Segment Runner
FPGA Game

Team: GravityGuyz

Faheem Quazi

Arash Shariatzadeh

Gabriel Torres

Zain Bhatti

ECE 5440

Dr. Yuhua Chen

MAY 10, 2022

Introduction

In a world of ever-changing and evolving technology, innovation becomes a key trait for engineers to thrive in this environment. One of the best ways to practice innovation is through quick prototyping and brainstorming ideas in a diverse team with different abilities and backgrounds. For instance, Field Programmable Gate Arrays (FPGAs) are considered a valuable enabler for innovation as shown in this project. To demonstrate the flexibility of FPGAs, technical ability and team harmony, a game developed entirely in an FPGA was created. The game is called Segment Runner and it was inspired by a mobile device classic, “Gravity Guy”, to which is not available nor supported anymore. This game is a tribute to the game many people, including the team, played in their early adolescence years or when smartphone use became more widespread.

Segment Runner is a game implemented on an FPGA prototype board such as the terasIC DE0-CV. Any modern FPGA could be used as the game is not very demanding on timing nor performance. A graphical depiction of the board used can be seen below in figure 1.

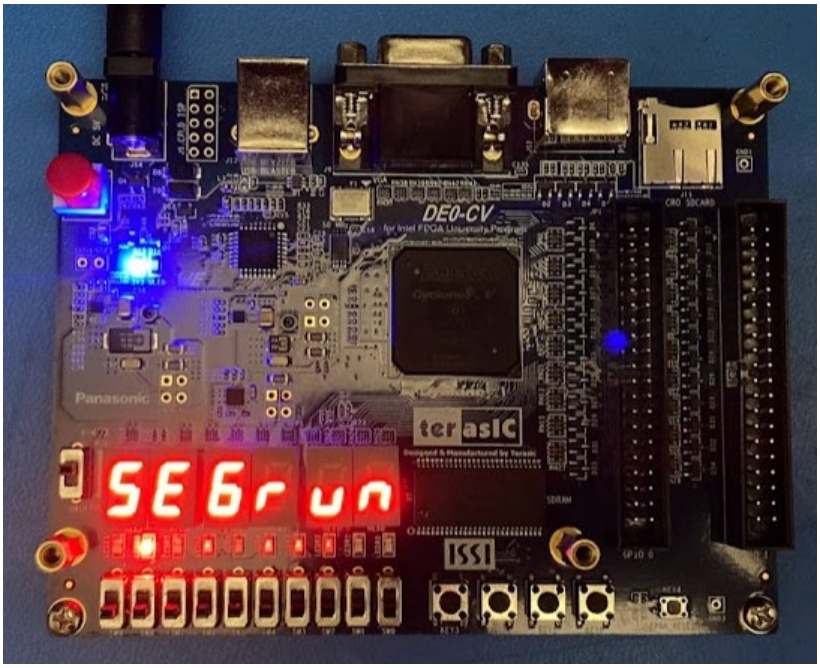
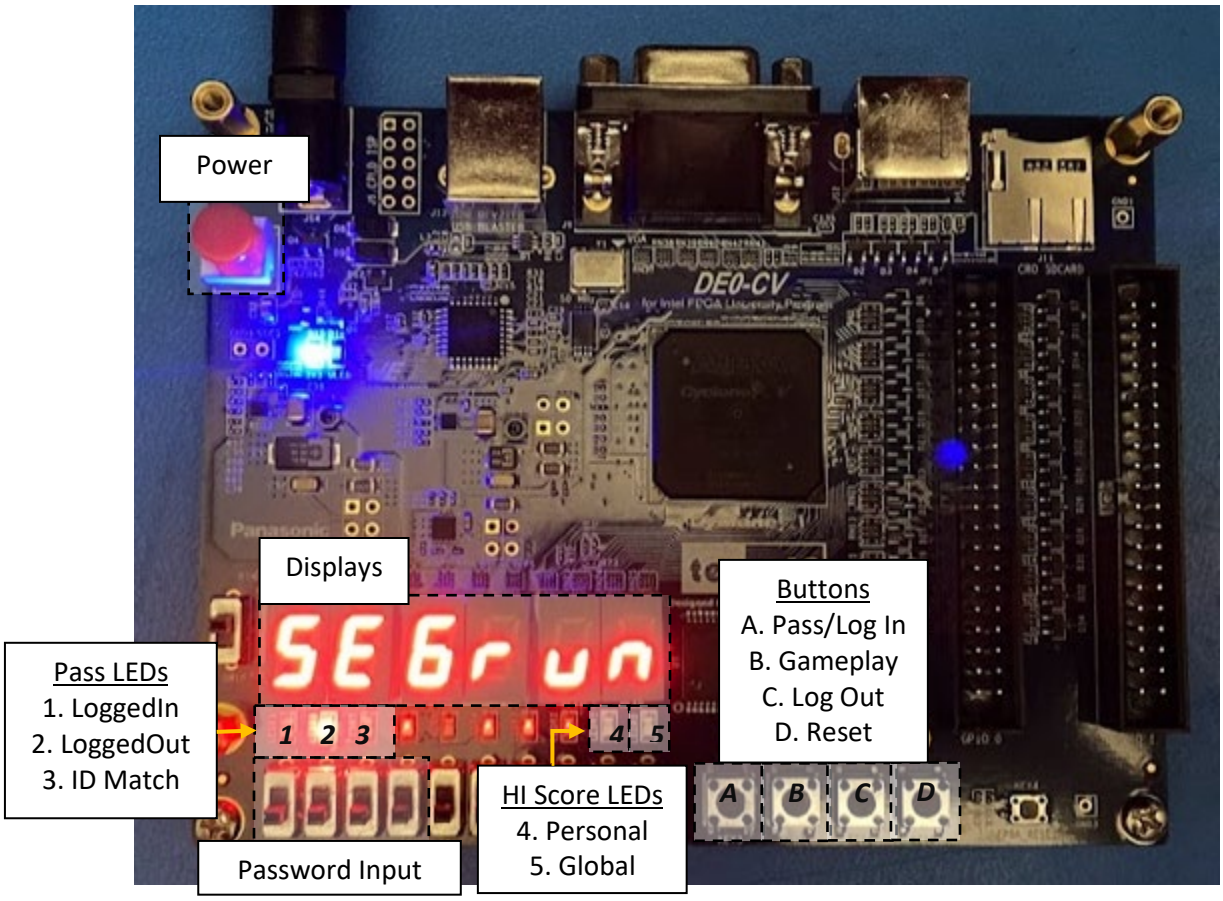


Figure 1. DE0-CV Development Board, Top View

The hardware interfaces available are an advantage of using the DE0-CV development board. As seen in the figure above, there are slide-switches, LEDs, and seven-segment displays. Segment runner will use all the seven-segment displays, LEDs, buttons and only the 4 leftmost slide-switches internally labeled SW9-SW6.

To explain operation of the system, the player must properly power the board with the included DC adapter and ensure that it is in RUN and not in PROG mode. After powering the board, the game is secured by 2-layer authentication (ID + Password). After entering the correct combination of ID and password, the user can begin playing. The seven-segment displays will depict a floor and a ceiling with a stick figure in it, the stick figure can control gravity in this game, so it can move between the floor and ceiling with the push of a button. The player must

successfully move the stick figure to prevent it from falling to a hole (depicted by an empty tile in the display) that would indicate game over. This game is technically endless. However, it has multiple levels of difficulty that makes the game faster and requires faster reflexes to achieve a high score.



System Architecture Design

The “breadboard” design is displayed in figure 2. The user interfaces can be seen, the displays, LEDs, buttons and switches. The bigger rectangle is the top-level module that encloses and instantiates four main submodules: the Authenticator, Game Controller, Score Tracker and Decoder Driver.

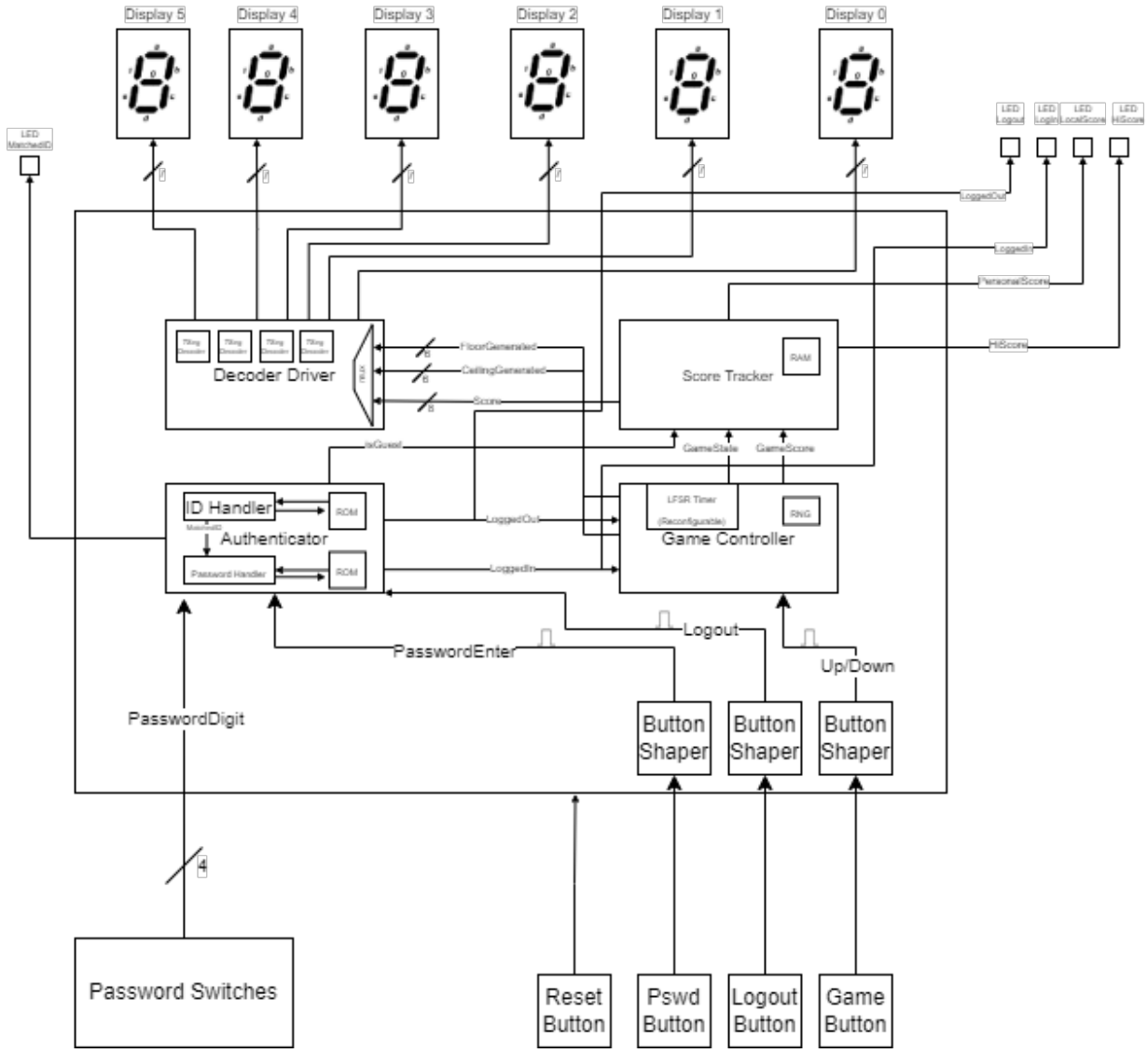


Figure 2. System Architecture Top-Level View

The netlist explorer utility from Quartus provides more details with the signal definitions as seen in figure 3. Additionally, the module hierarchy from each of the main submodules can be seen below.

