**Final Project:** “Gone Fishin’” Game on FPGA

**Authors:** Jon Genty (2849), George Hodgkins (4755), Nathan Keyes (5198), Fazal Mahmood (1127), Nick Trevino (2389)

**Semester:** Fall 2020

**Course:** ECE 5440

**Instructor:** Dr. Yuhua Chen

**Introduction:**

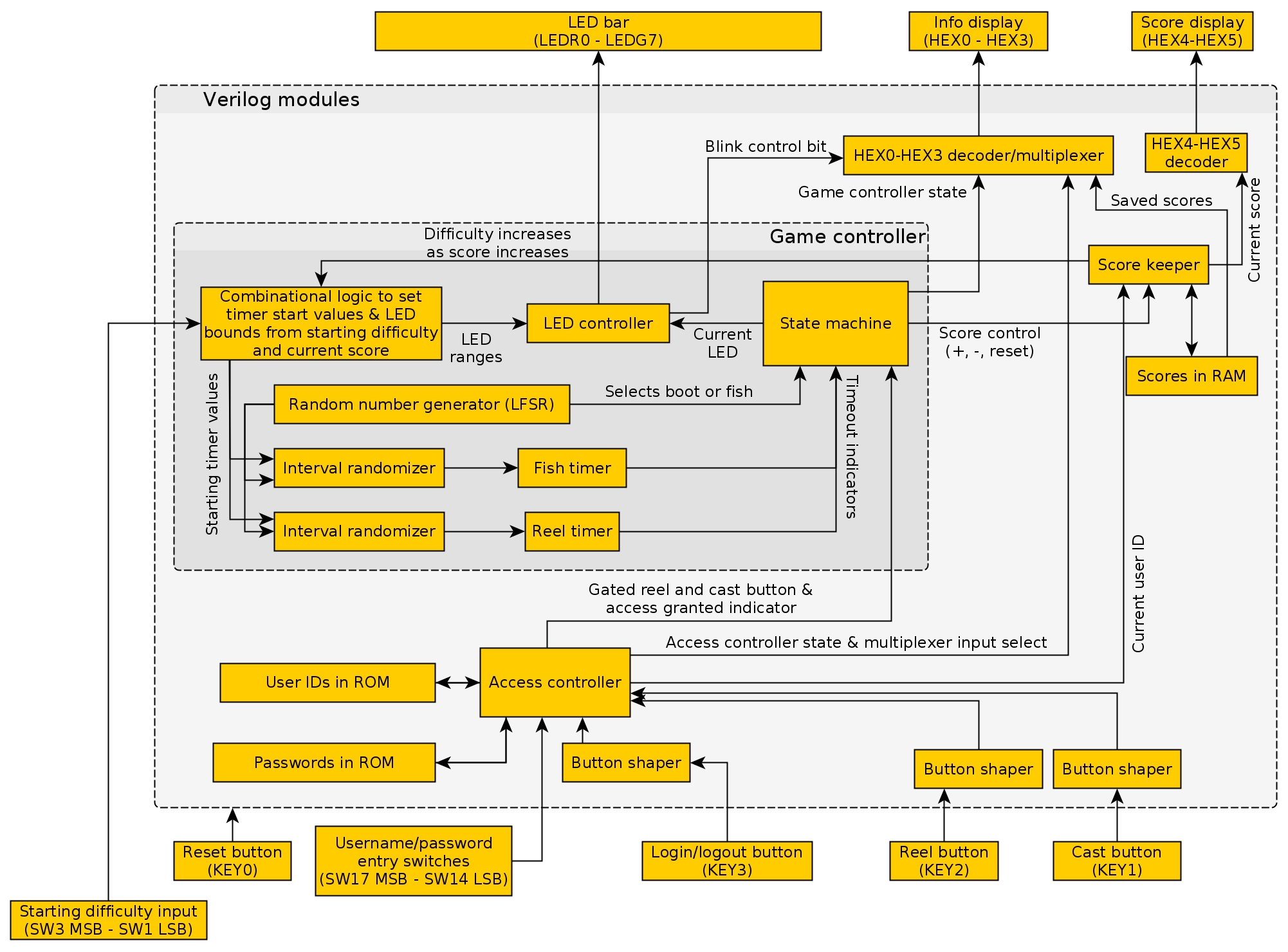
This FPGA based single-player game is designed to simulate a fishing experience, in which players must catch as many fish as possible in a row before they fail. The player has two buttons during gameplay, REEL and CAST. A player must first cast their fishing rod to start the game by pressing the CAST button. A timer will start ticking down, in which the player must wait until something is on their hook. Only two things can be caught, a fish or a boot. Successfully catching a fish will increase your score, while attempting to even catch a boot will decrease your score. When a fish is lost, the current game is over until the user casts their rod back into the water. The goal of the game is to get the highest score possible.

