Lab 2: Game Access Control on FPGA

Stefan Bucur 3153

ECE5440

Contents

[**1.** Introduction 3](#_Toc32878978)

[**2.** System Architecture and Design 4](#_Toc32878979)

[2.A: Button Shaper (bshaper) 5](#_Toc32878980)

[2.B: Load Register (loadreg) 6](#_Toc32878981)

[2.C: Access Control (access) 7](#_Toc32878982)

[2.D: Addition (adder) 9](#_Toc32878983)

[2.E: 4-to-7 Decoder (decoder7) 9](#_Toc32878984)

[2.F: Check Module (check) 9](#_Toc32878985)

[**3.** Simulations 10](#_Toc32878986)

[3.A: Button Shaper (bshaper) 10](#_Toc32878987)

[3.B: Load Register (loadreg) 10](#_Toc32878988)

[3.C: Access Control (access) 11](#_Toc32878989)

[3.D: Addition (adder) 12](#_Toc32878990)

[3.E: 4-to-7 Decoder (decoder7) 12](#_Toc32878991)

[3.F: Check Module (check) 13](#_Toc32878992)

[**4.** FPGA Board Testing results 14](#_Toc32878993)

[**5.** Conclusion 21](#_Toc32878994)

[**5.** References 21](#_Toc32878995)

[**I.** Appendix 21](#_Toc32878996)

# **1.** Introduction

Lab 2 is a math memorization game that tests players’ ability to add and convert between binary, decimal and hexadecimal in their minds. The digital logic described in this design was made for the Altera DE2-115 prototyping board. The development in this lab focuses on implementing a login system to the game and a more robust player input system. When the device is first powered on, the number displays and indication lights are visible, but the player(s) cannot immediately begin playing the game; they must first enter the correct sequence of four-bit numbers one by one into the login system. Once the correct sequence is read, the system will indicate the game can be played and players can begin entering numbers with their designated button and sets of switches.

Players enter numbers into the system by toggling switches that represent a binary number and pressing their button. If the two numbers add to 0xF (15 in decimal), the FPGA will change the indicator LEDs to verify the correct result. Players record score using their own means and can take turns being the first player to enter a number each round. If the players would like to leave the board for any reason and not want others to alter the inputs, they can log out of the game and the system will preserve the present values should the players log in again. At any point during operation the reset button can be pressed, reverting the system back to its initial state as if it was powered on for the first time. In the modules with a state machine, the transition upon activating RESET is assumed to be its default state.

# **2.** System Architecture and Design

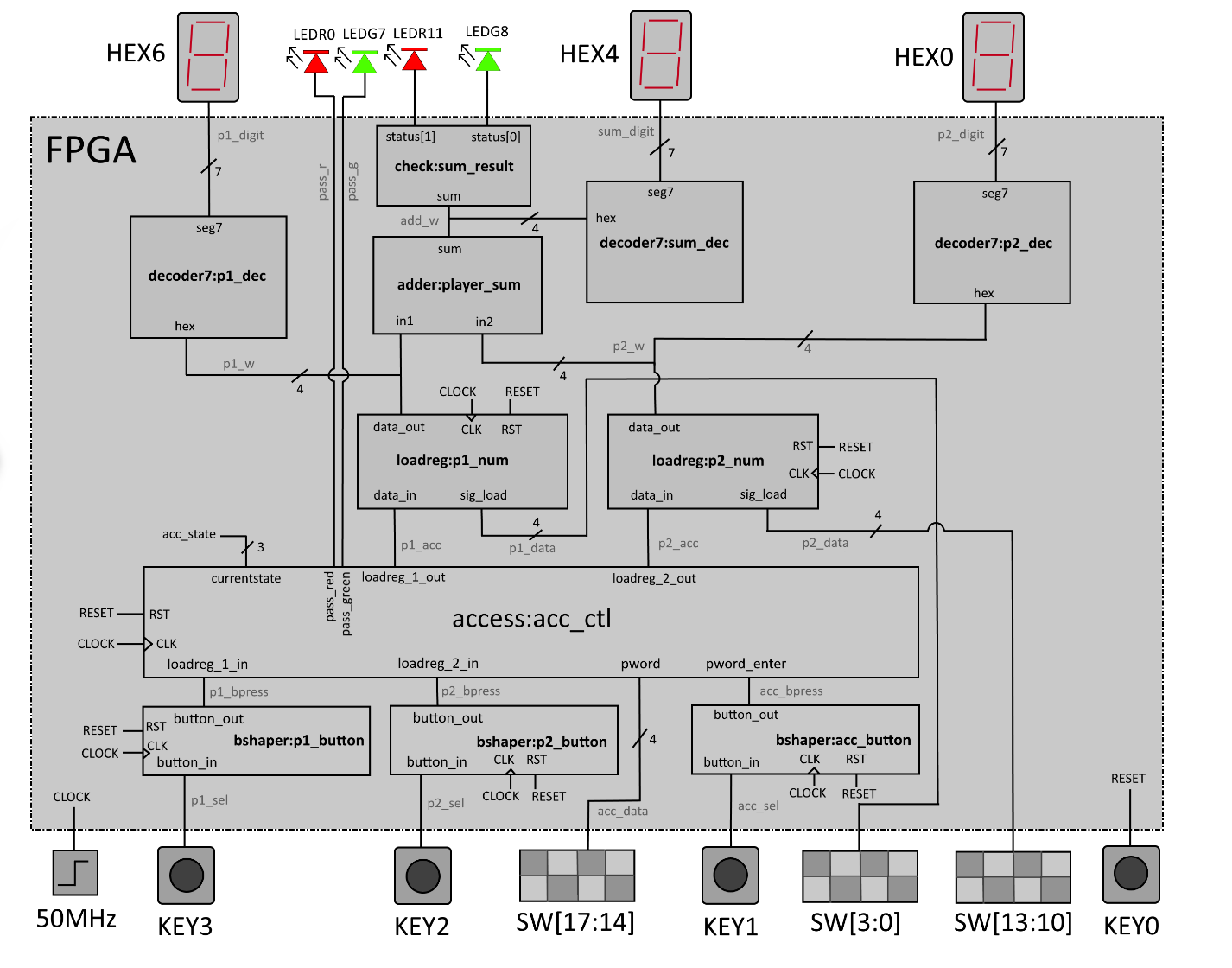


Figure : FPGA system architecture for Lab 2. KEY inputs pin out to the 4 onboard buttons. SW switches pin out to the onboard switches. A 50MHz clock is wired into the FPGA from an onboard oscillator. Input and output names can be found on the DE2-115 board.

|  |  |
| --- | --- |
| Input/output | Description |
| 50MHz | Input: Onboard 50MHz clock from the DE2-115 board. Clock signal is synchronized with a PLL. |
| SW[17:14] | Input: The four input switches for player 1 to enter a 4-bit value into the FPGA. |
| SW[13:10] | Input: The four input switches for player 2 to enter a 4-bit value into the FPGA. |
| SW[3:0] | Input: The four input switches to input a 4-bit value into the FPGA’s access control module. |
| KEY3 | Input: Push button to instruct the system to load player 1’s value into their load register. |
| KEY2 | Input: Push button to instruct the system to load player 2’s value into their load register. |
| KEY1 | Input: Push button to input a password digit from SW[3:0] into the FPGA’s access module. |
| KEY0 | Input: Push button to reset all system modules to their default state. |
| HEX6 | Output: Seven-segment display for player 1’s number within the FPGA. |
| HEX4 | Output: Seven-segment display for player 2’s number within the FPGA. |
| HEX0 | Output: Seven-segment display for the sum of the two player’s values within the FPGA. |
| LEDR0 | Output: Red LED indicating the system is in logout mode and requires a password. |
| LEDG7 | Output: Green LED indicating the system is in login mode and will accept player input. |
| LEDR11 | Output: Red LED indicating the sum of the two player’s numbers does not equal 15 (0xF). |
| LEDG8 | Output: Green LED indicating the sum of the two player’s numbers equals 15 (0xF). |

Table 1: short description of the I/O connected to the FPGA on the DE2-115 board.