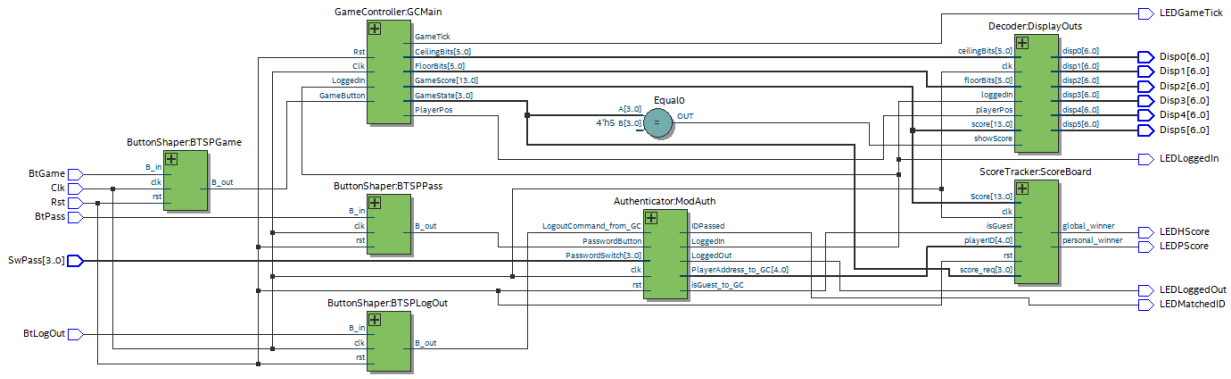
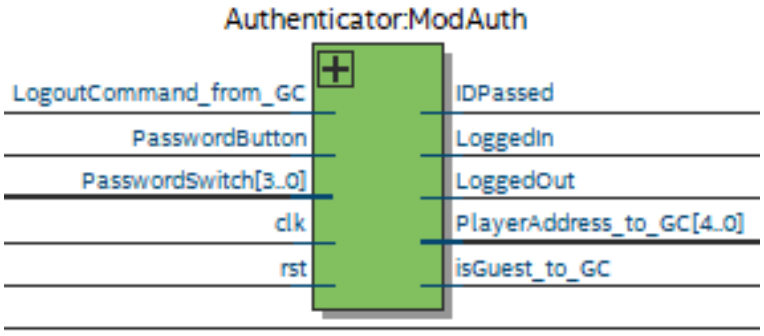


Figure 3. Quartus Netlist Diagram

The signals will be defined by function and n bits enclosed in square brackets using the notation Signal_Name [n] in the following section.

1. Authenticator Module



The Authenticator module was designed hierarchically and is comprised of the ID Handler, Password Handler, ID Handler ROM, and Password Handler ROM. The function of the authenticator is to prevent unauthorized users from accessing the game by requiring a 4-digit ID input followed by a 6-hexadecimal-digit password entered by the password switches and the password button.

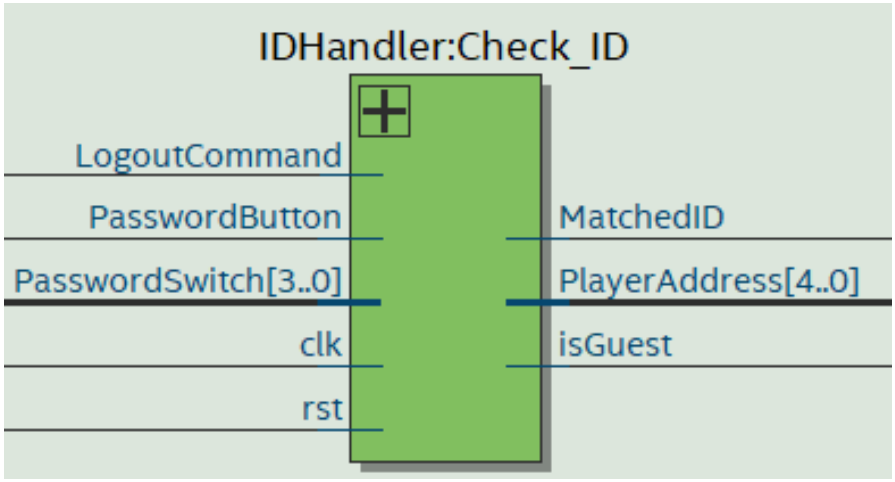
Inputs:

- LogoutCommand_from_GC[1] Active HIGH Pulse – This is the command to return to a state of authentication upon pressing the logout button. It takes a shaped button input and will prompt the user to reenter ID and password and exit the game.
- PasswordButton[1] Active HIGH Pulse – This is the shaped button input that submits the entry from the switches (Password Switch).
- PasswordSwitch[4]– This is a 4-bit bus connected straight to pins in the FPGA, providing a logic HIGH in the ‘’up” position and LOW in the “down” position. The user realizes decimal or hexadecimal digits by entering its binary equivalent with 4 bits.
- Clk [1]: Active HIGH – Main clock signal
- Rst [1]: Active LOW (Sync) – Reset button

Outputs:

- ID_Passed [1] Active HIGH Level – This signal is connected directly to Matched_ID net from the ID Handler module. However, the signal is propagated to an LED that indicates that the user passed the first layer of authentication.
- LoggedIn [1] Active HIGH Level – Connected to the LED and passed to the Game Controller. It indicates that the user passed authentication.
- LoggedOut [1] Active HIGH Level – Connected to an LED that indicates that the user is logged out.
- PlayerAddress_to_GC[5] – This is the internal designation of the player as the address of the ROM.
- isGuest_to_GC[1] – This flag indicates that the Guest ID “8888” was entered.

ID Handler



The ID Handler is the first layer of security from the authentication module. It takes the initial four digits and will check in the ROM below.

Inputs:

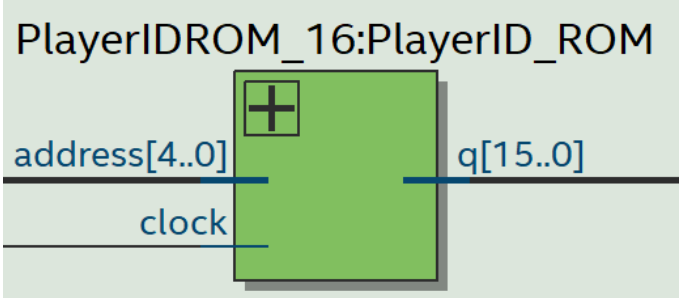
- LogoutCommand_from_GC[1] Active HIGH Pulse – This is the command to return to a state of authentication upon pressing the logout button. It takes a shaped button input and will prompt the user to reenter ID and password and exit the game.
- PasswordButton[1] Active HIGH Pulse – This is the shaped button input that submits the entry from the switches (Password Switch).
- PasswordSwitch[4] Active LOW Level – This is a 4-bit bus connected straight to pins in the FPGA, providing a logic HIGH in the ‘’up” position and LOW in the “down” position. The user realizes decimal or hexadecimal digits by entering its binary equivalent with 4 bits.
- Clk [1]: Active HIGH – Main clock signal
- Rst [1]: Active LOW (Sync) – Reset button

Outputs:

- Matched_ID [1] Active HIGH Level: This is the interface signal between the ID Handler and the Password Handler. This signal hands over the user inputs when the previous inputs match a value stored in the ID Handler ROM.

- PlayerAddress [5] – This is the address of the player that matches a result from the ROM sent to the Password Handler.

ID ROM



The ROM designed for the ID Handler module was autogenerated using the IP Catalog from Quartus. It came out to a 32-word memory, where each word is 16 bits corresponding to the 5 unique IDs for the authorized players (the creators of the project plus a guest), stored in contiguous memory locations.

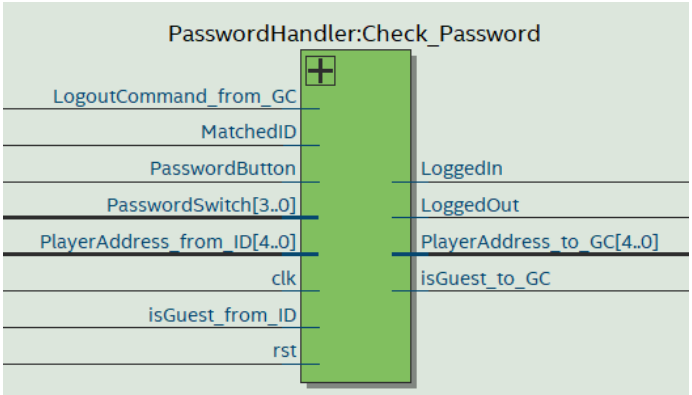
Inputs:

- Address [5] – An integer in the range 0-31 that signifies which memory location to access in the ROM. Values from 5 until 31 have been nullified since the minimum word count to generate a ROM module in Quartus is 32 and the other memory cells are unused.
- Clock [1]: Active HIGH – Main clock signal

Outputs:

- Q [16] – This is the data for an ID stored in a memory cell.

Password Handler



The password handler activates once a successful ID has been found in the ID Handler ROM and the ID Handler passed over the functionality via the interface signal. It is the second layer of security, which is designed to lock the system with a 24-bit or 6 hexadecimal digit password (0-F).

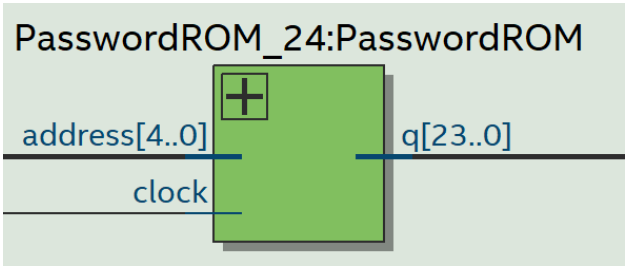
Inputs:

- LogoutCommand_from_GC[1] Active HIGH Pulse – This is the command to return to a state of authentication upon pressing the logout button. It takes a shaped button input and will prompt the user to reenter ID and password and exit the game.
- Matched_ID [1] Active HIGH Level: This is the interface signal between the ID Handler and the Password Handler. This signal hands over the user inputs when the previous inputs match a value stored in the ID Handler ROM.
- PasswordButton [1] – Same as Authentication module above.
- PasswordSwitch[4] – Same as Authentication module above.
- PlayerAddress_from_ID [5]– The address that will be sent to the Password ROM to check against the user input after passing the ID verification.
- isGuest_from_ID[1] – This flag indicates that the Guest ID “8888” was entered. When this flag is set, the password checking routine is bypassed and the player can begin playing with restricted access to score.
- Clk [1]: Active HIGH – Main clock signal
- Rst [1]: Active LOW (Sync) – Reset button

Outputs:

- LoggedIn[1] – Same as Authentication module above.
- LoggedOut[1] – Same as Authentication module above.
- PlayerAddress_to_GC[5] – Same as Authentication module above.
- isGuest_to_GC[1] – Same as Authentication module above.

Password ROM



The ROM designed for the Password Handler module was autogenerated using the IP Catalog from Quartus. It came out to a 32-word memory, where each word is 24 bits corresponding to 4 unique passwords created by the authorized players (the creators of the project and excluding the guest ID), stored in contiguous memory locations. Each memory location corresponds to the internal designation passed by the ID Handler to the Password Handler

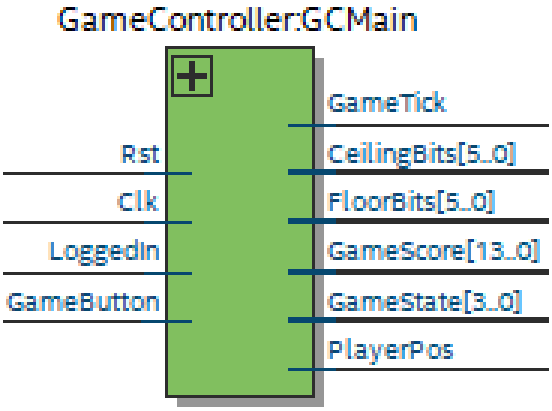
Inputs:

- Address[5] – This is the ROM memory location to be accessed given a Player ID passed by the ID Handler module. An integer in the range 0-31 that signifies which memory location to access in the ROM. Values from 5 until 31 have been nullified since the minimum word count to generate a ROM module in Quartus is 32 and the other memory cells are unused.
- Clock [1]: Active HIGH – Main clock signal

Outputs:

- Q [24] – This is the data for a Password stored in a memory cell.

2. Game Controller



The game controller was designed hierarchically and is the primary module which provides interfaces to the game. It is almost entirely self-contained, the primary inputs being a shaped GameButton and a LoggedIn signal to know whether gameplay should be allowed or not. It outputs all the required data to render the gameplay and the state of gameplay the controller is in.

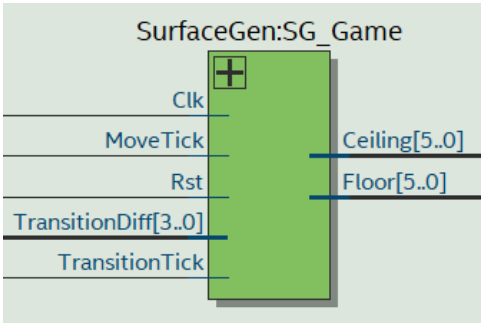
Inputs:

- Clk [1]: Active HIGH – Main clock signal
- Rst [1]: Active LOW (Sync) – Reset button
- LoggedIn [1]: Active HIGH – HIGH if gameplay should be allowed
- GameButton [1]: Active HIGH (Shaped) – Button input for gameplay interaction

Outputs:

- GameTick [1]: Active HIGH Pulse – The game internal “clock” signal; used for scoring, difficulty adjustment, and gameplay indication.
- CeilingBits [5]: Bit Array – Represents the “ceiling” segments of the game (HIGH = on/ceiling present).
- FloorBits [6]: Bit Array – Represents the “floor” segments of the game (HIGH = on/floor present).
- GameScore [14]: Unsigned Int – The current game score, max 9999, counts number of GameTick pulses until loss.
- GameState [4]: State Machine Output – Used by the Score Tracker and Decoder to determine what state the game is in
- PlayerPos [1]: Bit – LOW = Player on Floor, HIGH = Player on Ceiling.