When a fish is on the hook, a solid LED bar will appear, with two blinking LED’s surrounding it and a singular blinking LED in the middle. This system simulates the fishing process. The singular blinking LED represents your reel control. By default, this LED will move to the left, representing the fish getting away. If your reel control moves past the blinking-left boundary, the fish is lost and the game is over. By pressing the REEL button, this will move your reel control toward the right, representing reeling the fish closer. However, moving past the blinking-right boundary will snap the line, also losing the fish for aggressive reeling.

Hence, there is a “sweet-zone” to be in, which is the solid LED bar of green and red LED’s. Having your reel control in this zone will decrement a timer; this timer hitting zero catches the fish and increases your score. The difficulty of this task is determined by an initial difficulty setting, and dynamically increases as your score gets higher. Timing is also augmented with an LFSR-based RNG.

**System Architecture:**

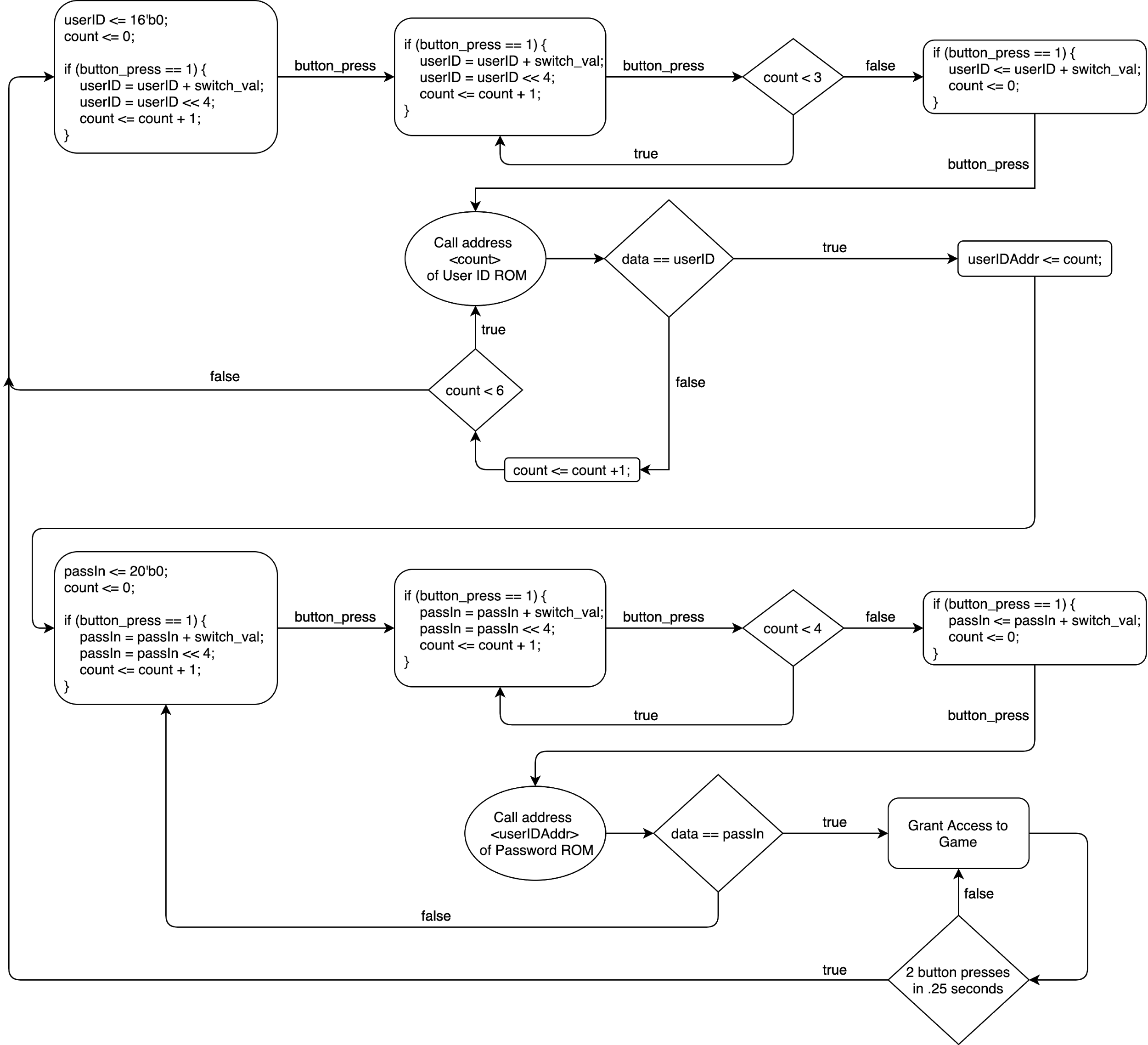
This system consists of 18 independent modules and 3 memory initialization files: access\_controller, hex0\_to\_hex3\_decoder, hex4\_to\_hex5\_decoder, User\_ROM (Quartus auto-generated file), Password\_ROM (Quartus auto-generated file), gameController, difficultySelector, RangeLFSR16, LEDBlinker, multiReg, LFSR16, countdownTimer, biDirBCD, scorekeeper, ramController, ram, SevenSeg, and button\_shaper. Explanations for auto-generated and memory initialization files along with SevenSeg and button\_shaper have been omitted. Figure 1 illustrates how these modules are connected and the System Architecture.



