            // Testing 7
            decoder_in_s = 4'd7;
            #10;
            // Testing 8
            decoder_in_s = 4'd8;
            #10;
            // Testing 9
            decoder_in_s = 4'd9;
            #10;
            // Testing 10 (A)
            decoder_in_s = 4'd10;
            #10;
            // Testing 11 (B)
            decoder_in_s = 4'd11;
            #10;
            // Testing 12 (C)
            decoder_in_s = 4'd12;
            #10;
            // Testing 13 (D)
            decoder_in_s = 4'd13;
            #10;
            // Testing 14 (E)
            decoder_in_s = 4'd14;
            #10;
            // Testing 15 (F)
            decoder_in_s = 4'd15;
        end
endmodule
```

7 Segment Decoder Testbench Source Code Part 2

```
// Course Number: ECE 5440
// Author: Smithson Arrey, 0046
// Module Name: Game_Checker
// Description: This module is a generic game checker that can interpret a 4-bit game condition
//              and output a 2-bit response indicating the game's state. Bit 0 of the response indicates a non-valid condition.
//              Bit 1 of the response indicates a valid condition. Only one bit will be activated at a time.
// Comments: The use of this module in Lab 1 will be to use the 4-bit sum of both player's numbers
//           as this module's input. The output will be wired to the verification leds

module Game_Checker(GameCondition, GameState);
    input [3:0] GameCondition;
    output [1:0] GameState;
    reg [1:0] GameState;

    always@(GameCondition)
    begin
        if(GameCondition == 4'b1111)
            begin GameState = 2'b10; end
        else
            begin GameState = 2'b01; end
    end
endmodule
```

Game_Checker Source Code

Lab2_ARREY_Smithson > src > LoadRegister.v

```
// Course Number: ECE 5440
// Author: Smithson Arrey, 0046
// Module Name: LoadRegister
// Description: This module is a generic load register.
// It shifts the input to the output on command. Acts as a simple memory device.
// Comments: N/A

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
            // normal operation after reset
                begin
                    if (Load == 1'b1)
                        begin
                            D_out <= D_in;
                        end
                end
        end
endmodule

```

LoadRegister Module Source Code

```
// Course Number: ECE 5440
// Author: Smithson Arrey, 0046
// Module Name: testbench_LoadRegister
// Description: This module is a testbench for the LoadRegister module
// it will show various functionality of the LoadRegister and demonstrate a single cycle pulse
// Comments: N/A

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

    always //empty sensitivity list
        begin
            Clk = 1'b0;
            #10;
            Clk = 1'b1;
            #10;
        end

    LoadRegister DUT_LoadRegister1 (D_in, D_out, Clk, Rst, Load);

    initial begin
        Rst = 1'b1;
        Load = 1'b0;
        D_in = 4'b0000;

        @(posedge Clk);
        @(posedge Clk);
        #5 Rst = 1'b0;
        @(posedge Clk);
        @(posedge Clk);
        @(posedge Clk);
        @(posedge Clk);
        #5 Rst = 1'b1;
        @(posedge Clk);
        @(posedge Clk);
        @(posedge Clk);
        @(posedge Clk);
        #5 D_in = 4'b0001;
        @(posedge Clk);
```

```
44          //single cycle pulse in test bench
45          #5 Load = 1'b1;
46          @(posedge Clk);
47          #5 Load = 1'b0;
48          @(posedge Clk);
49          //              //
50          #5 D_in = 4'b1111;
51          @(posedge Clk);
52          @(posedge Clk);
53          @(posedge Clk);
54          #5 D_in = 4'b0111;
55          @(posedge Clk);
56          @(posedge Clk);
57          #5 Load = 1'b1;
58          @(posedge Clk);
59          #5 Load = 1'b0;
60          @(posedge Clk);
61      end
62  endmodule
63
```

LoadRegister Testbench Source Code