Surface Generator



This module handles generation of the ceiling and floor bits, as well as the transition overlap between the ceiling and floor when the behavior should occur.

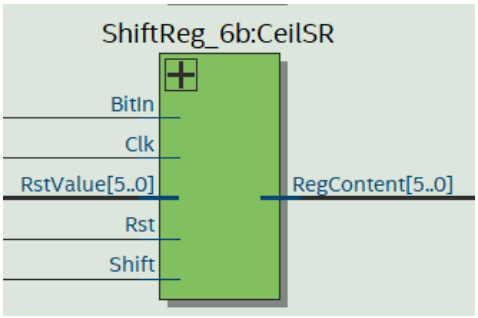
Inputs:

- MoveTick [1]: Active HIGH – GameTick clock signal; Triggers the ceiling or floor shift registers to shift
- TransitionDiff [4]: Unsigned Number – Specifies the number of tiles to overlap when a transition between ceiling and floor (or vice versa) occurs.
- TransitionTick [1]: Active HIGH – Triggers the transition from floor to ceiling or ceiling to floor
- Clk [1]: Active HIGH – Primary clock signal
- Rst [1]: Active Low (Sync) – Reset button

Outputs:

- Ceiling [5]: Bit Array – Represents the “ceiling” segments of the game (HIGH = on/ceiling present).
- Floor [6]: Bit Array – Represents the “floor” segments of the game (HIGH = on/floor present).

Shift Register



Used by the Surface Generator as the “storage” for the ceiling and floor tiles. A standard shift register with 6 bits. Unlike a normal shift register, this one does not output the bit being discarded, only the values currently inside the shift register.

Inputs:

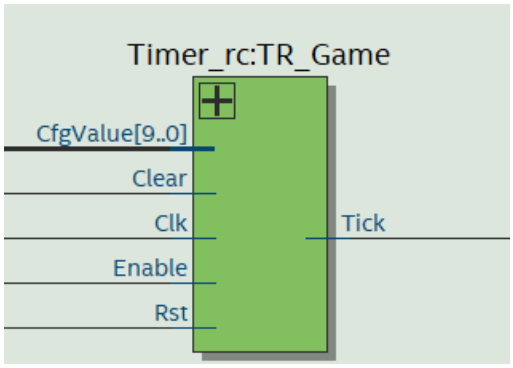
- BitIn [1]: Bit – The bit to pass into the shift register when shifting
- RstValue [6]: Bit Array – The values to initialize the internal registers to when resetting

- Shift [1]: Active HIGH – Triggers a shift of the values in the internal registers, discarding the highest bit and adding in the lowest bit based on BitIn's value.
- Clk [1]: Active HIGH – Primary clock signal
- Rst [1]: Active Low (Sync) – Reset button

Outputs:

- RegContent [6]: Bit Array – The current values inside the shift register

Reconfigurable Timer



Utilizes the 1ms timer to variably count milliseconds of elapsed time. This provides the basis for the dynamic speed changes in the game.

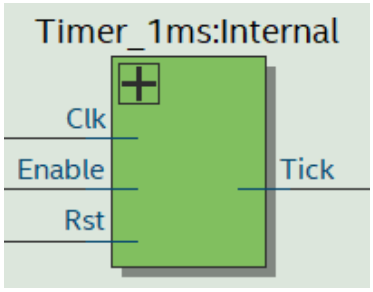
Inputs:

- CfgValue [10]: Unsigned Number – The number of milliseconds to count down to. Value can be updated dynamically.
- Clear [1]: Active HIGH – Set HIGH to clear the timer internal counter
- Clk [1]: Active HIGH – Primary clock signal
- Enable [1]: Active HIGH – Set HIGH to enable the timer
- Rst [1]: Active Low (Sync) – Reset button

Outputs:

- Tick [1]: Active HIGH Pulse – Will tick for a single pulse after CfgValue milliseconds have elapsed.

1ms Timer



Uses an LFSR-based counter to exactly count 1ms worth of clock cycles.

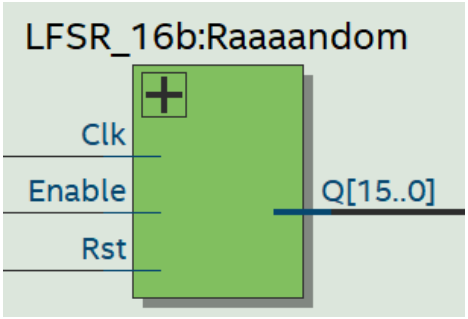
Inputs:

- Clk [1]: Active HIGH – Primary clock signal
- Rst [1]: Active Low (Sync) – Reset button
- Enable [1]: Active HIGH – Set HIGH to enable the timer

Outputs:

- Tick [1]: Active HIGH Pulse – Will tick for a single pulse after one millisecond has elapsed.

LFSR Pseudo-random Number Generator (RNG)



Used as the counter for the 1ms Timer and as a random number generator, this is a slightly modified LFSR which includes an “Enable” bit to allow easy disabling of the shift register.

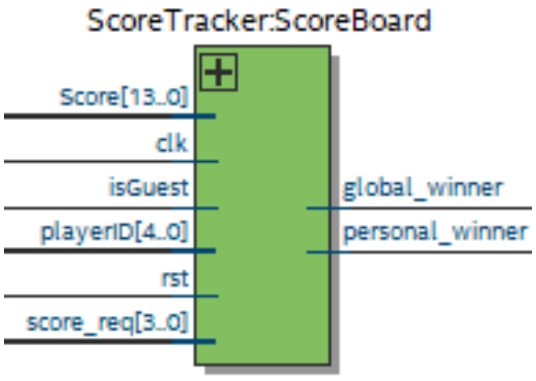
Inputs:

- Clk [1]: Active HIGH – Primary clock signal
- Rst [1]: Active Low (Sync) – Reset button
- Enable [1]: Active HIGH – Set HIGH to enable the LFSR operation

Outputs:

- Q [16]: Number – The current value in the LFSR

3. Score Tracker



The Score Tracker module was designed hierarchically, containing an instance of Score RAM. This Score Tracker is used to determine if a player’s score after each game is higher than their personal high score by accessing the Score RAM or higher than the global high score using an internal register.

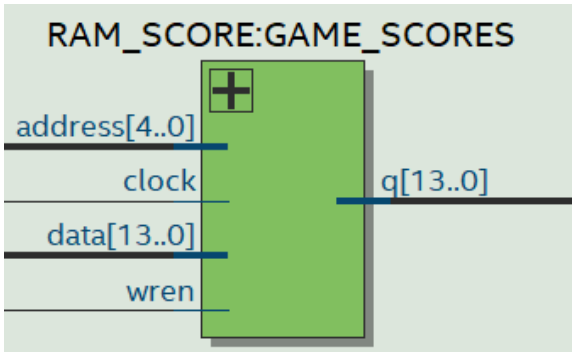
Inputs:

- Input Name [#bits]: Input Type (Active LOW/HIGH) – Purpose/Description
- clk [1]: Clock used for all sequential circuit modules
- rst [1]: Active LOW – Reset signal used to reconfigure all modules
- isGuest [1]: Signal used to indicate if user is a guest or logged in player
- playerID [5]: Bits used as internal player ID for each player if logged in that served as address for ROM/RAM modules
- score_req [4]: Used as activation signal to begin score fetching procedure
- Score [14]: Set of bits containing player's score from their latest game

Outputs:

- Output Name [#bits]: Output Type – Purpose/Description
- global_winner [1]: Signal used to drive an onboard LED to ON/OFF based on if score Is a new global high score or not respectively
- personal_winner [1]: Signal used to drive an onboard LED to ON/OFF based on if score Is a new personal high score or not respectively

Score RAM



RAM module used to store each player's personal high score. Accessed by Score Tracking module for use in determining if player has made new personal high score. Old personal best scores are overwritten by Score Tracking module when new personal best score is made.

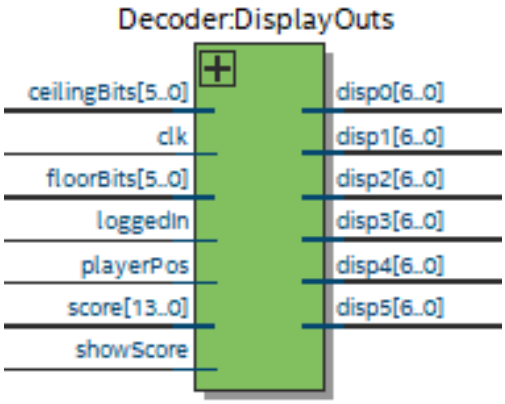
Inputs:

- Input Name [#bits]: Input Type (Active LOW/HIGH) – Purpose/Description
- clock [1]: Clock used for all sequential circuit modules
- address [5]: Address used to access each player's personal high score
- data [14]: Bits containing player's new personal high score to overwrite old high score
- wren [1]: Signal used to choose RAM mode; write is selected if signal is high and read is selected when signal is low

Outputs:

- Output Name [#bits]: Output Type – Purpose/Description
- q [14]: Data stored in cell indicated by address input signal

4. Decoder Driver



Used to handle gameplay display and score display

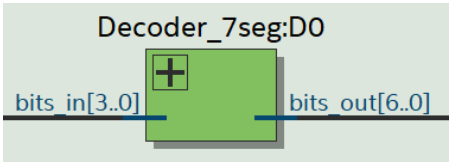
Inputs:

- ceilingBits [6]: Ceiling bits to be displayed on screen
- floorBits [6]: Floor bits to be displayed on screen
- clk [1]: Clock used for all sequential circuit modules
- playerPos [1]: Determines whether player is down (LOW) or player is up (HIGH)
- loggedIn [1]: HIGH if player logged in (show score or gameplay), LOW to display "SEGrun"
- showScore [1]: HIGH to show score, LOW for gameplay

Outputs:

- disp# [7]: Six 7-bit signals to drive displays. # is a number from 0 to 5 with 5 being the leftmost display and 0 being the rightmost.

Decoder_7seg



Converts a 4-bit binary number to a corresponding 7-bit output to display the input number on a 7-segment display

Inputs:

- bits_in [4]: 4-bit input in binary to be displayed as a decimal value

Outputs:

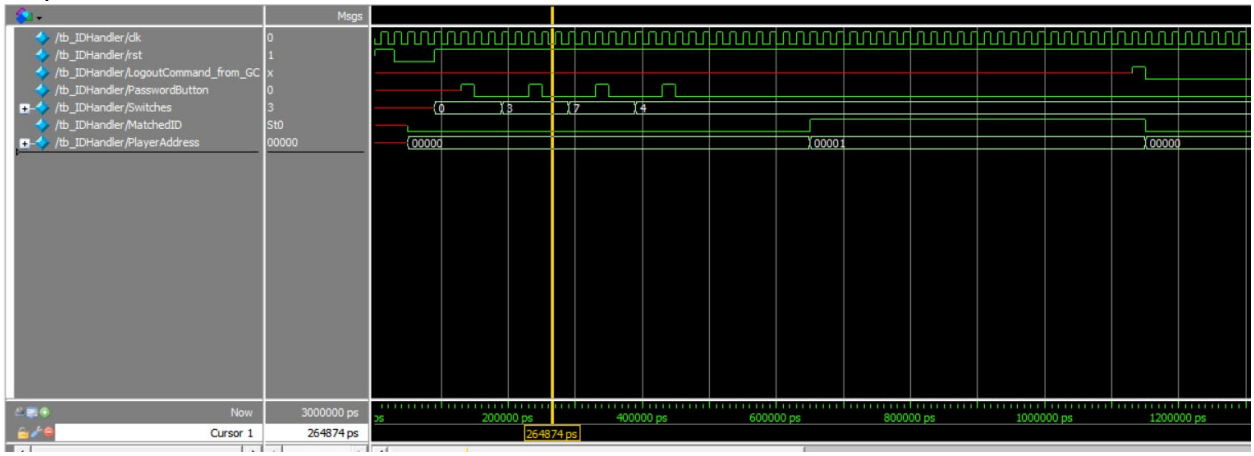
- bits_out [7]: 7-bit output to drive displays to show input number as a decimal value

Simulation Results

Authenticator

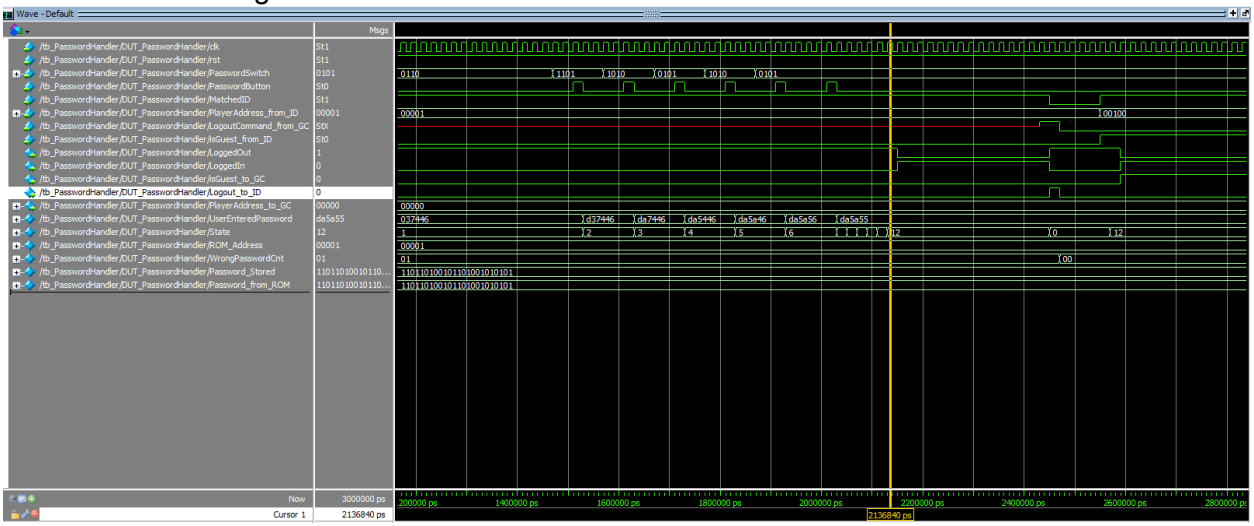
The authenticator was tested in two standalone runs, one for the ID Handler submodule and one for the Password Handler submodule to allow for portability in future projects.