*Figure 1. Gone Fishin’ System Architecture*

**access\_controller:**

*Description:* The access controller module will block any gameplay attempts before a user has been authenticated. To pass authentication, a user must enter a valid 4-digit User ID from the User\_ROM, and a matching 5-digit password from the Password\_ROM. The user must use 4 switches to input a 4-bit value for each of these digits, along with a button to load the button for checking. Once 4-digits have been loaded for the User ID, this module will check each ROM address (6 total addresses for 6 unique users) for a match. If a match is not found, the user can try a different User ID input until a match is found. If a match is found, it will store the address, and begin waiting for the user password input. Once 5-digits have been loaded for the User Password, this module will check the ROM address which the User ID match was found, but in the Password ROM. If a match is not found, the user can try a different User Password input until a match is found (with unlimited attempts). If a match is found, the user will be authenticated into the game. Once access is granted, the user may quickly double-click the same button used for load, at any point. The logout features sensitivity is .25 seconds in between the two button presses. These operations have been depicted in Figure 2, below.



*Figure 2. access\_controller flow diagram*

*Inputs/ Parameters:*

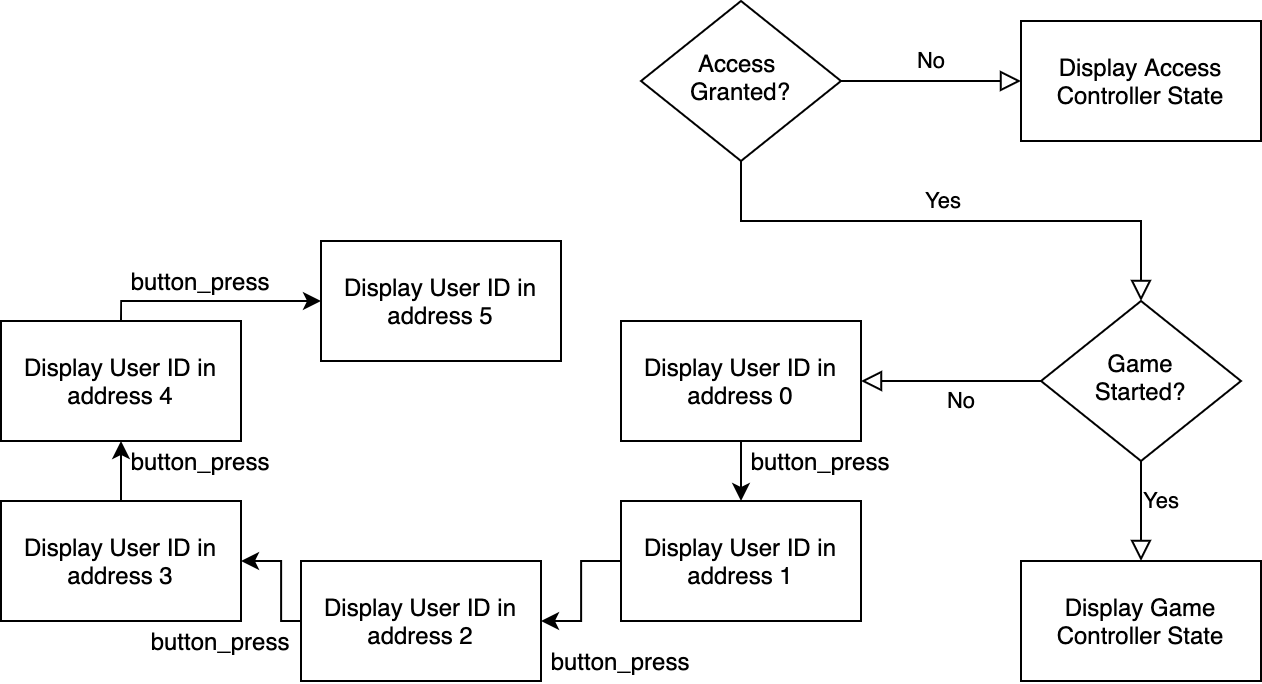
* clk: System clock signal.
* rst: System reset signal.
* access\_switch: User 4-bit input for User ID or Password digit.
* access\_button\_in: User digit load/logout button.
* reel\_button\_in: System game button.