```
// Course Number: ECE 5440
// Author: Smithson Arrey, 0046
// Module Name: ButtonShaper
// Description: This module is a generic button shaper. It transforms multi clock cycle input pulses into a single cycle pulse
// Comments: N/A
module ButtonShaper(B_in, B_out, Clk, Rst);
    input B_in, Clk, Rst;
    output B_out;
    reg B_out;
    parameter INIT = 0, PULSE = 1, WAIT = 2;
    reg [1:0] State, StateNext;

    always@(State, B_in) begin
        case(State)
            INIT: begin
                B_out = 1'b0;
                if(B_in == 1'b0)
                    StateNext = PULSE;
                else
                    StateNext = INIT;
            end
            PULSE: begin
                B_out = 1'b1;
                StateNext = WAIT;
            end
            WAIT: begin
                B_out = 1'b0;
                if(B_in == 1'b1)
                    StateNext = INIT;
                else
                    StateNext = WAIT;
            end
            default: begin
                B_out = 1'b0;
                StateNext = WAIT;
            end
        endcase
    end

    always@(posedge Clk) begin
        if(Rst == 1'b0)
            State <= WAIT;
        else
            State <= StateNext;
    end
endmodule
```

Button Shaper Module Source Code

```
// Course Number: ECE 5440
// Author: Smithson Arrey, 0046
// Module Name: testbench_ButtonShaper
// Description: This module is intended to test the ButtonShaper module. It will observe two multicycle button press to ensure the system is functional
// Comments: N/A
`timescale 1 ns/100 ps
module testbench_ButtonShaper();
    reg Clk;
    reg Rst;
    reg B_in;
    wire B_out;

    always //empty sensitivity list
        begin
            Clk = 1'b0;
            #10;
            Clk = 1'b1;
            #10;
        end

    ButtonShaper DUT (B_in, B_out, Clk, Rst);

    initial begin
        // Reset Cycle
        Rst = 1'b1;
        B_in = 1'b0;
        @(posedge Clk);
        @(posedge Clk);
        #5 Rst = 1'b0;
        @(posedge Clk);
        @(posedge Clk);
        @(posedge Clk);
        @(posedge Clk);
        #5 Rst = 1'b1;
        @(posedge Clk);
        @(posedge Clk);
        @(posedge Clk);
        @(posedge Clk);

        //            //
        // Button Triggered
        #5 B_in = 1'b1;
```

```
        @(posedge Clk);
        @(posedge Clk);
        @(posedge Clk);
        #5 Rst = 1'b1;
        @(posedge Clk);
        @(posedge Clk);
        @(posedge Clk);
        @(posedge Clk);

        //            //
        // Button Triggered
        #5 B_in = 1'b1;
        @(posedge Clk);
        @(posedge Clk);
        @(posedge Clk);
        @(posedge Clk);
        //            //
        //Button off
        #5 B_in = 1'b0;
        @(posedge Clk);
        @(posedge Clk);
        @(posedge Clk);
        @(posedge Clk);
        //            //

        // Button Triggered
        #5 B_in = 1'b1;
        @(posedge Clk);
        @(posedge Clk);
        @(posedge Clk);
        @(posedge Clk);
        //            //
        //Button off
        #5 B_in = 1'b0;
        @(posedge Clk);
        @(posedge Clk);
        @(posedge Clk);
        @(posedge Clk);

    end
endmodule

```

Button Shaper Module Testbench Source Code

Lab2_ARREY_Smithson > src > AccessController.v

```
// Course Number: ECE 5440
// Author: Smithson Arrey, 0046
// Module Name: AccessController
// Description: This module is prevents the user from accessing the game without first logging in.
// Uses sequential logic to input a predetermined password. Can be used to log in and out of the game.
// Comments: N/A