- ID Handler: This simulation tests the capability of checking an ID successfully. The second authorized player, Zain, was virtually in the simulator and it was found that after scanning in the ROM, the correct ID was found, and the address was passed as an output to be included in the Password handler.



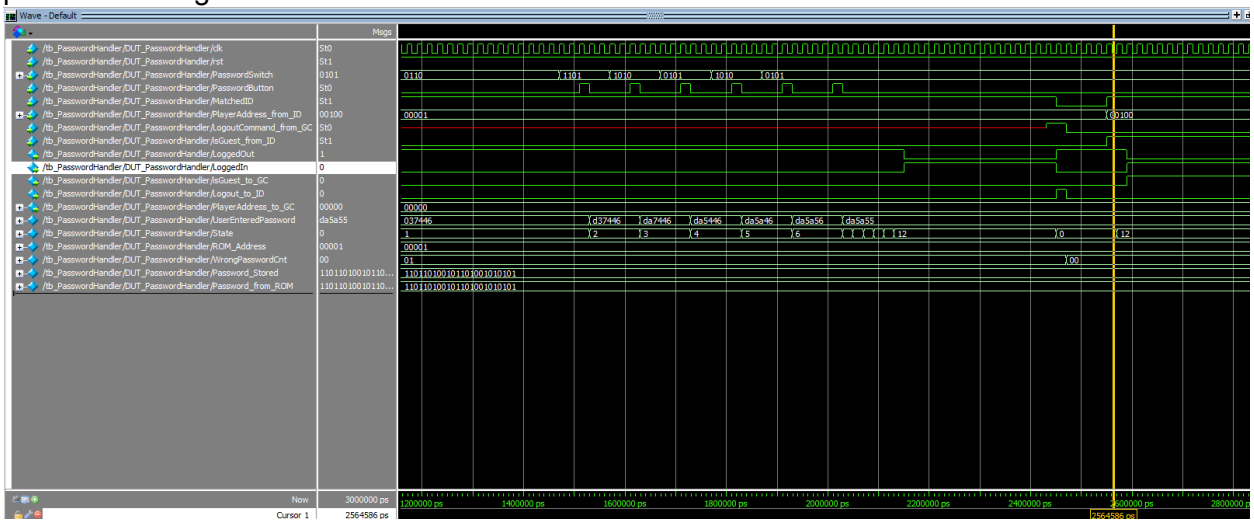
Therefore, the ID handler behaves as expected and the test was considered a success. The ROM was not tested since it is an already verified module, similar to the button shaper that produces the clean pulse PasswordButton inputs.

- Password Handler: This module was tested differently since it has a more complex control flow. First the case was tested where the correct password was entered as shown in the next figure.



The Password Handler behaves as expected. It does not activate until the MatchedID flag is set and responds to the correct password input and log out command from the Game Controller. However, it was also important to test the response to a wrong

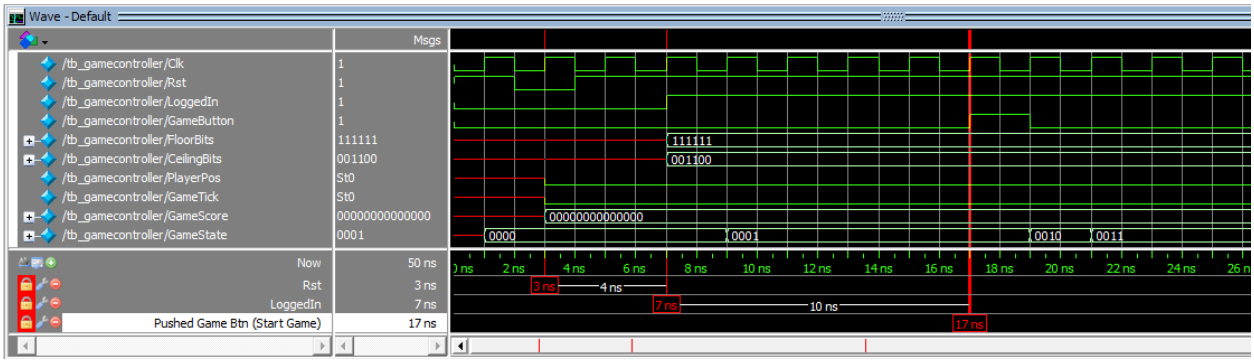
password and guest ID.

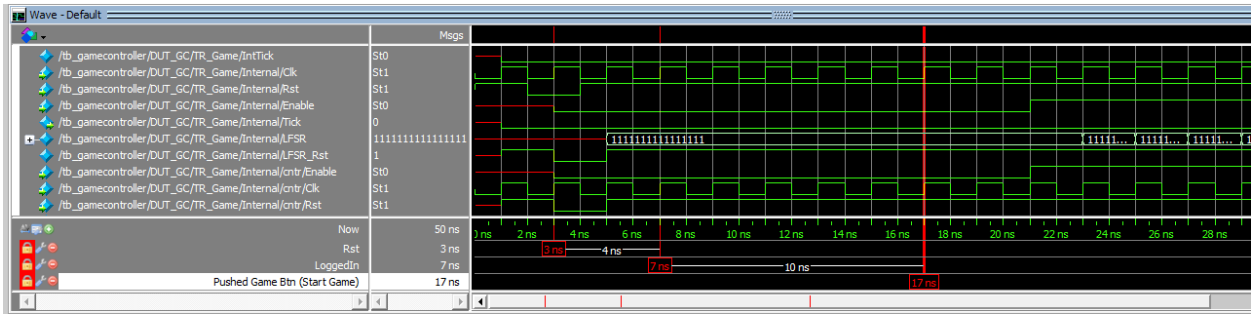


The cases for wrong password and Guest ID work as expected, the module was brought back to the standby state until the guest flag was set. The behavior is correct and it can be further integrated into the system.

Considering the results above, the authenticator module seems ready to be integrated with the rest of the code to be downloaded to the board.

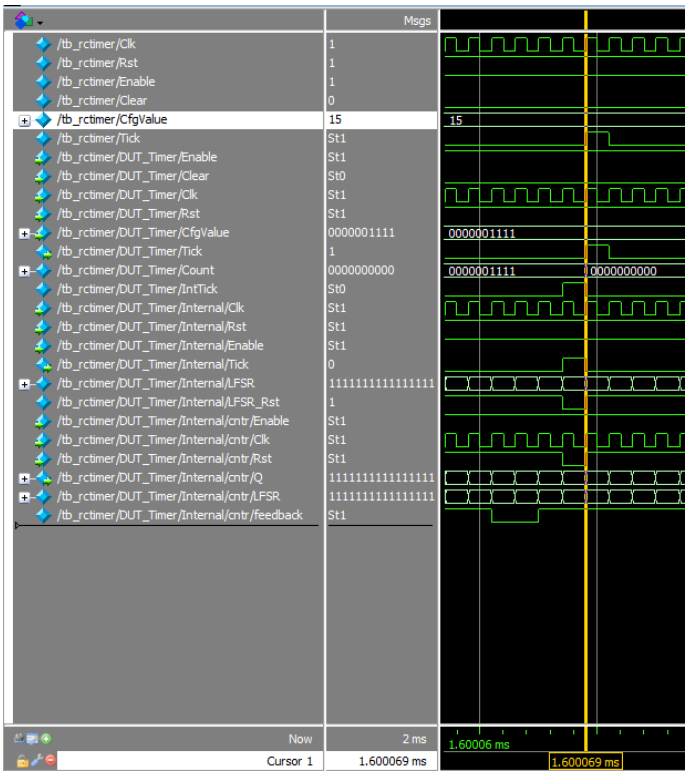
Game Controller



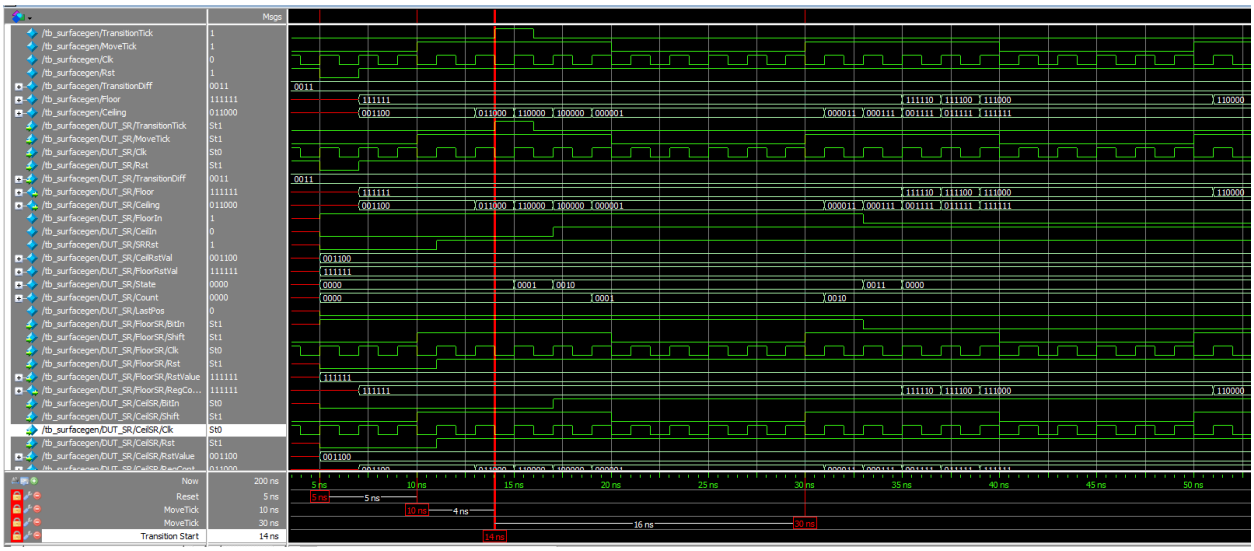


In this simulation, we can see the game operation on initialization and game start. The output is as expected, namely that the internal reconfigurable timer gets enabled, and game state gets set to “Gameplay” (0011), which will wait on an internal timer tick from the reconfigurable timer before shifting or transitioning (or doing any gameplay behavior). Note that the Floor and Ceiling

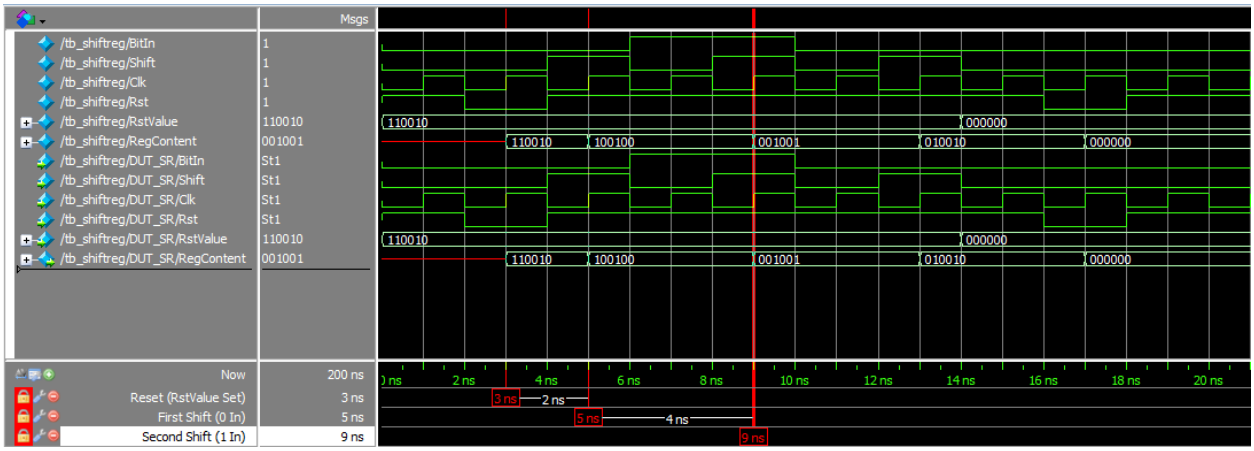
tiles don't get set (tri-stated/RED lines) until after LoggedOn gets set because the reset line for their shift register modules is controlled internally and not tied to the actual reset button. Full testing of the game controller module was done on-board rather than in the simulation since this module's behavior is tied to real-time operation (would be almost 1-3 seconds of simulation, which takes a lot longer than 1-3 seconds on the FPGA). The on-board results can be found below.