* game\_start\_button\_in: System game button.

*Outputs:*

* validOut: 1-bit positive-logic indicator of access granted.
* access\_button\_out: System game button (blocked or propagated).
* reel\_button\_out: System game button (blocked or propagated).
* game\_start\_button\_out: System game button (blocked or propagated).
* output\_to\_decoder: 4-bit value sent to hex0\_to\_hex3\_decoder for display.
* state\_for\_decoder: 4-bit value sent to hex0\_to\_hex3\_decoder for display.
* user\_ID: Address User ID was matched in ROM for ramController to save score.

**hex0\_to\_hex3\_decoder:**

*Description:* This module is used to output four 7-segment displays on the FPGA in an effort to enhance user experience. Since this module outputs directly to the 7-segments of the FPGA Board, it uses 4 instances of the SevenSeg module to convert 4-bit numbers to 7-bit versions used for display. When the user has not been authenticated to the game, this module will display “U” or “P” to HEX 3 for User ID or Password, 1-4 on HEX 2 depending on the digit access\_controller is waiting on to be loaded, nothing to HEX 1, and the current, real-time switch value. When the user has been authenticated, but the game has not been started these 7-segment displays are used to display the User ID’s in ROM, such that the user's high score can be viewed. When the access button is pressed, this module will switch to the next User ID in ROM in a round robin fashion. When the user has been authenticated and the game has been started, these 7-segment displays are used to display the game states such as: “FISH”, “LOSE”, “dOnE”, and <NOTHING>. Figure 3, below, illustrates these operations in a flow diagram.