The FPGA’s sequential logic modules use the DE2-115’s 50MHz internal clock signal to update on its rising edge signal. Apart from RESET, the other three input button signals are first passed through their own button shaper module. This is due to the internal state machines of both the Access Control and Load Register modules treating a button activation signal as a new button press upon each new clock cycle. In order to reduce the complexity of the state machine, the button shaper converts the button’s debounced active low signal into a positive logic pulse that lasts one clock cycle. RESET is not shaped because the sequential logic modules should ideally be subject to multiple subsequent reset signals to ensure the entire system is set to the same initial state. The acc\_state wire within the FPGA is left for debugging purposes in simulations. Explained further below is a detailed explanation of the functionality of each of the modules in Figure 1. Signal names within module borders are defined within the Verilog module. Wire names are defined in the top-level module.

## 2.A: Button Shaper (bshaper)

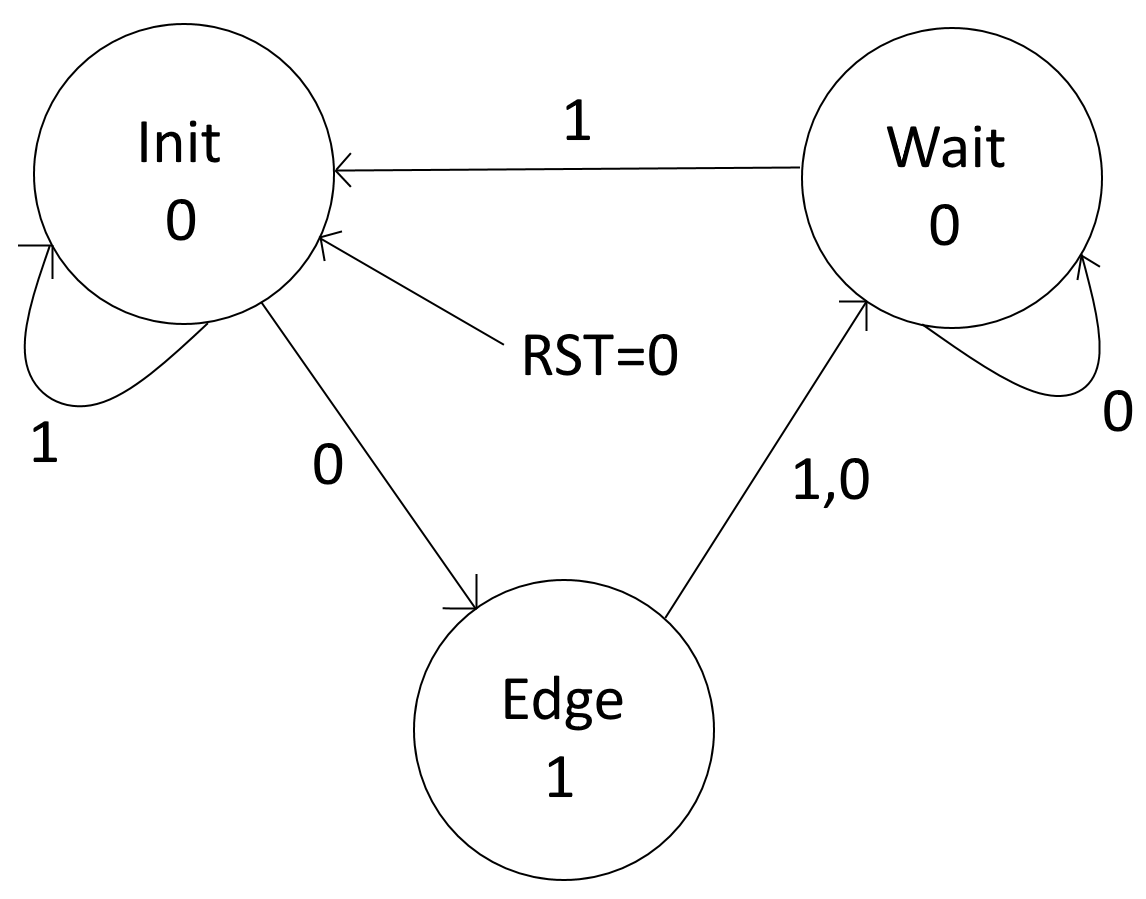


Figure : Finite state machine (FSM) for the button shaper. State values correspond to the button\_out signal. Transition arrow values correspond to the button\_in signal. RST = 1 does not affect the FSM and is therefore omitted.

|  |  |
| --- | --- |
| Input/output | Description |
| CLK | Input: Clock circuit for sequential logic. Module updates on rising clock edge. |
| RST | Input: Resets the state machine to its default state (Init) on active low pulse. |
| b\_in | Input: Raw button input from the board. Input assumed to be debounced. |
| b\_out | Output: Shaped button output to send to sequential logic circuits. |

Table 2: short description of the button shaper’s I/O.

The purpose of using the button shaper is to convert button inputs into signal events that are easier to process in sequential logic. By making the button press last one clock cycle at the start, the modules connected at the button shaper’s output will trigger an event at the next clock cycle without triggering another erroneously at the next cycle.

The button shaper module is a two-procedure finite state machine that converts an active low input signal lasting multiple clock cycles and converts it into an active high square pulse lasting only one clock cycle. The Init state is the default state, where it waits for the button active signal to change at which point it will transition to the edge state within that clock cycle. At the edge state, the output signal is set high and the state machine transitions to a wait state. Here it waits for the input signal to revert to its inactive value at which point the button shaper returns to the Init state. If RST is active when the button is held down, the state machine will ignore any button input until the reset is deactivated.