module AccessController(PasswordEnter, PasswordDigit, Load_P1_In, Load_P2_In, LoggedIn, LoggedOut, Load_P1_Out, Load_P2_Out, Clk, Rst);
    input PasswordEnter, Load_P1_In, Load_P2_In, Clk, Rst;
    output Load_P1_Out, Load_P2_Out, LoggedIn, LoggedOut;
    input [3:0] PasswordDigit;
    reg Load_P1_Out, Load_P2_Out, LoggedIn, LoggedOut;
    parameter DIGIT_ONE = 0, DIGIT_TWO = 1, DIGIT_THREE = 2, DIGIT_FOUR = 3, AUTHENTICATED = 4;
    reg [2:0] State;

    always@(posedge Clk)
    begin
        if(Rst == 1'b0)
            begin
            Load_P1_Out <= 1'b0;
            Load_P2_Out <= 1'b0;
            LoggedIn <= 1'b0;
            LoggedOut <= 1'b1;
            State <= DIGIT_ONE;
            end
        else

            case(State)
                DIGIT_ONE: begin
                    Load_P1_Out <= 1'b0;
                    Load_P2_Out <= 1'b0;
                    LoggedIn <= 1'b0;
                    LoggedOut <= 1'b1;
                    if (PasswordEnter == 1'b1)
                            begin
                            if(PasswordDigit == 4'd0)
                                State <= DIGIT_TWO;
                            else
                                State <= DIGIT_ONE;
                            end
                       else
                            begin State <= DIGIT_ONE; end
                    end
```

```
44                  DIGIT_TWO: begin
45                      Load_P1_Out <= 1'b0;
46                      Load_P2_Out <= 1'b0;
47                      LoggedIn <= 1'b0;
48                      LoggedOut <= 1'b1;
49                      if (PasswordEnter == 1'b1) begin
50                          if(PasswordDigit == 4'd0)
51                              State <= DIGIT_THREE;
52                          else
53                          //INCORRECT, START OVER
54                              State <= DIGIT_ONE;
55                          end
56                      else //NOT ENTERED, STAY PUT
57                          begin State <= DIGIT_TWO; end
58                  end
59                  DIGIT_THREE: begin
60                      Load_P1_Out <= 1'b0;
61                      Load_P2_Out <= 1'b0;
62                      LoggedIn <= 1'b0;
63                      LoggedOut <= 1'b1;
64                      if (PasswordEnter == 1'b1) begin
65                          if(PasswordDigit == 4'd4)
66                              State <= DIGIT_FOUR;
67                          else
68                          //INCORRECT, START OVER
69                              State <= DIGIT_ONE;
70                          end
71                      else //NOT ENTERED, STAY PUT
72                          begin State <= DIGIT_THREE; end
73                  end
74                  DIGIT_FOUR: begin
75                      Load_P1_Out <= 1'b0;
76                      Load_P2_Out <= 1'b0;
77                      LoggedIn <= 1'b0;
78                      LoggedOut <= 1'b1;
79                      if (PasswordEnter == 1'b1) begin
80                          if(PasswordDigit == 4'd6)
81                              State <= AUTHENTICATED;
82                          else
83                          //INCORRECT, START OVER
84                              State <= DIGIT_ONE;
85                          end
86                      else //NOT ENTERED, STAY PUT
```

```
                        //INCORRECT, START OVER
                            State <= DIGIT_ONE;
                        end
                    else //NOT ENTERED, STAY PUT
                        begin State <= DIGIT_FOUR; end
                end
                AUTHENTICATED: begin
                    Load_P1_Out = Load_P1_In;
                    Load_P2_Out = Load_P2_In;
                    LoggedIn <= 1'b1;
                    LoggedOut <= 1'b0;
                    if (PasswordEnter == 1'b1) begin
                        //LOG OUT
                        State <= DIGIT_ONE;
                    end
                    else //NOT ENTERED, STAY PUT
                        begin State <= AUTHENTICATED; end
                end
                default: begin
                    Load_P1_Out <= 1'b0;
                    Load_P2_Out <= 1'b0;
                    LoggedIn <= 1'b0;
                    LoggedOut <= 1'b1;
                    State <= DIGIT_ONE;
                end
            endcase
        end
endmodule

```

AccessController Source Code

Lab2_ARREY_Smithson > src > testbench_AccessController.v