*Figure 3. hex0\_to\_hex3\_decoder flow diagram*

*Inputs/ Parameters:*

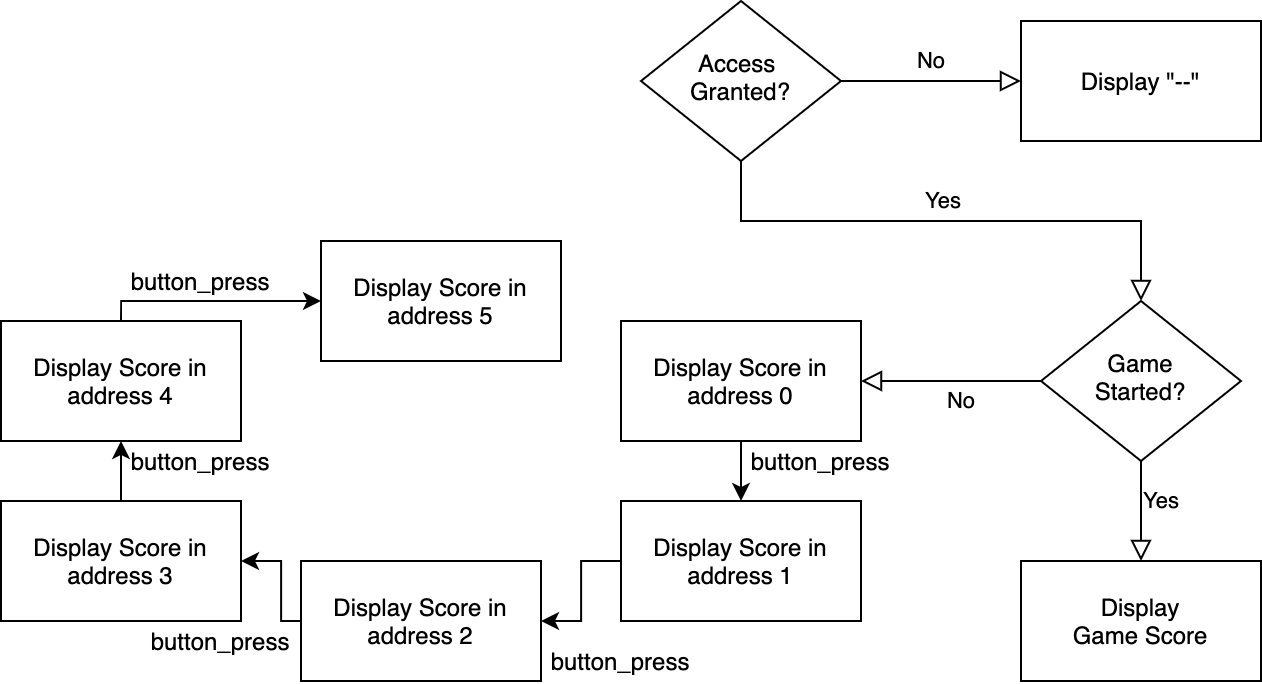
* clk: System clock signal.
* rst: System reset signal.
* access\_granted: 1-bit positive-logic indicator of access granted.
* ac\_button: User button used to switch between User IDs.
* scoreRst: 1-bit positive-logic indicator of gameController releasing control of display.
* switch\_val\_in: 4-bit switch value for user from access\_controller.
* state\_ac: 4-bit state of access\_controller used to properly output to displays.
* state\_gc: 4-bit state of gameController used to properly output to displays.

*Outputs:*

* hex0\_out: 7-bit output to properly directly to HEX 0 of FPGA Board.
* hex1\_out: 7-bit output to properly directly to HEX 1 of FPGA Board.
* hex2\_out: 7-bit output to properly directly to HEX 2 of FPGA Board.
* hex3\_out: 7-bit output to properly directly to HEX 3 of FPGA Board.

**hex4\_to\_hex5\_decoder:**

*Description:* This module is used to output two 7-segment displays on the FPGA in an effort to enhance user experience. Since this module outputs directly to the 7-segments of the FPGA Board, it uses 2 instances of the SevenSeg module to convert 4-bit numbers to 7-bit versions used for display. When the user has not been authenticated to the game, this module will display “--”. When the user has been authenticated, but the game has not been started these 7-segment displays are used to display high-scores for each User ID in RAM. When the access button is pressed, this module will switch to the next high-score saved in RAM, switching in a round robin fashion. When the user has been authenticated and the game has been started, these 7-segment displays are used to display the current game score. Figure 4, below, illustrates these operations in a flow diagram.