In this simulation, we can see the reconfigurable timer operating based on the input value (15 in this case). Since the clock period in the sim is 2ns, the expected time to run is 10x faster than on the actual FPGA, so for a 15 ms CfgValue, we should expect around 1.5ms in the simulation. We see here the result is 1.6ms, but this drift is imperceptible to the human eye, so we have settled that this is satisfactory for game operation.

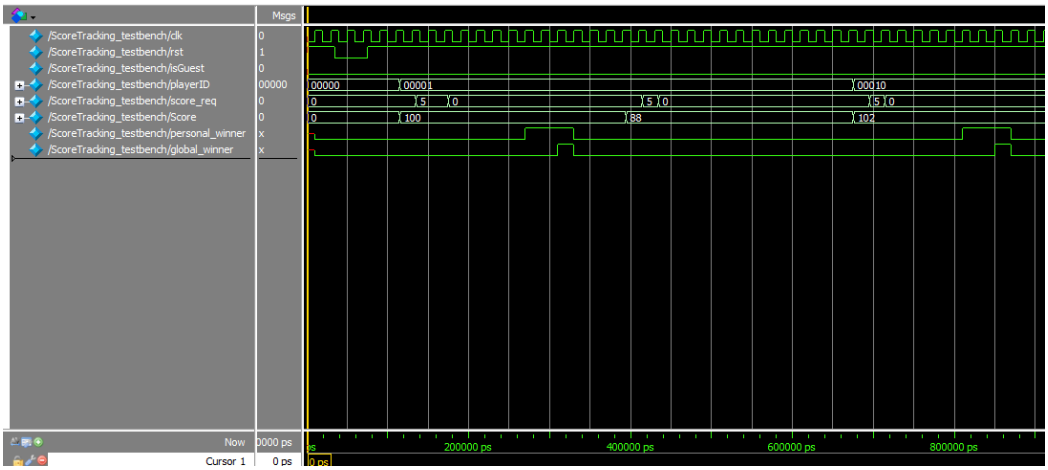


In this simulation, we can see the Surface Generator correctly shifting values and outputting the intended behavior for a transitional tick. The transition only happens *after* a couple clock cycle delay (see the state machine output DUT_SR/State), and the shifts only happen when MoveTick is HIGH.



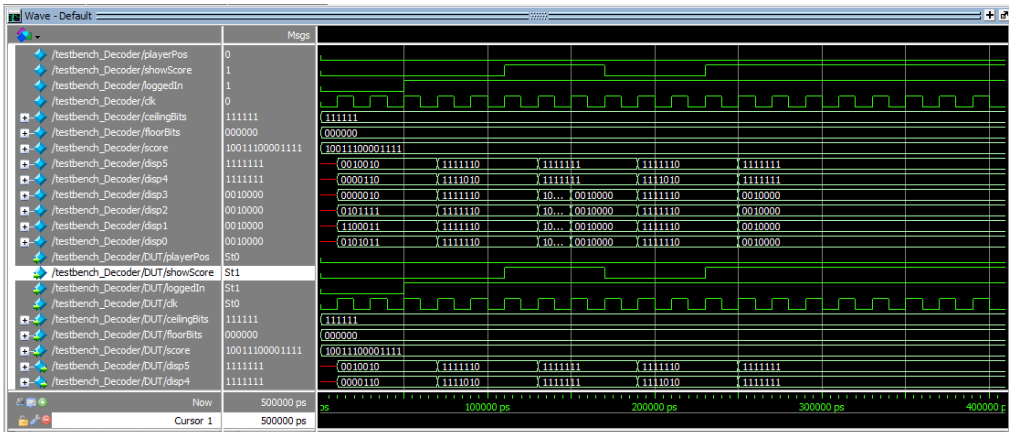
In this simulation, we can see the shift register behaving as expected on reset and shift, varying based on the input BitIn field and resetting based on the value in RstValue.

Score Tracker



When simulating the Score Tracker module, we begin by setting the input signals to initial values and simulate a press of the reset button to reconfigure the module to its correct initial state. After this, the playerID, score, and score_req signals are set to see if the first score correctly registers as a personal and global high score. We see the personal_winner and global_winner signals become high indicating the module behaved correctly. The score is then changed to a lower value, and we see that the personal_winner and global_winner are low as this not a high score. Finally, a new playerID is entered with a score that should be the global high score. Again, the module functions properly as the personal_winner and global_winner signals are set to high as expected.

Decoder Driver



The decoder correctly outputs the six 7-bit signals for the displays to show the word “SEGrun” when loggedIn = 0, outputs the six 7-bit signals for the displays for the gameplay when loggedIn = 1 and showScore = 0, and correctly outputs the signals to display the score 9999 when score = 9999, loggedIn = 1, and showScore = 1.

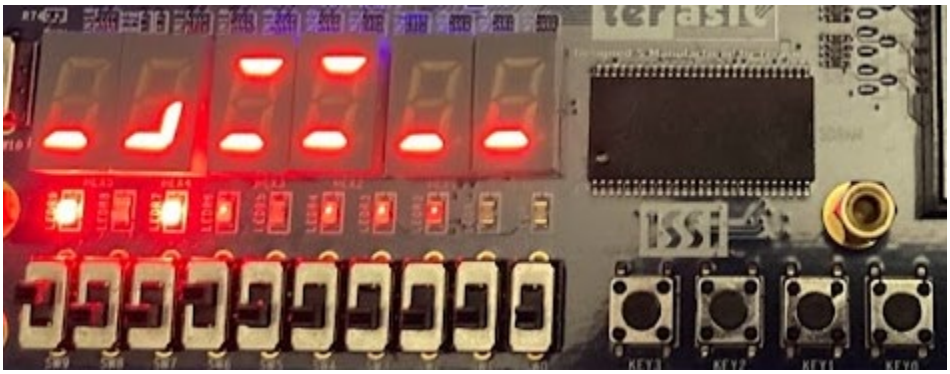
FPGA Board Testing Results

Side note: For all screenshots, the switches are left in their last position, this is a simple way to identify that the user did change as no users have the same final digit in these testing runs.

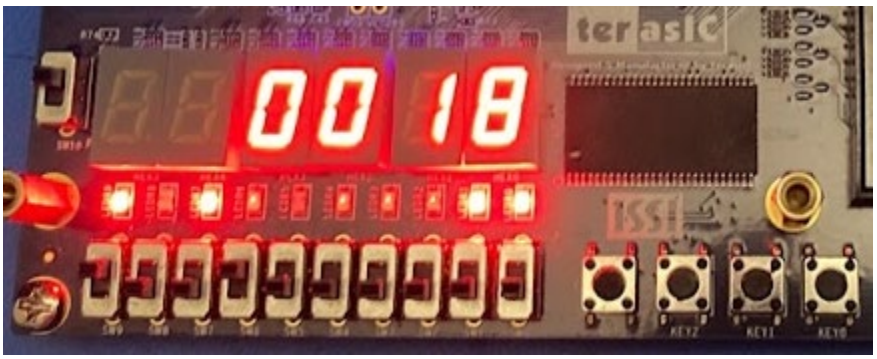
Upon first startup and/or reset, the word “SegRun” is displayed and the LoggedOut LED is illuminated.

Once a correct user ID is inputted (except the guest ID), the MatchedID LED illuminates, and the LoggedOut LED stays illuminated. A password can now be entered for that ID.

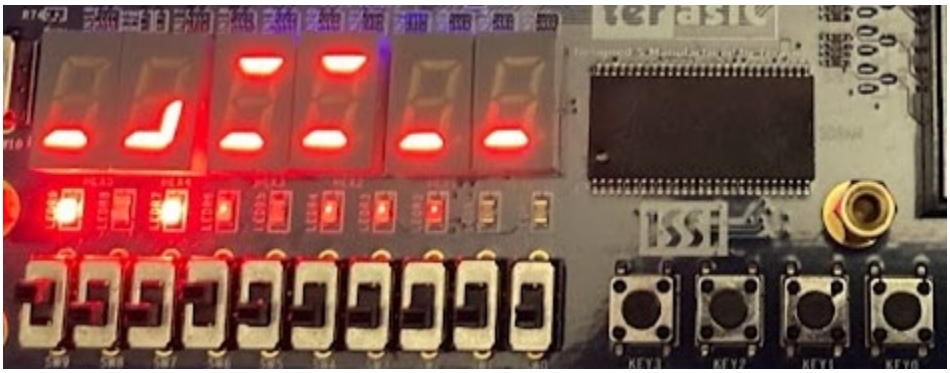
If a correct password is entered, the starting game screen will show, and the LoggedIn LED will be illuminated, LoggedOut LED turned off. At this time, the user can press the Game Button to start playing.



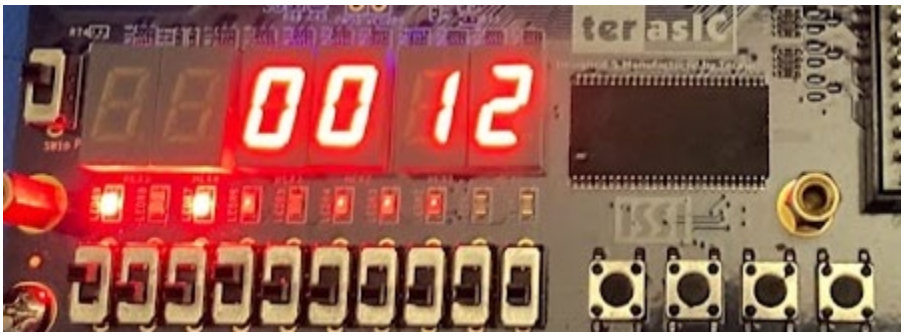
Here, a round was played, and both the personal high score and global high score was set, hence both LEDs are illuminated.



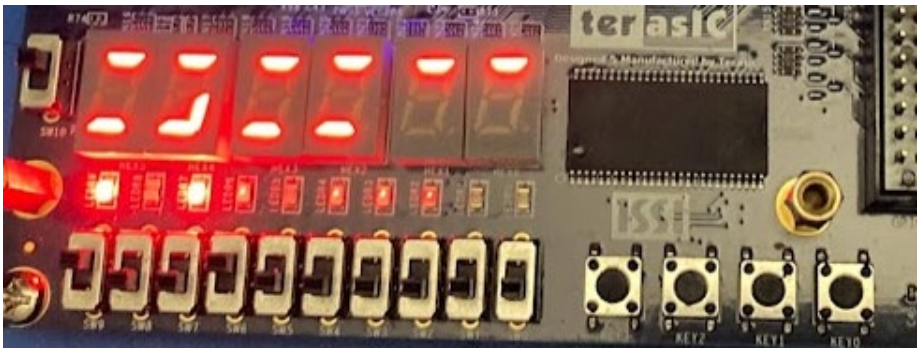
Pressing the Game Button while the score is visible returns the player to the starting game screen.