## 2.B: Load Register (loadreg)

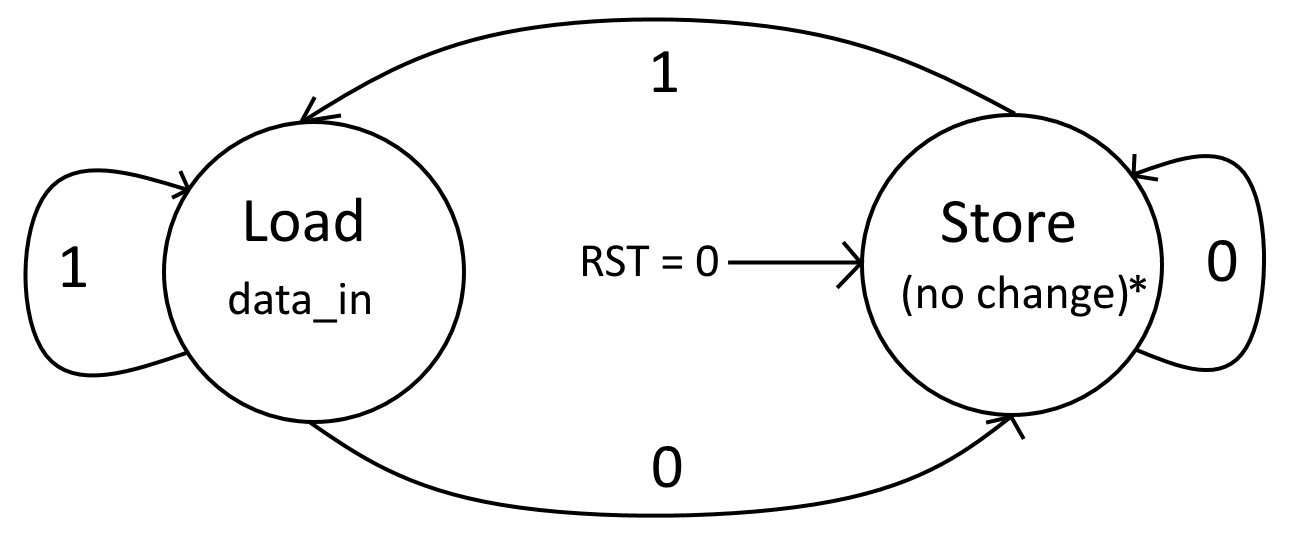


Figure : Finite state machine (FSM) for the load register module. State values correspond to the data\_out signal. Transition arrow values correspond to the sig\_load signal. RST = 1 does not affect the FSM and is therefore omitted.

|  |  |
| --- | --- |
| Input/output | Description |
| CLK | Input: Clock circuit for sequential logic. Module updates on rising clock edge. |
| RST | Input: Resets the state machine to its default state (Init) on active low pulse. |
| data\_in | Input: 4-bit number that will be written to the register. |
| sig\_load | Input: shaped button signal to instruct the register to load a new value to its register. |
| data\_out | Output: Push button to instruct the system to load player 1’s value into their load register. |

Table 3: short description of the load register’s I/O.

The load register, while not being a requirement for proper operation, is implemented using a one-procedure finite state machine. There are only two states, where Store will keep the same output value unless it is reset, and upon receiving a high pulse from the button shaper will read from the player’s switch input (internally labelled data\_in) in the Load state. Note in Figure 1 that sig\_load is blocked by the access module and will only receive a signal if the access module passes it from the assigned button shaper as explained in section 2.C. Resetting the module with the RST signal will force the state machine to the Store state and overwrite the output to 0.

## 2.C: Access Control (access)

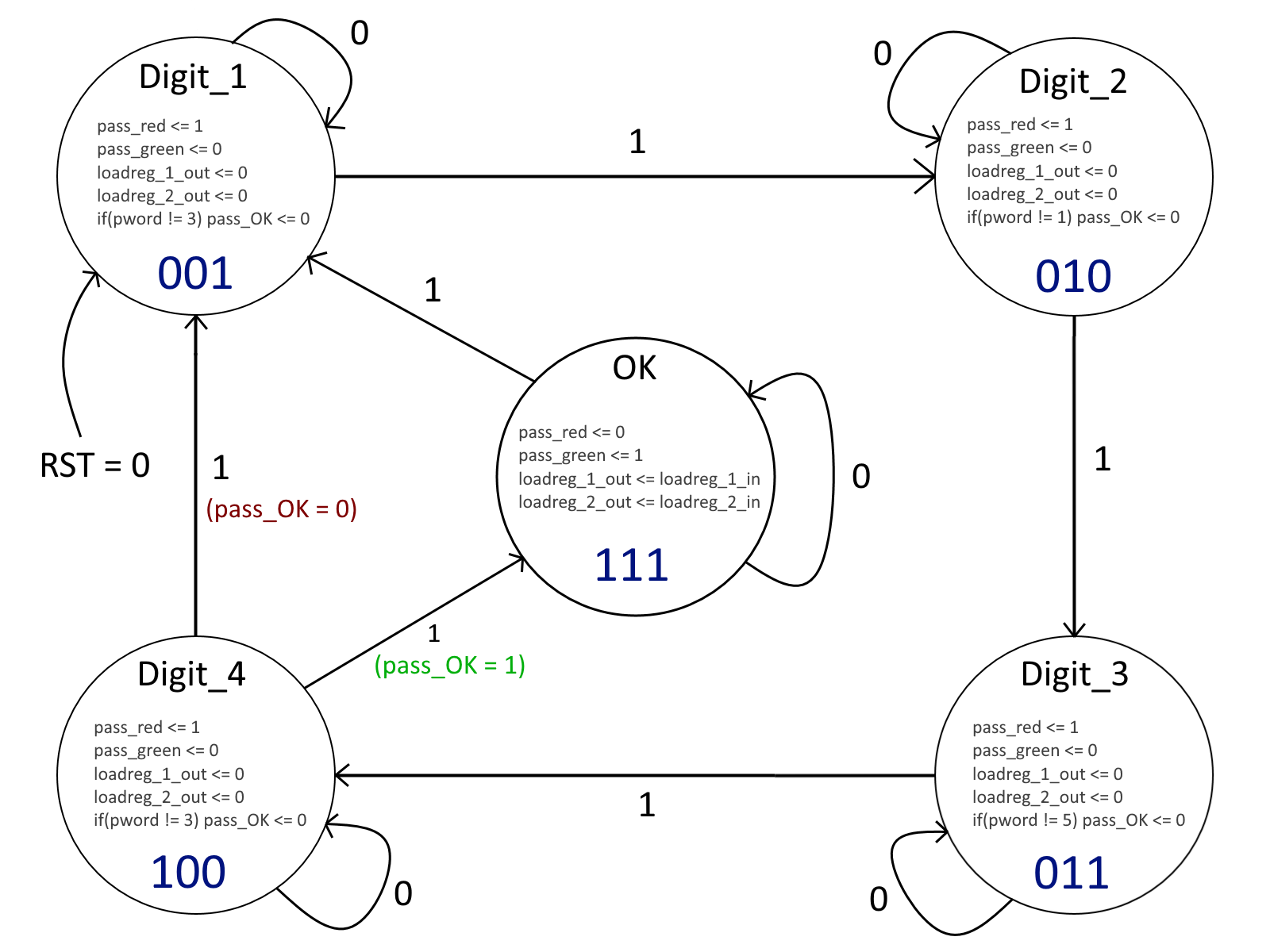


Figure : Finite state machine (FSM) for the access control module. Transition arrow values correspond to the pword\_enter signal. RST = 1 does not affect the FSM and is therefore omitted. The if statement within each state does not evaluate until the FSM changes the current state.

|  |  |
| --- | --- |
| Input/output | Description |
| CLK | Input: Clock circuit for sequential logic. Module updates on rising clock edge. |
| RST | Input: Resets the state machine to its default state (Init) on active low pulse. |
| loadreg\_1\_in | Input: Player 1’s button signal from the button shaper’s sig\_load output. |
| loadreg\_2\_in | Input: Player 2’s button signal from the button shaper’s sig\_load output. |
| pword | Input: The four input switches for the next digit in the password sequence. |
| pword\_enter | Input: Push button to instruct the system to load player 2’s value into their load register. |
| loadreg\_1\_out | Output: Passthrough for player 1 button signal; blocked if logged out. |
| loadreg\_2\_out | Output: Passthrough for player 2 button signal; blocked if logged out. |
| pass\_red | Output: On only when the user is not logged in. |
| pass\_green | Output: On only if the user successfully logs in with the right passcode. |
| currentstate | Output: Value corresponding to the current state within the finite state machine. |

Table 4: short description of the access control module.