*Figure 4. hex4\_to\_hex5\_decoder flow diagram*

*Inputs/ Parameters:*

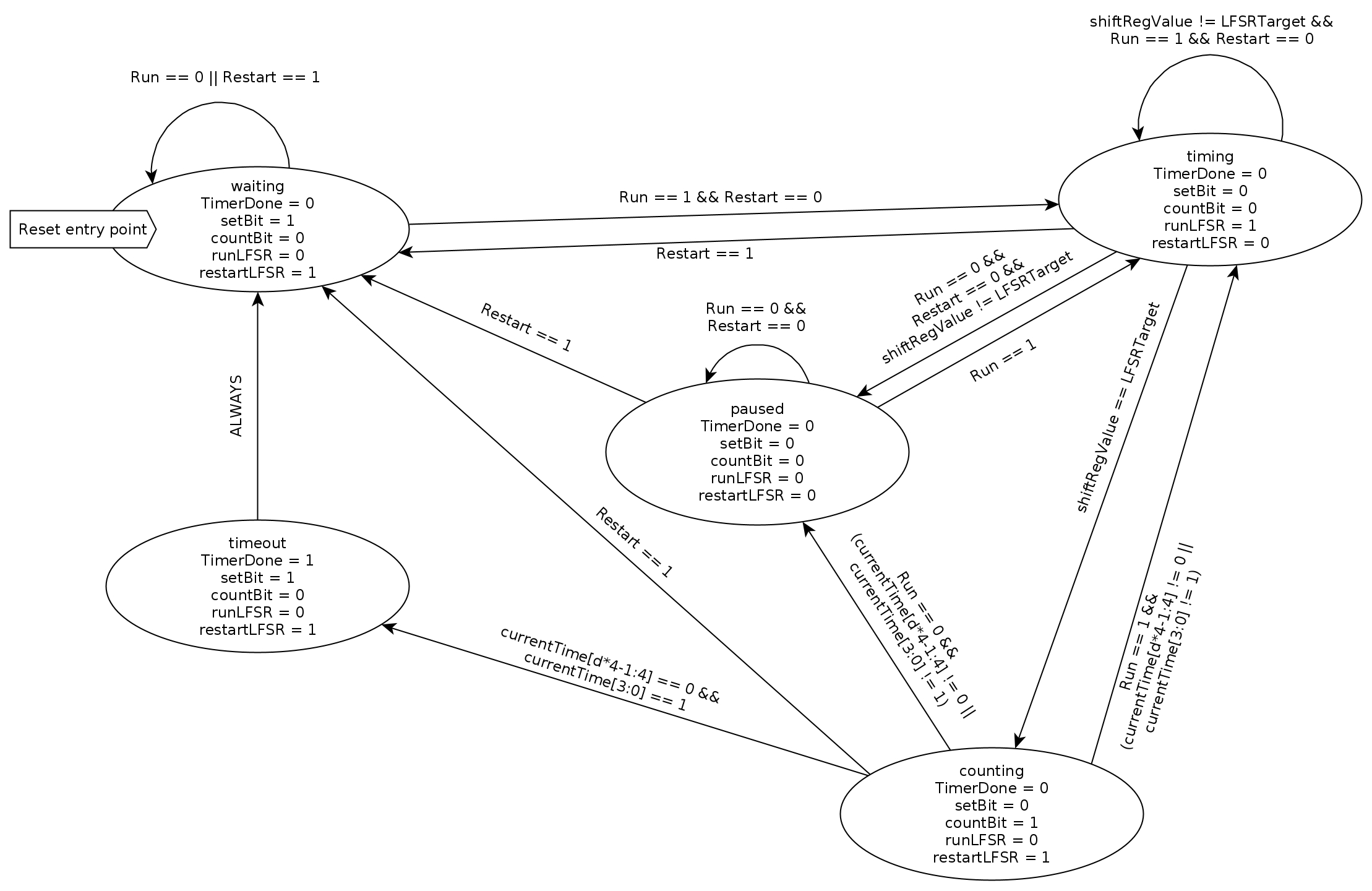
* clk: System clock signal.
* rst: System reset signal.
* access\_granted: 1-bit positive-logic indicator of access granted.
* ac\_button: User button used to switch between User high-scores.
* scoreRst: 1-bit positive-logic indicator of gameController releasing control of display.
* userID: The User ID of user currently logged in \*Not used\*.
* currentGameScore: 8-bit score of current game (includes 10’s and 1’s digit).
* scoreUserAddr0: 8-bit high-score in RAM of User ID 1 (includes 10’s and 1’s digit).
* scoreUserAddr1: 8-bit high-score in RAM of User ID 2 (includes 10’s and 1’s digit).
* scoreUserAddr2: 8-bit high-score in RAM of User ID 3 (includes 10’s and 1’s digit).
* scoreUserAddr3: 8-bit high-score in RAM of User ID 4 (includes 10’s and 1’s digit).
* scoreUserAddr4: 8-bit high-score in RAM of User ID 5 (includes 10’s and 1’s digit).
* scoreUserAddr5: 8-bit high-score in RAM of User ID 6 (includes 10’s and 1’s digit).