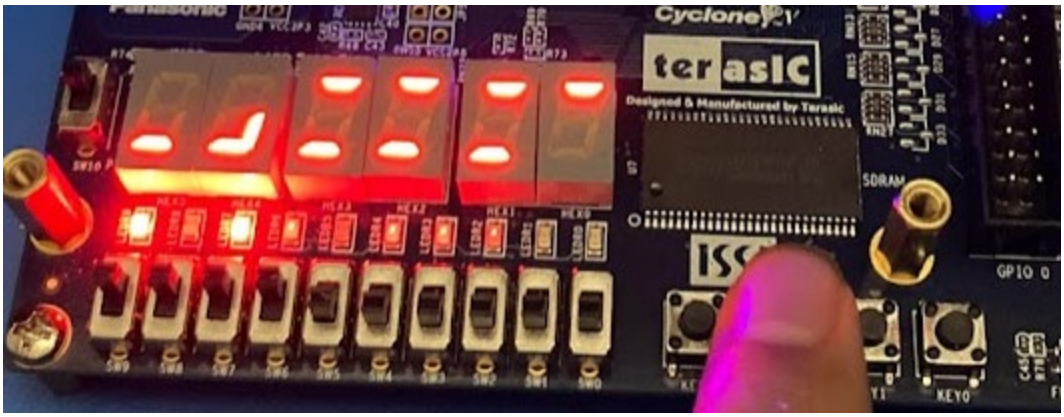
Another round was immediately played after setting that last score by the same user. Since the score is both lower than the personal high score and global high score, no score LEDs are illuminated.



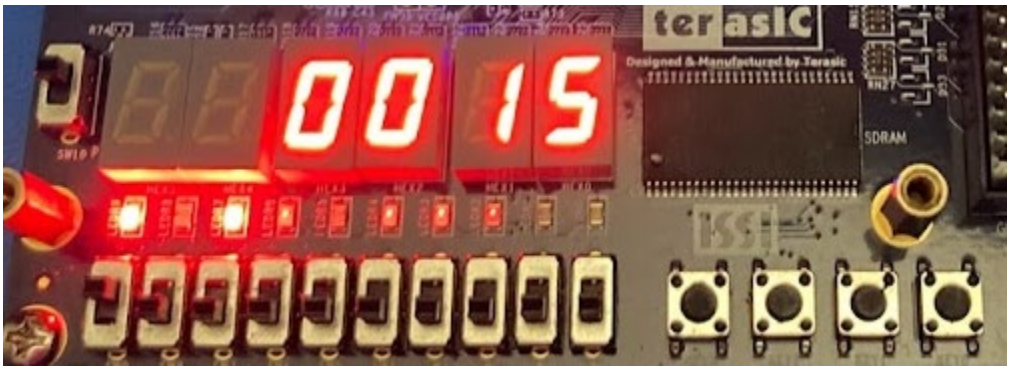
Here we can see the initial difficulty of the gameplay, with 4 overlapping segments.



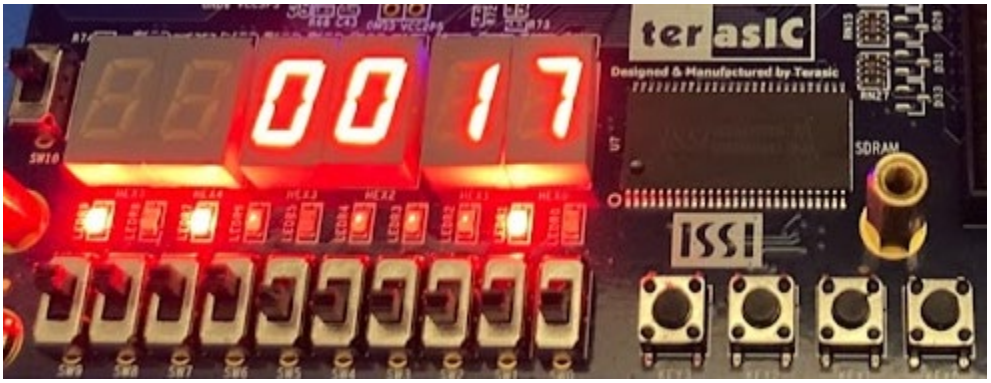
After some time has elapsed, the difficulty increases, and the overlap reduces to 3 segments.



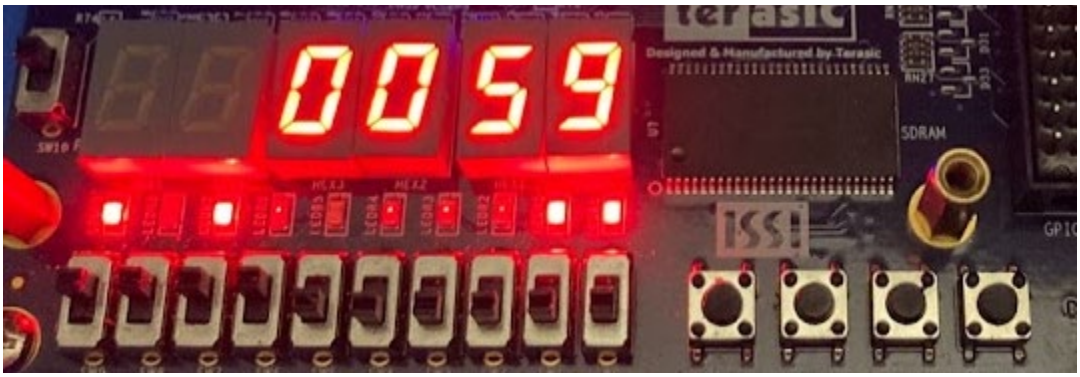
When switching to the guest account, no high score or personal score can be set. In addition, no password is required.



However, logging in as another user and setting a personal best is still possible. Here, the user set a personal best, but did not beat the global high score.



The user can play another round, and then set both their personal high score and global high score together.



Pushing the Log Out Button returns the game back to the original logged out state.

Video Demo

Authentication Demo: https://photos.app.goo.gl/J7PufhAFeo5dxM9L9

Guest Gameplay: https://photos.app.goo.gl/J7PufhAFeo5dxM9L9

User Gameplay: https://photos.app.goo.gl/J7PufhAFeo5dxM9L9
(This is the main gameplay demo)

User Guide Video: https://photos.app.goo.gl/ruB8dZ46ZVutshYZA

Conclusion

All the individual modules were able to properly function as desired, and the game is 100% playable. We were able to successfully design our own individual modules in manner such that if we wanted to change a functionality of the game or one of the other modules, it would be easy to do so without having to tamper with any other modules. All the individual modules themselves could be reused in other projects with minimal changes required to provide any desired purposes. Initially, the goal was to also implement the game using the VGA ports. However, due to timing issues and deadlines, we were unable to do so, so the goal now is to implement that in the future.

Appendix: Main Source Code

```
module Authenticator(clk, rst, PasswordSwitch, PasswordButton, LogoutCommand_from_GC, LoggedOut, LoggedIn, isGuest_to_GC, PlayerAddress_to_GC, IDPassed);

    input clk, rst;
    input [3:0] PasswordSwitch;
    input PasswordButton;
    input LogoutCommand_from_GC;

    output LoggedOut, LoggedIn;
    output isGuest_to_GC;
    output [4:0] PlayerAddress_to_GC;

    wire MatchedID;
    wire isGuest_from_ID;
    wire[4:0] PlayerAddress_from_ID;
    wire Logout_to_ID; // Log Out due to wrong password attempts exceeded

    output IDPassed;
    assign IDPassed = MatchedID;

    IDHandler Check_ID(clk, rst, PasswordSwitch, PasswordButton, LogoutCommand_from_GC, MatchedID, PlayerAddress_from_ID, isGuest_from_ID);


    PasswordHandler Check_Password(clk, rst, PasswordSwitch, PasswordButton, MatchedID, PlayerAddress_from_ID,
                        LogoutCommand_from_GC, isGuest_from_ID, LoggedOut, LoggedIn, isGuest_to_GC, Logout_to_ID, PlayerAddress_to_GC);

endmodule
```

```
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 = 2;
    reg[1:0] State, StateNext;
    // State Combinational Logic Procedure
    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 = INIT;
            end
        endcase
    end
    // State Register (Sequential)
    always@(posedge clk) begin
        if (rst == 0)
            State <= INIT;
        else
            State <= StateNext;
    end
endmodule

```

```
module Decoder_7seg(bits_in, bits_out);
    input [3:0] bits_in;
    output [6:0] bits_out;
    reg [6:0] bits_out;

    always @(bits_in) begin
        case (bits_in)
            4'b0000: begin bits_out = 7'b1000000; end // 0
            4'b0001: begin bits_out = 7'b1111001; end // 1
            4'b0010: begin bits_out = 7'b0100100; end // 2
            4'b0011: begin bits_out = 7'b0110000; end // 3
            4'b0100: begin bits_out = 7'b0011001; end // 4
            4'b0101: begin bits_out = 7'b0010010; end // 5
            4'b0110: begin bits_out = 7'b0000010; end // 6
            4'b0111: begin bits_out = 7'b1111000; end // 7
            4'b1000: begin bits_out = 7'b0000000; end // 8
            4'b1001: begin bits_out = 7'b0010000; end // 9
            4'b1010: begin bits_out = 7'b0001000; end // A
            4'b1011: begin bits_out = 7'b0000011; end // B
            4'b1100: begin bits_out = 7'b1000110; end // C
            4'b1101: begin bits_out = 7'b0100001; end // D
            4'b1110: begin bits_out = 7'b0000110; end // E
            4'b1111: begin bits_out = 7'b0001110; end // F
            default: begin bits_out = 7'b0110110; end // ≡ (unknown)
        endcase
    end
endmodule