The access module is responsible for ensuring players can send button signals into the load registers only after the correct sequence is entered into the module via the pword 4-bit input and the pword\_enter signal which is first passed through a button shaper. For this design, the sequence is **3 1 5 3**. The state names represent the number that the access module is asking for at that time. The 3-bit binary number at the bottom of each state is the numerical value corresponding to that state within the FSM and is stored as the variable currentstate. The module is configured to allow currentstate to be wired to an output pin, but for this design it is intentionally left out, as there already exists a pair of indicator lights (pass\_r for a red LED and pass\_g for green) that show when the player(s) are logged out (only red on) or logged in (only green on). The module still has it available as an output for either future use in other designs or debugging within simulations.

When pword\_enter is 1, the if statement in each state circle will check if the correct number is present within pword. If incorrect, the 1-bit value pass\_OK acts as a flag for the state machine to indicate that a number within the sequence is incorrect. The module then transitions to the next Digit state. Digit\_4 not only evaluates the last digit, but the next state it transitions to depends on the final value of pass\_OK. If pass\_OK has not been altered from its default value of 1, the state machine is set to an OK state where the button signals from the player buttons are allowed through the module and carried into the load register’s sig\_load input each clock cycle.

It is important to note that if an incorrect number is entered, the state machine does not return to Digit\_1 and carries on asking for the next digit in the sequence. Meaning if the 2nd digit is incorrect and the player tries to enter the sequence again from its current state, the access module will not go to the OK state. The player can either press the KEY1 button until the module returns to the Digit\_1 state or press the RESET button.

The access module features a log out system so that players can prevent unwanted changes to the system and preserve the current load register values during that time. When in the OK state, pressing KEY1 again will return to the Digit\_1 state. Pressing RESET does the same action from any state, however it will reset other sequential modules to their respective default values and should only be used if any of the saved value.

## 2.D: Addition (adder)

|  |  |
| --- | --- |
| Input/output | Description |
| in\_1 | Input: One of the 4-bit numbers to add. |
| in\_2 | Input: One of the 4-bit numbers to add. |
| sum | Output: Unsigned 4-bit sum of the two input values. |

Table 5: short description of the adder module.

The system uses a 4-bit adder to sum up the two player numbers and output the result. In this system the in1 and in2 values are wired to the output of the load register modules and update as soon as one of the input signals change. The sum is wired to a seven-segment display decoder to present it to the user.

## 2.E: 4-to-7 Decoder (decoder7)

|  |  |
| --- | --- |
| Input/output | Description |
| hex | Input: 4-bit hex digit value. |
| seg7 | Output: 7-bit value to wire to the seven-segment display. |

Table 6: short description of the decoder module.

In order to display numerical values in a clear and readable manner on the DE2-115 board, the FPGA must first convert the number into a 7-bit value to output to one of the seven-segment displays available on the board. The decoder7 module converts a 4-bit value and interprets it as a hexadecimal digit and output the matching 7-bit value that displays the digit in standard hex format.

## 2.F: Check Module (check)

|  |  |
| --- | --- |
| Input/output | Description |
| sum | Input: 4-bit value to compare. |
| status | Output: Two LED values indicating if the input is equal to 15 (green) or not (red). |

Table 5: short description of the adder module.

The system will turn on a red or green LED if the sum of the two player’s numbers is or is not 15, respectively. The check module takes the output of the sum module and checks if the output is equal to 0xF and turns on the proper LED upon receiving the sum value. status[0] is to be wired to a red LED and status[1] wired to a green LED.

# **3.** Simulations

Described below are the system modules running in a simulation testbench. Note that some modules are not tested for every possible value, however edge cases were carefully chosen to reflect real behavior. All simulations of the system modules are done in Altera ModelSim.

## 3.A: Button Shaper (bshaper)

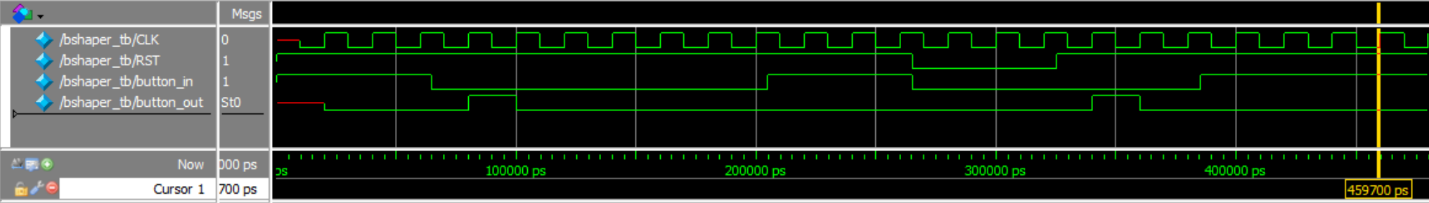


Figure : Simulation results for the button shaper testbench. First test produces a single pulse form a button input. Second test suppresses button input with the reset button.

Before the first clock cycle, the current state of the FSM is not known until the first clock cycle. The button shaper defaults to the Init state if no previous state exists or is undefined. The button shaper will output a one clock cycle wide high pulse at the output right after the first rising edge of the clock where the input equals 0. The input is held down, but the button shaper only produces the one pulse. The button is pressed down again, but the reset is held down for the first few clock cycles. When the reset is held down, the state machine is forced to stay in its Init state, preventing the module from producing a pulse wave until the first clock cycle where the reset is not held down. Again, only one square pulse is output for the entire duration of the button press.

## 3.B: Load Register (loadreg)

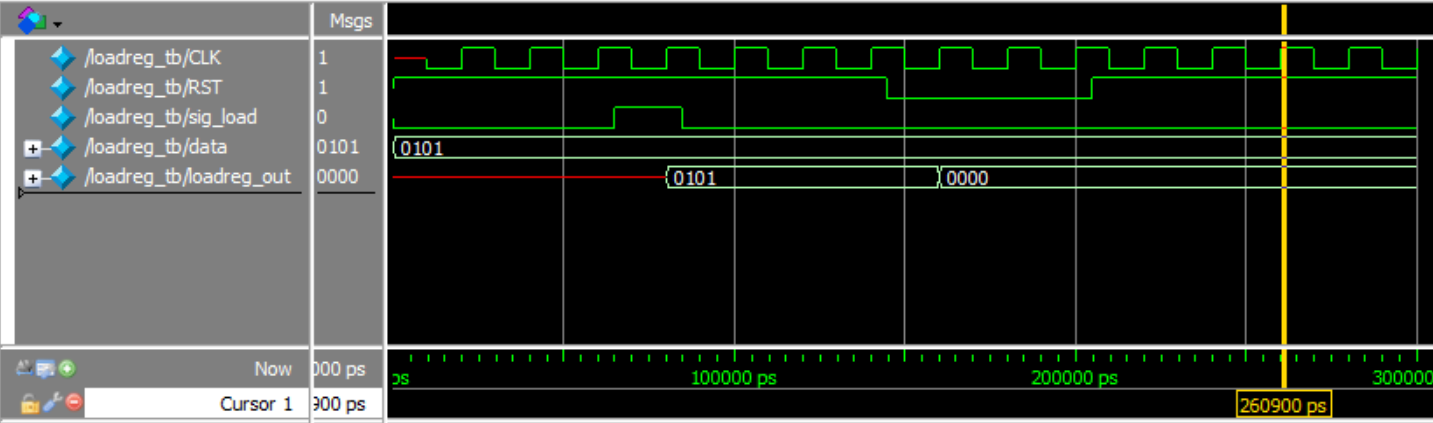


Figure : Simulation results for the load register testbench. The number 5 is loaded into the register after the load signal (sig\_load) and is reset after pressing reset.