```
// Course Number: ECE 5440
// Author: Smithson Arrey, 0046
// Module Name: testbench_AccessController
// Description: This module is intended to test the AccessController module.
// It will ensure that this module correctly handles both a correct log in sequence and an incorrect log in sequence
// Comments: N/A
`timescale 1 ns/100 ps
module testbench_AccessController();
    reg Clk;
    reg Rst;
    reg tPasswordEnter, tLoad_P1_In, tLoad_P2_In;
    reg [3:0] tPasswordDigit;
    wire tLoggedIn, tLoggedOut, tLoad_P1_Out, tLoad_P2_Out;

    always //empty sensitivity list
        begin
            Clk = 1'b0;
            #10;
            Clk = 1'b1;
            #10;
        end

    AccessController DUT_AccessController(tPasswordEnter, tPasswordDigit, tLoad_P1_In, tLoad_P2_In, tLoggedIn, tLoggedOut, tLoad_P1_Out, tLoad_P2_Out, Clk, Rst);

    initial begin
        Rst = 1'b1;
        tPasswordEnter = 1'b0;
        tLoad_P1_In = 1'b0;
        tLoad_P2_In = 1'b0;

        //testing valid password            //
        //
        #5 tPasswordDigit = 4'd0;
        @(posedge Clk);
        @(posedge Clk);
        #5 tPasswordEnter = 1'b1;
        @(posedge Clk);
        #5 tPasswordEnter = 1'b0;
        @(posedge Clk);
        //              //
        #5 tPasswordDigit = 4'd0;
        @(posedge Clk);
        @(posedge Clk);
```

```
44          #5 tPasswordEnter = 1'b1;
45          @(posedge Clk);
46          #5 tPasswordEnter = 1'b0;
47          @(posedge Clk);
48
49          //            //
50          #5 tPasswordDigit = 4'd4;
51          @(posedge Clk);
52          @(posedge Clk);
53          #5 tPasswordEnter = 1'b1;
54          @(posedge Clk);
55          #5 tPasswordEnter = 1'b0;
56          @(posedge Clk);
57
58          //            //
59          #5 tPasswordDigit = 4'd6;
60          @(posedge Clk);
61          @(posedge Clk);
62          #5 tPasswordEnter = 1'b1;
63          @(posedge Clk);
64          #5 tPasswordEnter = 1'b0;
65          @(posedge Clk);
66
67          //   input          //
68
69          #5 tLoad_P1_In = 1'b1;
70          #5 tLoad_P2_In = 1'b1;
71          @(posedge Clk);
72          @(posedge Clk);
73          #5 tLoad_P1_In = 1'b0;
74          #5 tLoad_P2_In = 1'b0;
75          @(posedge Clk);
76          @(posedge Clk);
77
78          //logging out via password enter button
79          #5 tPasswordEnter = 1'b1;
80          @(posedge Clk);
81          @(posedge Clk);
82          #5 tLoad_P1_In = 1'b1; //confirming that there will be no output
83          #5 tLoad_P2_In = 1'b0;
84          @(posedge Clk);
85          @(posedge Clk);
86
```

Lab2_ARREY_Smithson > src > testbench_AccessController.v

```
87          // Reset Cycle
88          #5 Rst = 1'b1;
89
90          @(posedge Clk);
91          @(posedge Clk);
92          #5 Rst = 1'b0;
93          @(posedge Clk);
94          @(posedge Clk);
95          @(posedge Clk);
96          @(posedge Clk);
97          #5 Rst = 1'b1;
98          @(posedge Clk);
99          @(posedge Clk);
100         @(posedge Clk);
101         @(posedge Clk);
102
103         //testing invalid password            //
104         //
105         #5 tPasswordDigit = 4'd1;
106         @(posedge Clk);
107         @(posedge Clk);
108         #5 tPasswordEnter = 1'b1;
109         @(posedge Clk);
110         @(posedge Clk);
111         tPasswordEnter = 1'b0;
112         //             //
113         #5 tPasswordDigit = 4'd2;
114         @(posedge Clk);
115         @(posedge Clk);
116         #5 tPasswordEnter = 1'b1;
117         @(posedge Clk);
118         @(posedge Clk);
119         #5 tPasswordEnter = 1'b0;
120         //             //
121         #5 tPasswordDigit = 4'd3;
122         @(posedge Clk);
123         @(posedge Clk);
124         #5 tPasswordEnter = 1'b1;
125         @(posedge Clk);
126         @(posedge Clk);
127         #5 tPasswordEnter = 1'b0;
128         //             //
129         #5 tPasswordDigit = 4'd4;
```