```

```
module Decoder(clk, loggedIn, ceilingBits, floorBits, playerPos, score, showScore, disp5, disp4, disp3, disp2, disp1, disp0);
    input playerPos, showScore, loggedIn, clk;
    input [5:0] ceilingBits, floorBits;
    input [13:0] score;
    output [6:0] disp5, disp4, disp3, disp2, disp1, disp0; // disp5 is the leftmost display and disp0 is the rightmost
    reg [6:0] disp5, disp4, disp3, disp2, disp1, disp0;
    reg [3:0] seg3_in, seg2_in, seg1_in, seg0_in;
    wire [6:0] seg3_out, seg2_out, seg1_out, seg0_out;

    Decoder_7seg D3(seg3_in, seg3_out);
    Decoder_7seg D2(seg2_in, seg2_out);
    Decoder_7seg D1(seg1_in, seg1_out);
    Decoder_7seg D0(seg0_in, seg0_out);

    always @(posedge clk)
        begin
        if(loggedIn == 1'b0)
            begin
            seg3_in <= 4'b0000;
            seg2_in <= 4'b0000;
            seg1_in <= 4'b0000;
            seg0_in <= 4'b0000;
            disp5 <= 7'b0010010;
            disp4 <= 7'b0000110;
            disp3 <= 7'b0000010;
            disp2 <= 7'b0101111;
            disp1 <= 7'b1100011;
            disp0 <= 7'b0101011;
            end
        else
            begin
            if(showScore == 1'b0) //Gameplay
                begin
                disp5 <= {3'b111, ~floorBits[5], 2'b11, ~ceilingBits[5]};
                disp3 <= {3'b111, ~floorBits[3], 2'b11, ~ceilingBits[3]};
                disp2 <= {3'b111, ~floorBits[2], 2'b11, ~ceilingBits[2]};
                disp1 <= {3'b111, ~floorBits[1], 2'b11, ~ceilingBits[1]};
                disp0 <= {3'b111, ~floorBits[0], 2'b11, ~ceilingBits[0]};
                if(playerPos == 1'b0) //Player on floor
                    begin
                    disp4 <= {3'b111, ~floorBits[4], 2'b01, ~ceilingBits[4]};
                    end
                else //Player on ceiling
                    begin
                    disp4 <= {3'b111, ~floorBits[4], 2'b10, ~ceilingBits[4]};
                    end
                end
            else //Display score
                begin
                disp5 <= 7'b1111111;
                disp4 <= 7'b1111111;
                seg3_in <= (score %10000) / 1000;
                seg2_in <= (score % 1000) / 100;
                seg1_in <= (score % 100) / 10;
                seg0_in <= (score % 10);
                disp3 <= seg3_out;
                disp2 <= seg2_out;
                disp1 <= seg1_out;
                disp0 <= seg0_out;
                end
            end
        end
endmodule
```

```
module GameController(LoggedIn, GameButton, FloorBits, CeilingBits, PlayerPos, GameTick, GameState, GameScore, Clk, Rst);
    input LoggedIn, GameButton, Clk, Rst;
    output [5:0] FloorBits, CeilingBits;
    output PlayerPos, GameTick;
    output [13:0] GameScore;
    reg PlayerPos;
    reg [13:0] GameScore;

    reg [3:0] SG_TransitionDiff;
    reg SG_TransitionTick;
    reg SG_Reset;

    reg TR_Enable, TR_Clear;
    reg [9:0] TR_CfgValue;

    wire GTick;
    Timer_rc TR_Game(TR_Enable, TR_Clear, TR_CfgValue, GTick, Clk, Rst);
    SurfaceGen SG_Game(SG_TransitionTick, SG_TransitionDiff, FloorBits, CeilingBits, GTick, Clk, SG_Reset);

    // RNG
    reg [15:0] RNG_Thresh;
    reg RNG_Enable;
    wire [15:0] RNG_Q;
    LFSR_16b Raaaandom(RNG_Q, RNG_Enable, Clk, Rst);

    // FSM
    output [3:0] GameState;
    reg [3:0] GameState;
    parameter LOGGEDOUT=4'd0, GAMEWAIT=4'd1, GAMESTART=4'd2, GAMEPLAY=4'd3, GAMEJUMP=4'd4, GAMEEND=4'd5;

    always@(posedge Clk) begin
        if (Rst == 1'b0) begin
            PlayerPos <= 0;
            GameScore <= 14'd0;
            SG_TransitionDiff <= 4'd4;
            SG_TransitionTick <= 1'b0;
            TR_Enable <= 1'b0;
            TR_Clear <= 1'b0;
            TR_CfgValue <= 10'd150;
            RNG_Thresh <= 16'b1000000011111111; // approx 25%
            RNG_Enable <= 1'b1;
            SG_Reset <= 1'b0;
            GameState <= LOGGEDOUT;
        end else begin
            if (LoggedIn == 1'b0) begin
                GameState <= LOGGEDOUT;
            end
            case (GameState)
                LOGGEDOUT: begin
                    // sit idle
                    SG_TransitionDiff <= 4'd4;
                    SG_TransitionTick <= 1'b0;
                    SG_Reset <= 1'b0;
                    TR_Enable <= 1'b0;
                    TR_Clear <= 1'b0;
                    TR_CfgValue <= 10'd150;
                    if (LoggedIn == 1'b1) begin
                        GameState <= GAMEWAIT;
                    end
                end
                GAMEWAIT: begin
                    TR_Clear <= 1'b1;
                    TR_Enable <= 1'b0;
                    TR_CfgValue <= 10'd150;
                    SG_Reset <= 1'b1;
                    SG_TransitionTick <= 1'b0;
                    if (GameButton == 1'b1) begin
                        GameState <= GAMESTART;
                    end
                end
                GAMESTART: begin
                    GameScore <= 14'd0;
                    TR_Enable <= 1'b1;
                    TR_Clear <= 1'b0;
                    PlayerPos <= 1'b0;
                    SG_TransitionDiff <= 4'd4;
                    SG_TransitionTick <= 1'b0;
                    RNG_Thresh <= 16'b1000000011111111;
                    GameState <= GAMEPLAY;
                end
                GAMEPLAY: begin

                    if (GTick == 1'b1) begin
                        if (GameScore < 14'd9999) begin // cap score at 9999
                            GameScore <= GameScore + 14'd1;
                            // if (GameScore > 250) begin

                            // end
                        end

                        if (RNG_Q > RNG_Thresh) begin
                            SG_TransitionTick <= 1'b1;
                            RNG_Thresh <= 16'b1000000011111111;

                            if (TR_CfgValue > 10'd100) begin
                                TR_CfgValue <= TR_CfgValue - 1;
                            end else begin
                                if (SG_TransitionDiff > 1) begin
                                    SG_TransitionDiff <= SG_TransitionDiff - 1;
                                    TR_CfgValue <= 10'd150;
                                end else begin
                                    TR_CfgValue <= 10'd90;
                                end
                            end
                        end else begin
                            SG_TransitionTick <= 1'b0;
                            RNG_Thresh <= RNG_Thresh - 16'd50;
                        end



                        if (PlayerPos == 1'b1 && CeilingBits[4] == 1'b0) begin
                            GameState <= GAMEEND;
                        end else if (PlayerPos == 1'b0 && FloorBits[4] == 1'b0) begin
                            GameState <= GAMEEND;
                        end
                    end else begin
                        SG_TransitionTick <= 1'b0;
                    end

                    if (GameButton == 1'b1) begin
                        GameState <= GAMEJUMP;
                    end
                end
                GAMEJUMP: begin
                    if (PlayerPos == 1'b0 && CeilingBits[4] == 1'b0) begin
                        GameState <= GAMEEND;
                    end else if (PlayerPos == 1'b1 && FloorBits[4] == 1'b0) begin
                        GameState <= GAMEEND;
                    end else begin
                        GameState <= GAMEPLAY;
                        PlayerPos <= ~PlayerPos;
                    end
                end
                GAMEEND: begin
                    TR_Enable <= 1'b0;
                    if (GameButton == 1'b1) begin
                        GameState <= GAMEWAIT;
                    end
                    SG_Reset <= 1'b0;
                end
                default: begin
                    GameState <= LOGGEDOUT;
                end
            endcase
        end
    end

    assign GameTick = GTick;
endmodule
```

```
module IDHandler(clk, rst, PasswordSwitch, PasswordButton, LogoutCommand, MatchedID, PlayerAddress, isGuest);
    input clk, rst;
    input[3:0] PasswordSwitch;
    input PasswordButton, LogoutCommand;
    output MatchedID;
    output[4:0] PlayerAddress;
    output isGuest;
    reg isGuest;

    parameter DIGIT1 = 0, DIGIT2 = 1, DIGIT3 = 2,
        DIGIT4 = 3, FETCH_ROM = 4, WAIT = 5, CATCH_ROM = 6,
        COMPARE = 7, CHECK_GUEST = 8, ID_MATCH = 9;

    reg MatchedID;
    reg [4:0] PlayerAddress;
    reg[3:0] State;
    reg[1:0] WaitCnt;
    reg [15:0] UserEnteredID;



    reg [4:0] ROM_Address;
    reg[15:0] PlayerID_Stored;

    wire[15:0] PlayerID_from_ROM;

    PlayerIDROM_16 PlayerID_ROM(ROM_Address, clk, PlayerID_from_ROM);

    always @(posedge clk) begin
        if (rst == 1'b0) begin
            ROM_Address <= 0;
            MatchedID <= 1'b0;
            PlayerAddress <= 0;
            State <= DIGIT1;
            isGuest <= 0;
        end
        case(State)
        DIGIT1: begin
            if(PasswordButton == 1'b1) begin
                UserEnteredID[15:12] <= PasswordSwitch;
                State <= DIGIT2;
            end
            //ROM_Address <= 0;
        end
        DIGIT2: begin
            if(PasswordButton == 1'b1) begin
                UserEnteredID[11:8] <= PasswordSwitch;
                State <= DIGIT3;
            end
        end
        DIGIT3: begin
            if(PasswordButton == 1'b1) begin
                UserEnteredID[7:4] <= PasswordSwitch;
                State <= DIGIT4;
            end
        end
        DIGIT4: begin
            if(PasswordButton == 1'b1) begin
                UserEnteredID[3:0] <= PasswordSwitch;
                State <= FETCH_ROM;
            end
        end
        FETCH_ROM: begin
            if(ROM_Address != 0) begin // 0 address is already loaded to ROM, prevents unnecesary wait on startup
                State <= WAIT;
            end
            else begin
                State <= CATCH_ROM;
            end
        end
        WAIT: begin
            if(WaitCnt == 2) begin
                State <= CATCH_ROM;
            end
            WaitCnt <= WaitCnt + 1;
        end
        CATCH_ROM: begin
            WaitCnt <= 0;
            PlayerID_Stored <= PlayerID_from_ROM;
            State <= COMPARE;
        end
        COMPARE: begin
            if(PlayerID_Stored == UserEnteredID) begin
                State <= CHECK_GUEST;
            end
            else begin
                if(PlayerID_Stored == 16'hFFFF) begin
                    State <= DIGIT1;
                    ROM_Address <= 0;
                end
                else begin
                    ROM_Address <= ROM_Address + 1;
                    State <= FETCH_ROM;
                end
            end
        end
        CHECK_GUEST: begin
            if (PlayerID_Stored == 16'h8888) begin
                isGuest <= 1'b1;
                State <= ID_MATCH;
            end
            else begin
                isGuest <= 1'b0;
                State <= ID_MATCH;
            end
        end
        ID_MATCH: begin
            MatchedID <= 1'b1;
            PlayerAddress <= ROM_Address;
            if(LogoutCommand == 1'b1) begin
                MatchedID <= 1'b0;
                PlayerAddress <= 0;
                ROM_Address <= 0;
                State <= DIGIT1;
            end
        end
        endcase
    end
endmodule
```

```
module LFSR_16b(Q, Enable, Clk, Rst);
  input Enable, Clk, Rst;
  output [15:0] Q;

  reg [15:0] LFSR;
  wire feedback = LFSR[15];

  always @(posedge Clk) begin
    if (Rst == 1'b0) begin
      LFSR <= 16'hFFFF;
    end else if (Enable == 1'b1) begin
      LFSR[0] <= feedback;
      LFSR[1] <= LFSR[0];
      LFSR[2] <= LFSR[1] ^ feedback;
      LFSR[3] <= LFSR[2] ^ feedback;
      LFSR[4] <= LFSR[3];
      LFSR[5] <= LFSR[4] ^ feedback;
      LFSR[6] <= LFSR[5];
      LFSR[7] <= LFSR[6];
      LFSR[8] <= LFSR[7];
      LFSR[9] <= LFSR[8];
      LFSR[10] <= LFSR[9];
      LFSR[11] <= LFSR[10];
      LFSR[12] <= LFSR[11];
      LFSR[13] <= LFSR[12];
      LFSR[14] <= LFSR[13];
      LFSR[15] <= LFSR[14];
    end
  end

  assign Q = LFSR;
endmodule
```

```
module PasswordHandler(clk, rst, PasswordSwitch, PasswordButton, MatchedID, PlayerAddress_from_ID,
                       LogoutCommand_from_GC, isGuest_from_ID, LoggedOut, LoggedIn, isGuest_to_GC, Logout_to_ID, PlayerAddress_to_GC);
    input clk, rst;
    input[3:0] PasswordSwitch;
    input PasswordButton, MatchedID;
    input[4:0] PlayerAddress_from_ID;
    input LogoutCommand_from_GC;
    input isGuest_from_ID;
    output LoggedOut, LoggedIn, isGuest_to_GC, Logout_to_ID;
    reg LoggedOut, LoggedIn, isGuest_to_GC, Logout_to_ID;
    output[4:0] PlayerAddress_to_GC;
    reg[4:0] PlayerAddress_to_GC;

    reg[23:0] UserEnteredPassword;

    parameter STANDBY = 0, DIGIT1 = 1, DIGIT2 = 2, DIGIT3 = 3,
              DIGIT4 = 4, DIGIT5 = 5, DIGIT6 = 6, FETCH_ROM = 7,
              WAIT1 = 8, WAIT2 = 9, CATCH_ROM = 10, COMPARE = 11, PASSED = 12;

    reg [3:0] State;
    reg[4:0] ROM_Address;
    reg [1:0] WrongPasswordCnt;
    reg[23:0] Password_Stored;
    wire[23:0] Password_from_ROM;

    PasswordROM_24 PasswordROM(ROM_Address, clk, Password_from_ROM);

    always@(posedge clk) begin
        if(rst == 1'b0) begin
            LoggedOut <= 1'b1;
            LoggedIn <= 1'b0;
            isGuest_to_GC <= 1'b0;
            PlayerAddress_to_GC <= 0;
            State <= STANDBY;
        end
        else begin
            case(State)
            STANDBY: begin
                LoggedOut <= 1'b1;
                LoggedIn <= 1'b0;
                Logout_to_ID = 1'b0; // reset logout pulse
                WrongPasswordCnt <= 0;
                if(MatchedID == 1'b1) begin
                    if(isGuest_from_ID == 1'b1) begin
                        State <= PASSED; // Guest skips password entry
                    end
                    else begin
                        State <= DIGIT1;