Before the first clock cycle, the current state of the FSM is not known until the first clock cycle. The load register defaults to the Store state if no previous state exists or is undefined. Because no previous data\_out exists, the output is undefined until a value is successfully loaded in. Once sig\_load receives a short pulse, the value 5 is loaded from the switches and into the register. Pressing the reset button will set the register to 0.

## 3.C: Access Control (access)

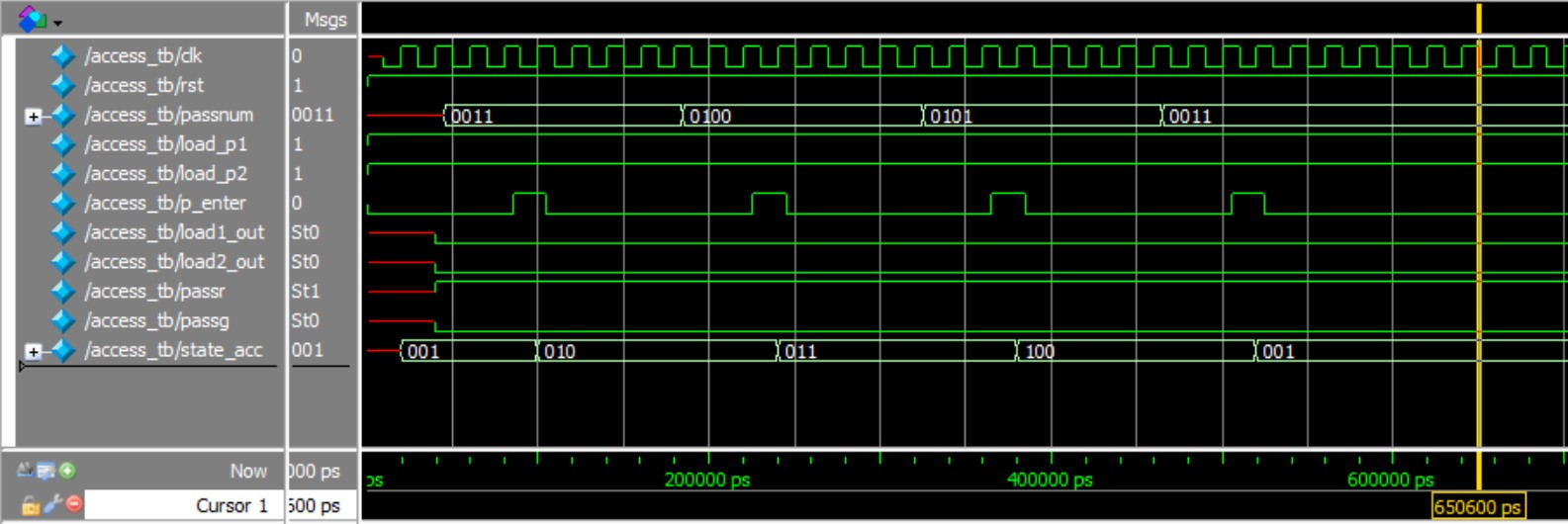


Figure : First simulation of the access control testbench. An incorrect password is entered (3453).

In the first simulation, the initial outputs of the access control are not known due to a lack of a current state value. After it defaults to the Digit\_1 state, the outputs change to the state’s defined values and waits for a button press. After the press, a 3 is read from pword and the state machine advances to Digit\_2 without modifying the pass\_OK flag. The same process occurs on the next press with a 4 loaded in, however since the access module was expecting a 1, the pass\_OK is set to 0 before changing to state Digit\_3. The next correct digit is entered, but pass\_OK remains 0. Once the last correct digit is entered, the state returns to Digit\_1 since pass\_OK changed to 0 in previous states. pass\_OK is reset to 1 before the state machine completes the loop. Note that despite the load inputs being set to 1, the load outputs are 0 because the system is not logged in and therefore blocks the signals.

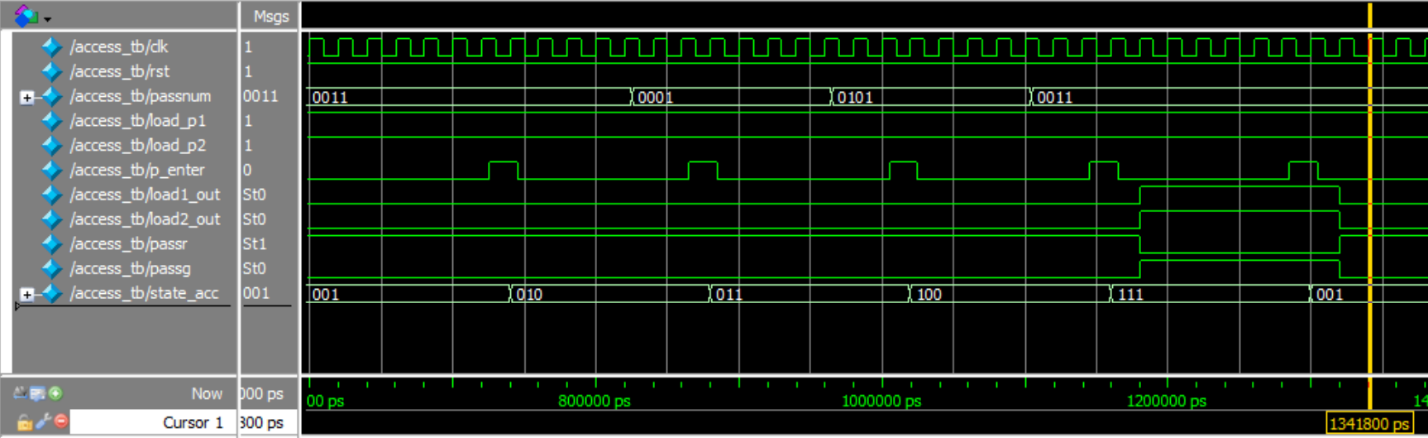


Figure : Second simulation of the access control testbench. Correct sequence is entered followed by a logout.

The test in Figure 8 is a repeat of the test in Figure 7, however the second digit is now correct (value is 1). After the fourth digit is entered pass\_OK is still 1, so the state transitions to OK and the output changes: pass\_r turns off, pass\_g turns on, and the loadreg output signals change to 1 from the input signal values. Pressing the password enter button again will log out of the module, returning the state back to it’s Digit\_1 state and blocking the load signals.