```
128         //             //
129         #5 tPasswordDigit = 4'd4;
130         @(posedge Clk);
131         @(posedge Clk);
132         #5 tPasswordEnter = 1'b1;
133         @(posedge Clk);
134         @(posedge Clk);
135         #5 tPasswordEnter = 1'b0;
136         //   input          //
137
138         #5 tLoad_P1_In = 1'b1;
139         #5 tLoad_P2_In = 1'b1;
140         @(posedge Clk);
141         @(posedge Clk);
142         #5 tLoad_P1_In = 1'b0;
143         #5 tLoad_P2_In = 1'b0;
144         @(posedge Clk);
145         @(posedge Clk);
146
147     end
148 endmodule
149
```

AccessController Testbench Source Code

```
// Course Number: ECE 5440
// Author: Smithson Arrey, 0046
// Module Name: Lab2_ARREY_Smithson
// Description: This is a top-level module intended to build the full circuit of Lab 2: MentalBinaryMathGame
// Comments: N/A

module Lab2_ARREY_Smithson(PasswordEnterButton, Load_Button_1, Load_Button_2, SwitchSetOne, SwitchSetTwo, PlayerOneDisplayCode, PlayerTwoDisplayCode,
    input [3:0] SwitchSetOne, SwitchSetTwo;
    input Rst, Clk;

    input PasswordEnterButton, Load_Button_1, Load_Button_2;
    output [6:0] PlayerOneDisplayCode, PlayerTwoDisplayCode, SummationDisplayCode;
    output [1:0] VerificationLEDCode;

    output LoggedInLED, LoggedOutLED;
    wire Load_P1_In, Load_P2_In, Load_P1_Out, Load_P2_Out, PasswordEnter;
    wire [3:0] P1_D_out, P2_D_out;

    //input shaping
    ButtonShaper LoadButton1Shaper(Load_Button_1, Load_P1_In, Clk, Rst);
    ButtonShaper LoadButton2Shaper(Load_Button_2, Load_P2_In, Clk, Rst);
    ButtonShaper PasswordInputButtonShaper(PasswordEnterButton, PasswordEnter, Clk, Rst);

    //Authentication
    AccessController GameAccessor(PasswordEnter, SwitchSetTwo, Load_P1_In, Load_P2_In, LoggedInLED, LoggedOutLED, Load_P1_Out, Load_P2_Out, Clk, Rst)

    //Player One
    LoadRegister PlayerOneLoadRegister(SwitchSetOne, P1_D_out, Clk, Rst, Load_P1_Out);
    Decoder_4to7seg PlayerOneDecoder(P1_D_out, PlayerOneDisplayCode);

    //Player Two
    LoadRegister PlayerTwoLoadRegister(SwitchSetTwo, P2_D_out, Clk, Rst, Load_P2_Out);
    Decoder_4to7seg PlayerTwoDecoder(P2_D_out, PlayerTwoDisplayCode);

    //Summation
    wire [3:0] InputAdderResult;

    Adder_NC_4 InputAdder(P1_D_out, P2_D_out, InputAdderResult);
    Decoder_4to7seg SummationDecoder(InputAdderResult, SummationDisplayCode);

    //Verification
    Game_Checker SummationChecker(InputAdderResult, VerificationLEDCode);
```

Top Level Module Source Code