                    end
                end
            end
            DIGIT1: begin
                if(PasswordButton == 1'b1) begin
                    UserEnteredPassword[23:20] <= PasswordSwitch;
                    State <= DIGIT2;
                end
            end
            DIGIT2: begin
                if(PasswordButton == 1'b1) begin
                    UserEnteredPassword[19:16] <= PasswordSwitch;
                    State <= DIGIT3;
                end
            end
            DIGIT3: begin
                if(PasswordButton == 1'b1) begin
                    UserEnteredPassword[15:12] <= PasswordSwitch;
                    State <= DIGIT4;
                end
            end
            DIGIT4: begin
                if(PasswordButton == 1'b1) begin
                    UserEnteredPassword[11:8] <= PasswordSwitch;
                    State <= DIGIT5;
                end
            end
            DIGIT5: begin
                if(PasswordButton == 1'b1) begin
                    UserEnteredPassword[7:4] <= PasswordSwitch;
                    State <= DIGIT6;
                end
            end
            DIGIT6: begin
                if(PasswordButton == 1'b1) begin
                    UserEnteredPassword[3:0] <= PasswordSwitch;
                    State <= FETCH_ROM;
                end
            end
            FETCH_ROM: begin
                ROM_Address <= PlayerAddress_from_ID;
                State <= WAIT1;
            end
            WAIT1: begin
                State <= WAIT2;
            end
            WAIT2: begin
                State <= CATCH_ROM;
            end
            CATCH_ROM: begin
                Password_Stored <= Password_from_ROM;
                State <= COMPARE;
            end
            COMPARE: begin
                if (WrongPasswordCnt == 3) begin // three wrong attempts trigger change in ID
                    Logout_to_ID <= 1'b1;
                    State <= STANDBY;
                end
                else begin
                    if (Password_Stored == UserEnteredPassword) begin
                        State <= PASSED;
                    end
                    else begin
                        WrongPasswordCnt <= WrongPasswordCnt + 1;
                        State <= DIGIT1;
                    end
                end
            end
            PASSED: begin
                LoggedOut <= 1'b0;
                LoggedIn <= 1'b1;
                isGuest_to_GC <= isGuest_from_ID;
                PlayerAddress_to_GC <= PlayerAddress_from_ID;
                if(LogoutCommand_from_GC == 1'b1) begin
                    LoggedIn <= 1'b0;
                    LoggedOut <= 1'b1;
                    Logout_to_ID <= 1'b1; // set logout pulse
                    State <= STANDBY;
                end
            end
            default: begin
                State <= STANDBY;
            end
            endcase
        end
    end
endmodule
```

```
module ScoreTracker(clk, rst, score_req, playerID, isGuest, Score, personal_winner, global_winner);
    input clk, rst;
    input isGuest;
    input [4:0] playerID;
    input [3:0] score_req;
    input [13:0] Score;
    output personal_winner, global_winner;
    reg personal_winner, global_winner;

    parameter RAMINIT = 0, WAITSCORE = 1, RAMFETCH = 2, RAMFETCH1 = 3, RAMFETCH2 = 4, RAMCATCH = 5, COMPARE = 6, RAMWRITE = 7, GLOBALCHECK = 8, WAITGAME = 9;

    reg [13:0] Player_score, Global_score, RAM_Data;
    reg [3:0] State;
    reg [4:0] Player_ID;
    reg [1:0] WinningPlayer;
    reg [1:0] counter;
    reg resetflag;

    reg read_write;
    reg [4:0] RAM_addr;
    reg [13:0] RAM_out;

    wire [13:0] RAM_in;

    RAM_SCORE GAME_SCORES(RAM_addr, clk, RAM_out, read_write, RAM_in);

    always @(posedge clk) begin
        if (rst == 1'b0) begin
            Global_score <= 14'b00000000000000;
            WinningPlayer <= 2'b00;
            personal_winner <= 1'b0;
            global_winner <= 1'b0;
            counter <= 2'b00;
            read_write <= 1'b0;
            RAM_addr <= 5'b00000;
            RAM_out <= 14'b00000000000000;
            State <= RAMINIT;
          end
        else
          begin
            case (State)
              RAMINIT: begin
                  read_write <= 1'b1;
                  RAM_addr <= {3'b000, counter};
                  RAM_out <= 14'b00000000000000;
                  counter <= counter + 1;
                  if (counter == 2'b11) begin
                      read_write <= 1'b0;
                      State <= WAITSCORE;
                    end
                end
              WAITSCORE: begin
                  if (score_req == 5) begin
                      if (isGuest == 1'b1) begin
                          State <= WAITGAME;
                        end
                      else
                        begin
                          Player_score <= Score;
                          Player_ID <= playerID;
                          State <= RAMFETCH;
                        end
                    end
                  else
                    begin
                      State <= WAITSCORE;
                    end
                end
              RAMFETCH: begin
                  RAM_addr <= Player_ID;
                  read_write <= 1'b0;
                  State <= RAMFETCH1;
                end
              RAMFETCH1: begin
                  State <= RAMFETCH2;
                end
              RAMFETCH2: begin
                  State <= RAMCATCH;
                end
              RAMCATCH: begin
                  RAM_Data <= RAM_in;
                  State <= COMPARE;
                end
              COMPARE: begin
                  if (Player_score > RAM_Data) begin
                      RAM_out <= Player_score;
                      personal_winner <= 1'b1;
                      State <= RAMWRITE;
                    end
                  else
                    begin
                      personal_winner <= 1'b0;
                      State <= GLOBALCHECK;
                    end
                end
              RAMWRITE: begin
                  read_write <= 1'b1;
                  State <= GLOBALCHECK;
                end
              GLOBALCHECK: begin
                  if (Player_score > Global_score) begin
                      Global_score <= Player_score;
                      WinningPlayer <= Player_ID;
                      global_winner <= 1'b1;
                    end
                  else
                    begin
                      global_winner <= 1'b0;
                    end
                  State <= WAITGAME;
                end
              WAITGAME: begin
                  read_write <= 1'b0;
                  if (score_req != 5) begin
                      personal_winner <= 1'b0;
                      global_winner <= 1'b0;
                      State <= WAITSCORE;
                    end
                  else
                    begin
                      State <= WAITGAME;
                    end
                end
              default: begin
                  RAM_addr <= 5'b00000;
                  personal_winner <= 1'b0;
                  global_winner <= 1'b0;
                  Player_score <= 14'b00000000000000;
                  read_write <= 1'b0;
                  State <= WAITSCORE;
                end
            endcase
          end
      end
endmodule
```

```
module SegmentRunner(SwPass, BtPass, BtGame, BtLogOut, Disp5, Disp4, Disp3, Disp2, Disp1, Disp0, LEDPScore, LEDHScore, LEDLoggedIn, LEDLoggedOut, LEDMatchedID, LEDGameTick, Clk, Rst);
    input [3:0] SwPass;
    input BtPass, BtGame, BtLogOut, Clk, Rst;
    output [6:0] Disp5, Disp4, Disp3, Disp2, Disp1, Disp0;
    output LEDPScore, LEDHScore, LEDLoggedIn, LEDLoggedOut, LEDMatchedID, LEDGameTick;

    // Shaped Inputs
    wire BTSPass, BTSGame, BTSLogOut;
    ButtonShaper BTSPPass(BtPass, BTSPass, Clk, Rst);
    ButtonShaper BTSPGame(BtGame, BTSGame, Clk, Rst);
    ButtonShaper BTSPLogOut(BtLogOut, BTSLogOut, Clk, Rst);

    // Authenticator
    wire AuthLoggedIn, AuthLoggedOut, AuthIsGuest, AuthMatchedID;
    wire [4:0] AuthPlayerID;
    Authenticator ModAuth(Clk, Rst, SwPass, BTSPass, BTSLogOut, AuthLoggedOut, AuthLoggedIn, AuthIsGuest, AuthPlayerID, AuthMatchedID);
    assign LEDLoggedIn = AuthLoggedIn;
    assign LEDLoggedOut = AuthLoggedOut;
    assign LEDMatchedID = AuthMatchedID;

    // Game Controller
    wire [13:0] GCGameScore;
    wire [5:0] GCFloorBits, GCCeilingBits;
    wire [3:0] GCGameState;
    wire GCPlayerPos, GCGameTick;
    GameController GCMain(AuthLoggedIn, BTSGame, GCFloorBits, GCCeilingBits, GCPlayerPos, GCGameTick, GCGameState, GCGameScore, Clk, Rst);
    assign LEDGameTick = GCGameTick;

    // Score Tracking
    ScoreTracker ScoreBoard(Clk, Rst, GCGameState, AuthPlayerID, AuthIsGuest, GCGameScore, LEDPScore, LEDHScore);

    // Display
    reg DispShowScore;
    always@(GCGameState) begin
        if (GCGameState == 4'd5) begin
            DispShowScore = 1'b1;
        end else begin
            DispShowScore = 1'b0;
        end
    end
    Decoder DisplayOuts(Clk, AuthLoggedIn, GCCeilingBits, GCFloorBits, GCPlayerPos, GCGameScore, DispShowScore, Disp5, Disp4, Disp3, Disp2, Disp1, Disp0);


endmodule
```

```
module ShiftReg_6b(BitIn, Shift, RegContent, Clk, Rst, RstValue);
    input BitIn, Shift, Clk, Rst;
    input [5:0] RstValue;
    output [5:0] RegContent;
    reg [5:0] RegContent;

    always@(posedge Clk) begin
        if (Rst == 1'b0) begin
            RegContent <= RstValue;
        end else if (Shift == 1'b1) begin
            RegContent[5] <= RegContent[4];
            RegContent[4] <= RegContent[3];
            RegContent[3] <= RegContent[2];
            RegContent[2] <= RegContent[1];
            RegContent[1] <= RegContent[0];
            RegContent[0] <= BitIn;
        end
    end
endmodule
```

```
module SurfaceGen(TransitionTick, TransitionDiff, Floor, Ceiling, MoveTick, Clk, Rst);
    input TransitionTick, MoveTick, Clk, Rst;
    input [3:0] TransitionDiff;
    output [5:0] Floor, Ceiling;

    // Shift Registers for FLoor and Ceiling
    reg FloorIn, CeilIn, SRRst;
    reg [5:0] CeilRstVal, FloorRstVal;
    ShiftReg_6b FloorSR(FloorIn, MoveTick, Floor, Clk, SRRst, FloorRstVal);
    ShiftReg_6b CeilSR(CeilIn, MoveTick, Ceiling, Clk, SRRst, CeilRstVal);

    // State Machine
    reg [3:0] State;
    reg [3:0] Count;
    reg LastPos; // 0 = floor, 1 = ceil
    parameter NONE=0, TRANSITIONSTART=1, TRANSITION=2, TRANSITIONFIN=3;

    always@(posedge Clk) begin
        if (Rst == 1'b0) begin
            FloorRstVal <= 6'b111111;
            CeilRstVal <= 6'b001100;
            FloorIn <= 1'b1;
            CeilIn <= 1'b0;
            LastPos <= 1'b0;
            SRRst <= 1'b0;
            Count <= 0;
            State <= NONE;
        end

        if (TransitionTick == 1'b1) begin
            if (State == NONE) State <= TRANSITIONSTART;
        end

        if (MoveTick == 1'b1) begin
            case (State)
                NONE: begin
                    SRRst <= 1'b1;
                end
                TRANSITIONSTART: begin
                    if (FloorIn == 1'b1) begin
                        LastPos <= 0;
                    end else begin
                        LastPos <= 1;
                    end
                    FloorIn <= 1;
                    CeilIn <= 1;
                    Count <= 0;
                    State <= TRANSITION;
                end
                TRANSITION: begin
                    if (Count < TransitionDiff - 1) begin
                        Count <= Count + 1;
                    end else begin
                        if (LastPos <= 1'b0) begin
                            CeilIn <= 1;
                            FloorIn <= 0;
                        end else begin
                            CeilIn <= 0;
                            FloorIn <= 1;
                        end
                        State <= TRANSITIONFIN;
                    end
                end
                TRANSITIONFIN: begin
                    // one move cycle delay
                    State <= NONE;
                end
                default: begin
                    SRRst <= 1'b1;
                    State <= NONE;
                end
            endcase
        end
    end

endmodule
```

```
1  module Timer_1ms(Enable, Tick, Clk, Rst);
2      input Clk, Rst, Enable;
3      output Tick;
4      reg Tick;
5
6      wire [15:0] LFSR;
7      reg LFSR_Rst;
8      LFSR_16b cntr(LFSR, Enable, Clk, LFSR_Rst);
9
10     always @(posedge Clk) begin
11         if (Rst == 1'b0) begin
12             LFSR_Rst <= 1'b0;
13             Tick <= 1'b0;
14         end else begin
15             if (Enable == 1'b1) begin
16                 if (LFSR == 16'hDB6C) begin
17                     Tick <= 1'b1;
18                     LFSR_Rst <= 1'b0;
19                 end else begin
20                     Tick <= 1'b0;
21                     LFSR_Rst <= 1'b1;
22                 end
23             end else begin
24                 Tick <= 1'b0;
25                 LFSR_Rst <= 1'b1;
26             end
27         end
28     end
29 endmodule
```

```
module Timer_rc(Enable, Clear, CfgValue, Tick, Clk, Rst);
    input Enable, Clear, Clk, Rst;
    input [9:0] CfgValue;
    output Tick;
    reg Tick;

    reg [9:0] Count;
    wire IntTick;
    Timer_1ms Internal(Enable, IntTick, Clk, Rst);

    always@(posedge Clk) begin
        if (Rst == 1'b0) begin
            Tick <= 0;
            Count <= 0;
        end else if (Clear == 1'b1) begin
            Count <= 0;
        end else if (IntTick == 1'b1) begin
            if (Count < CfgValue) begin
                Count <= Count + 1;
                Tick <= 1'b0;
            end else begin
                Count <= 1'b0;
                Tick <= 1'b1;
            end
        end else if (Tick <= 1'b1) begin
            Tick <= 1'b0;
        end
    end
endmodule
```