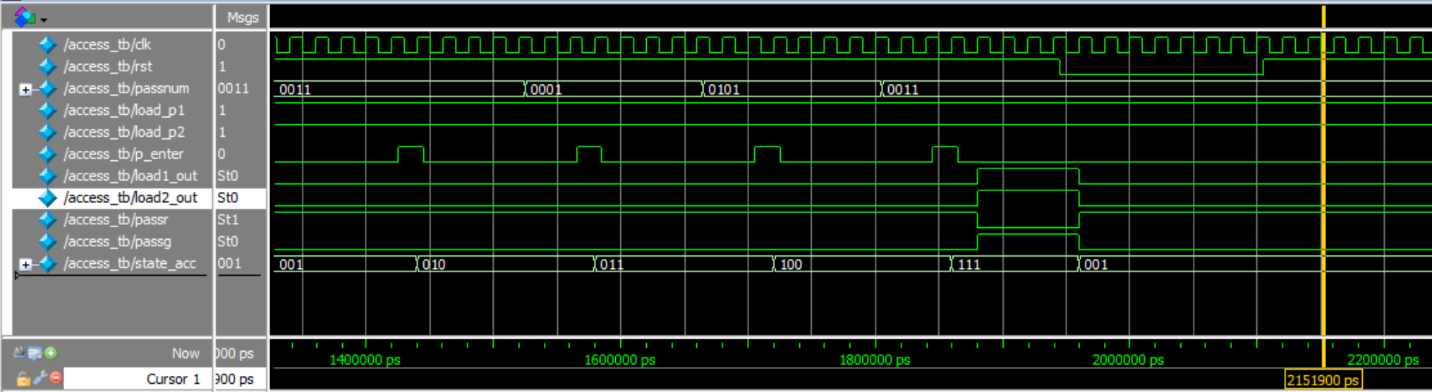
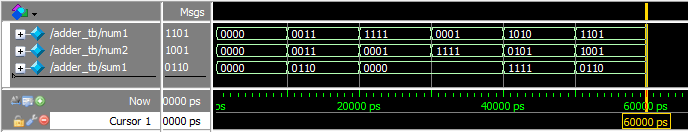


Figure : Third simulation of the access control testbench. Correct sequence is entered followed by a reset.

Figure 9 repeats the same previous test, however the reset button ins pressed in order to log the system out. Holding reset in real world testing will ensure the state machine reverts to the initial state. Resetting while logged in behaves the same way as logging out, however remember that other modules will be reset as well.

## 3.D: Addition (adder)

Figure : Simulation of the adder testbench. Six different pairs of numbers were chosen to display the adder's behavior during normal and edge cases.

Since there are 256 possible input combinations for the adder, the testbench shows multiple example of normal and edge case behavior (in order): correctly adding to zero, correctly adding two numbers that can be displayed with four bits, addition overflow to 0, addition overflow with swapped values to show the commutative property of addition (A+B=B+A), two numbers that add to the ideal value of 15, and finally another overflow to a nonzero number. Note that there is no carry bit out for the adder, making it not ideal for chaining together to form larger adders. All values that do not add to 15 regardless of being able to be stored in 4 bits will count as a failure from the check module as intended; larger values simply overflow to 14 or lower.

## 3.E: 4-to-7 Decoder (decoder7)



Figure : Simulation of the decoder7 testbench. All possible inputs are tested in order.

The DE2-115 manual states a specific pinout for the corresponding seven-segment lights for each display. More information can be found in the DE2-115 user manual in chapter 4.4[1]. Testing each of the possible 4-bit inputs yields the matching 7-bit sequence necessary to correctly light up the segment lines and display the 4-bit value as a hexadecimal digit.

## 3.F: Check Module (check)

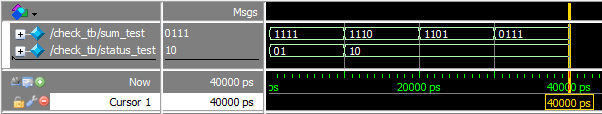


Figure : Simulation results for the check testbench. Testbench inputs the winning value of 15 followed by 3 losing numbers.

The check testbench only needs to check for two cases: winning (15) and losing (0 to 14). status\_test[0] is on only for the losing values and status\_test[1] is on only for the winning value of 15.

# **4.** FPGA Board Testing results

This section details a general demonstration of the system working on a DE2-115 development board. A short video of the steps show below is also available by clicking [this Google Drive link](https://drive.google.com/open?id=18D4qsM7hmdAZwi8sKi5wUMHpbNIN8CYy).



Figure : Board state upon powering up.

Upon powering the device, the system is logged out by default. By default, the load registers will start with a value of zero. The four switches labelled green enter are used to enter the 4-digit passcode for the access controller, with a press of the green-circled button for each digit.

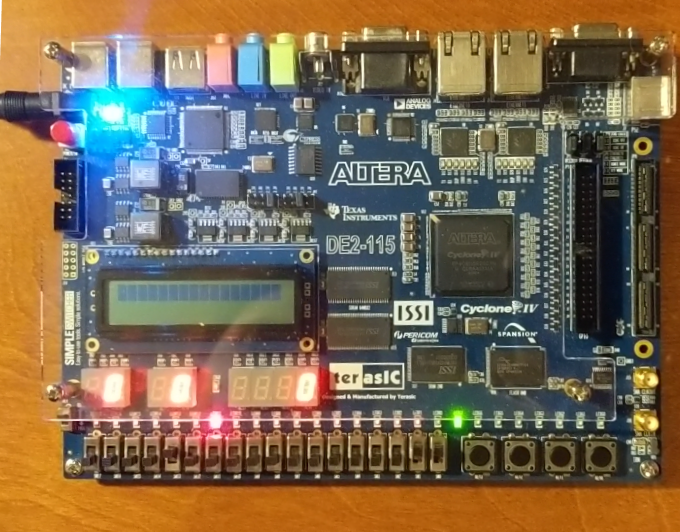


Figure : Board is logged in, indicated by the green LED on the right.

Entering the correct sequence turns the red access LED off and the green LED on. Players may at this point begin playing the game by changing toggling their switches and pressing the buttons circled red and blue in Figure 13 for player 1 and 2 respectively. Note that logging in does not update the load registers from the player switches; the load buttons must be pressed afterwards.

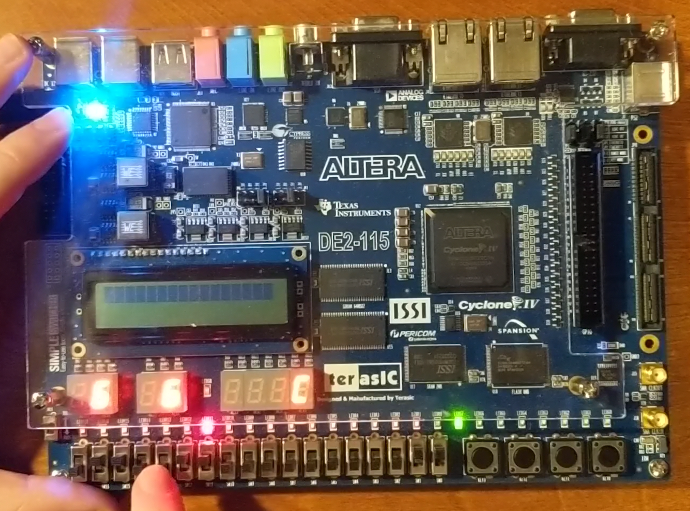


Figure : Players entering an incorrect match. The check light is still red for a result of C (12).

Now the players can begin. Entering an incorrect combination will not change the check module LEDs. Remember from Figures 10 and 12 that the only way to have the check module indicate a correct answer is to add to exactly 15.

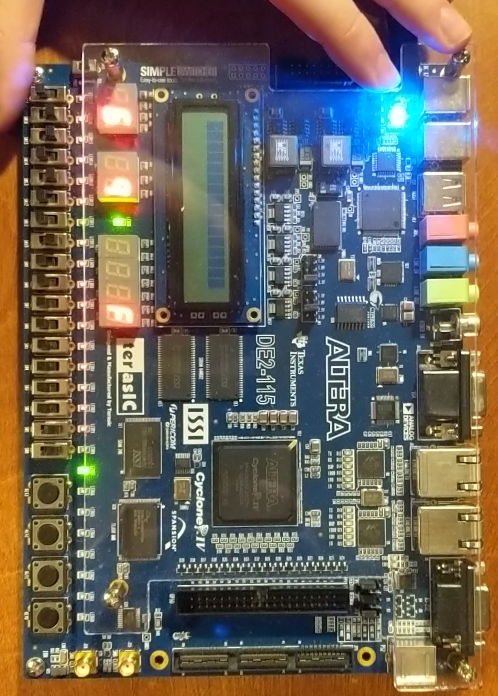


Figure : Correct result entered. Check light turns green for a result of F (15).

Here player 2 wins. The red status LED from the check module turns off and its green LED turns on.

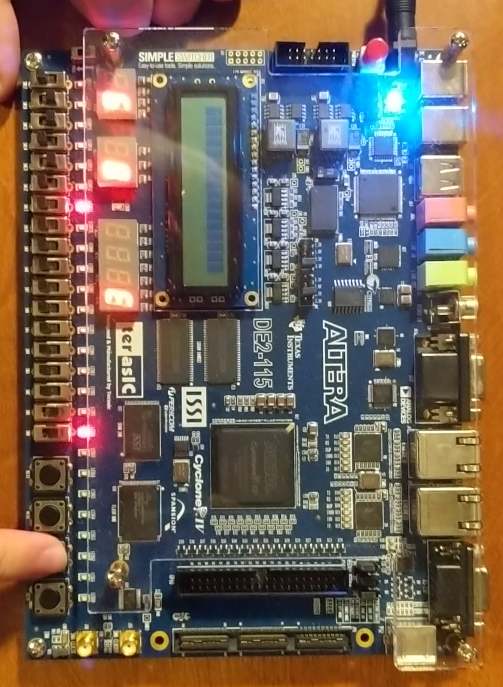


Figure : Pressing the access button again logs the system out. Players cannot update the register values.

When the players want to log out but preserve the register values, pressing the access control button will log out without resetting the whole board. If the players toggle any of their switches and presses their button, the load registers will not update.

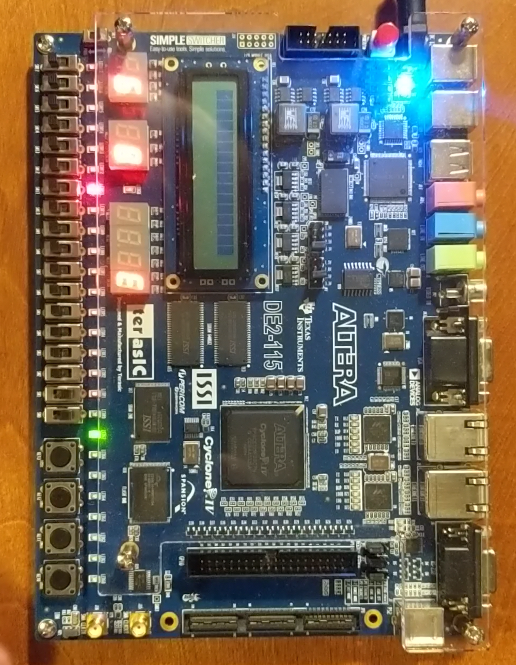


Figure : Player logs in again, with all numbers still preserved.

Upon re-entering the passcode, the system logs in and the players can continue.

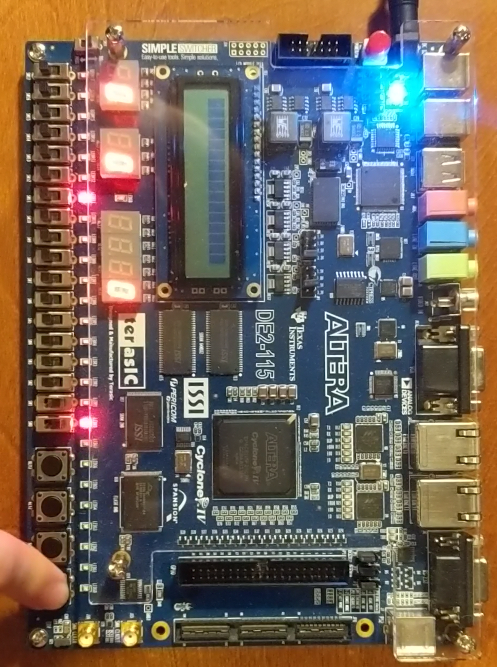


Figure : Pressing reset logs the system out and rewrites the registers back to 0.

Pressing RESET will not only log the game out, but also sets the register values to 0. Again, during logout players cannot change the load registers. Pressing reset will do the same thing as power cycling the device as the system is in the same state as it was in Figure 13.

# **5.** Conclusion

The binary math game now features a simple security system with an even simpler method of inputting numbers. At 50MHz the sequential logic appears instantaneous. All required features were successfully implemented as shown by both the simulations in chapter 3 and the video demonstration. The ability to log out via the access control button and preserving the load registers was a bonus feature that was successfully implemented as demonstrated in Figure 17.

# **5.** References

[1] DE2-115 User Manual. (2013). [ebook] Terasic. Available at: <https://www.intel.com/content/dam/altera-www/global/en_US/portal/dsn/42/doc-us-dsnbk-42-1404062209-de2-115-user-manual.pdf>.

# **I.** Appendix

Attached below is the Verilog code that was flashed onto the FPGA. Module Lab2\_BUCUR\_S is the top-level module.

module bshaper(

    CLK, RST,

    b\_in, b\_out

    );

    input b\_in, CLK, RST;

    output b\_out;

    reg b\_out;

    reg [1:0] currentstate, nextstate;

    parameter Init = 2'b00, Edge = 2'b01, Wait = 2'b10;

    always @ (posedge CLK) begin

        if(RST == 1'b0)

        begin

            currentstate <= Init;

        end

        else begin

            currentstate <= nextstate;

        end

    end

    always @ (currentstate, b\_in) begin

        case (currentstate)

            Init: begin

                b\_out <= 1'b0;

                if(b\_in == 1'b1)

                begin

                    nextstate <= Init;

                end

                else begin

                    nextstate <= Edge;

                end

            end

            Edge: begin

                b\_out <= 1'b1;

                nextstate <= Wait;

            end

            Wait: begin

                b\_out <= 1'b0;

                if(b\_in == 1'b1)

                begin

                    nextstate <= Init;

                end

                else begin

                    nextstate <= Wait;

                end

            end

            default: begin

                nextstate <= Init;

            end

        endcase

    end

endmodule

module loadreg(

    CLK, RST,

    data\_in, sig\_load,

    data\_out

    );

    input CLK, RST, sig\_load;

    input [3:0] data\_in;

    output [3:0] data\_out;

    reg [3:0] data\_out;

    reg loadreg\_state;

    parameter Store = 1'b0, Load = 1'b1;

    always @ (posedge CLK) begin

        if (RST == 0)

        begin

            loadreg\_state <= Store;

            data\_out <= 4'b0000;

        end

        else begin case (loadreg\_state)

                Store: begin

                    if(sig\_load == 1'b1)

                    begin

                        loadreg\_state <= Load;

                        data\_out <= data\_in;

                    end

                end

                Load: begin

                    if(sig\_load == 1'b0)

                    begin

                        loadreg\_state <= Store;

                    end

                    if(sig\_load == 1'b1)

                    begin

                        loadreg\_state <= Load;

                        data\_out <= data\_in;

                    end

                end

                default: begin loadreg\_state <= Store; end

            endcase

        end

    end

endmodule

odule access (

    CLK, RST,

    loadreg\_1\_in, loadreg\_2\_in,

    pword, pword\_enter,

    loadreg\_1\_out, loadreg\_2\_out,

    pass\_red, pass\_green, currentstate

    );

    input CLK, RST;

    input loadreg\_1\_in, loadreg\_2\_in;

    input [3:0] pword;

    input pword\_enter;

    output loadreg\_1\_out, loadreg\_2\_out;

    output pass\_red, pass\_green;

    output [2:0] currentstate;

    reg loadreg\_1\_out, loadreg\_2\_out;

    reg pass\_red, pass\_green;

    reg [2:0] currentstate, nextstate;

    reg pass\_OK;

    parameter Digit\_1 = 3'b001, Digit\_2 = 3'b010;

    parameter Digit\_3 = 3'b011, Digit\_4 = 3'b100, OK = 3'b111;

    always @ (posedge CLK) begin

if (RST == 1'b0) begin

            currentstate <= Digit\_1;

            pass\_OK <= 1'b1;

            pass\_red <= 1'b1;

            pass\_green <= 1'b0;

            loadreg\_1\_out <= 1'b0;

            loadreg\_2\_out <= 1'b0;

        end

        else begin case (currentstate)

                Digit\_1: begin

                    pass\_OK <= 1'b1;

                    if(pword\_enter == 1'b0)

                    begin

                        pass\_red <= 1'b1;

                        pass\_green <= 1'b0;

                        loadreg\_1\_out <= 1'b0;

                        loadreg\_2\_out <= 1'b0;

                        currentstate <= Digit\_1;

                    end

                    else begin

                        if(pword !== 4'b0011) //3

                        begin

                            pass\_OK <= 1'b0;

                        end

                        currentstate <= Digit\_2;

                    end

                end

                Digit\_2: begin

                    if(pword\_enter == 1'b0)

                    begin

                        pass\_red <= 1'b1;

                        pass\_green <= 1'b0;

                        loadreg\_1\_out <= 1'b0;

                        loadreg\_2\_out <= 1'b0;

                        currentstate <= Digit\_2;

                    end

                    else begin

                        if(pword !== 4'b0001) //1

                        begin

                            pass\_OK <= 1'b0;

                        end

                        currentstate <= Digit\_3;

                    end

                end

                Digit\_3: begin

                    if(pword\_enter == 1'b0)

                    begin

                        pass\_red <= 1'b1;

                        pass\_green <= 1'b0;

                        loadreg\_1\_out <= 1'b0;

                        loadreg\_2\_out <= 1'b0;

                        currentstate <= Digit\_3;

                    end

                    else begin

                        if(pword !== 4'b0101) //5

                        begin

                            pass\_OK <= 1'b0;

                        end

                        currentstate <= Digit\_4;

                    end

                end

                Digit\_4: begin

                    if(pword\_enter == 1'b0)

                    begin

                        pass\_red <= 1'b1;

                        pass\_green <= 1'b0;

                        loadreg\_1\_out <= 1'b0;

                        loadreg\_2\_out <= 1'b0;

                        currentstate <= Digit\_4;

                    end

                    else begin

                        if(pword !== 4'b0011) //3

                        begin

                            pass\_OK <= 1'b0;

                        end

                        else begin

                            if(pass\_OK == 1'b1)

                            begin

                                currentstate <= OK;

                                pass\_OK <= 1'b1;

                            end

                            else begin

                                currentstate <= Digit\_1;

                            end

                        end

                    end

                end

                OK: begin

                    pass\_red <= 1'b0;

                    pass\_green <= 1'b1;

                    loadreg\_1\_out <= loadreg\_1\_in;

                    loadreg\_2\_out <= loadreg\_2\_in;

                    if(pword\_enter == 1'b1)

                    begin

                        currentstate <= Digit\_1;

                    end

                    else begin

                        currentstate <= OK;

                    end

                end

                default: begin currentstate <= Digit\_1; end

            endcase

        end

    end

endmodule

module adder (in1, in2, sum);

    input [3:0] in1, in2;

    output [3:0] sum;

    reg [3:0] sum;

    always @ (in1, in2)

        begin

            sum = in1 + in2;

        end

endmodule

module decoder7 (hex, seg7);

    input [3:0] hex;

    output [6:0] seg7;

    reg [6:0] seg7;

    always @ (hex)

    begin

        case(hex)

            4'b0000: begin seg7 = 7'b1000000; end

            4'b0001: begin seg7 = 7'b1111001; end

            4'b0010: begin seg7 = 7'b0100100; end

            4'b0011: begin seg7 = 7'b0110000; end

            4'b0100: begin seg7 = 7'b0011001; end

            4'b0101: begin seg7 = 7'b0010010; end

            4'b0110: begin seg7 = 7'b0000010; end

            4'b0111: begin seg7 = 7'b1111000; end

            4'b1000: begin seg7 = 7'b0000000; end

            4'b1001: begin seg7 = 7'b0011000; end

            4'b1010: begin seg7 = 7'b0001000; end

            4'b1011: begin seg7 = 7'b0000011; end

            4'b1100: begin seg7 = 7'b1000110; end

            4'b1101: begin seg7 = 7'b0100001; end

            4'b1110: begin seg7 = 7'b0000110; end

            4'b1111: begin seg7 = 7'b0001110; end

        endcase

    end

endmodule

module check(

    sum,

    status);

    input [3:0] sum;

    output [1:0] status;

    reg [1:0] status;

    always @(sum)

    begin

        if( sum == 4'b1111 )

        begin

            status = 2'b01;

        end

        else

        begin

            status = 2'b10;

        end

    end

endmodule

module Lab2\_BUCUR\_S(

    CLOCK, RESET,

    p1\_data, p2\_data, acc\_data,

    p1\_sel, p2\_sel, acc\_sel,

    p1\_digit, p2\_digit, sum\_digit,

    sum\_status, pass\_r, pass\_g

    );

    input CLOCK, RESET;

    input [3:0] p1\_data, p2\_data, acc\_data;

    input p1\_sel, p2\_sel, acc\_sel;

    output [6:0] p1\_digit, p2\_digit, sum\_digit;

    output [1:0] sum\_status;

    output pass\_r, pass\_g;

    wire [2:0] acc\_state;

    wire p1\_bpress, p2\_bpress, acc\_bpress;

    wire p1\_acc, p2\_acc;

    wire [3:0] p1\_w, p2\_w, add\_w;

    decoder7 p1\_dec(p1\_w, p1\_digit);

    decoder7 p2\_dec(p2\_w, p2\_digit);

    decoder7 sum\_dec(add\_w, sum\_digit);

    adder player\_sum(p1\_w, p2\_w, add\_w);

    check sum\_result(add\_w, sum\_status);

    loadreg p1\_num(CLOCK, RESET, p1\_data, p1\_acc, p1\_w);

    loadreg p2\_num(CLOCK, RESET, p2\_data, p2\_acc, p2\_w);

    access acc\_ctl(CLOCK, RESET,

                   p1\_bpress, p2\_bpress,

                   acc\_data, acc\_bpress,

                   p1\_acc, p2\_acc,

                   pass\_r, pass\_g, acc\_state

    );

    bshaper p1\_button(CLOCK, RESET, p1\_sel, p1\_bpress);

    bshaper p2\_button(CLOCK, RESET, p2\_sel, p2\_bpress);

    bshaper acc\_button(CLOCK, RESET, acc\_sel, acc\_bpress);

endmodule