*Outputs:*

* hex4\_out: 7-bit output to properly directly to HEX 4 of FPGA Board.
* hex5\_out: 7-bit output to properly directly to HEX 5 of FPGA Board.

**countdownTimer:**

*Description:* This module is an LFSR-based, decimal-valued timer that has a resolution of 1 ms and a compile-time configurable number of digits. It has run and reset inputs, as well as a configurable starting time, and outputs the current time as well as a timeout indicator bit. Internally, this functionality is implemented using a chain of biDirBCD modules and a 5-state state machine (shown in the Figure 5).



*Figure 5. countdownTimer Finite State Machine diagram*

*Inputs/Parameters:*

* Digits (parameter): Number of decimal digits in the time, starting from the 1 ms digit up (i.e. 5 digits is xx.xxx seconds).
* LFSRTarget (parameter): The LFSR counter value corresponding to the number of clock cycles in 1 ms for your system’s clock speed, minus two.
* StartValue: The value that the timer will count down from, after it is reloaded and restarted using the Restart and Run inputs. Each decimal digit is represented as a 4-bit hex number (so on a 5-digit timer this input would be 20 bits wide).
* Restart: This input stops the timer and reloads the starting value. It takes priority over the Run input, so the timer will not run while it is high.
* Run: This input runs the timer. If neither it nor the Restart input is asserted, the timer will pause but not restart.
* CLK: System clock signal.
* RST: Negative-logic reset.

*Outputs:*

* CurrentValue: The time remaining before the timer times out. Same width and representation as the StartValue input.
* TimerDone: This input is driven high for one clock cycle when the timer times out.

**biDirBCD:**

*Description:* This module is a decimal-valued counter which can be configured to count either up or down (rolling over when reaching 9/0), and can be chained together with others of the same module to represent multi-digit decimal numbers. It can be loaded to start at any decimal digit.

*Inputs/Parameters:*

* Direction (parameter): This parameter controls the counter direction. If it is equal to 1, the counter will count up; otherwise, it will count down.
* Count: If this input is high on a rising clock edge, the counter will count in the configured direction.
* SetValue: 4-bit hex number representing a decimal digit to load into the counter when the Set input is high across a rising clock edge. If this input is 10 or greater, it will not be loaded even if the Set input is high.
* Set: If this input is high across a clock edge, SetValue will be loaded into the counter, if it is a valid decimal digit.
* CLK: System clock signal.
* RST: Negative-logic reset.

*Outputs:*

* OutValue: Current value of the counter, as a 4-bit hex number.
* CarryOut: If the value of the counter is 0 if counting down, or 9 if counting up, and the Count bit is asserted (and the Set bit is not asserted), this output will be driven high. This is used to create a chain of these counters, by connecting this output to the Count input of the next greater digit (i.e. if this were the ones digit it would be connected to the tens digit). This output is driven asynchronously so that carries can propagate through multiple counters if necessary (i.e. when transitioning from 099 to 100).

**LEDBlinker:**

*Description:* This module is used to control the LED bar on the board, that is, all of the LEDs except LEDG8. Its inputs select which LEDs should blink (using an internal timer with a compile-time configurable period) and which should be on constantly; it also has an enable/disable input to completely turn off the output if necessary, and it also outputs the blink bit for the use of other modules which should blink.

*Inputs/Parameters:*

* blinkTime (parameter): Sets the period of the blinking LEDs (specifically, the amount of time the LEDs spend on or off, so it is equal to the period divided by two). Represented as three 4-bit hex digits representing tenths, hundredths, and thousandths of seconds for input into an internal countdownTimer module (see above).
* LFSRTarget (parameter): Passed through to the countdownTimer module (see above).
* lightThese: These LEDs will be lit continuously, if blinkEnab is high. Represented as 26 bits, one per LED, where the MSB represents the leftmost LED (LEDR0) and the LSB represents the rightmost LED (LEDG7).
* blinkThese: These LEDs will blink at the rate specified by blinkTime, if blinkEnab is high; represented in the same format as lightThese. If an LED is selected in both lightThese and blinkThese, it will blink.
* blinkEnab: Enables or disables LED output; when this input is low, no LEDs will be lit, and the blinkBit output will be high continuously.
* CLK: System clock signal.
* RST: Negative-logic reset.

*Outputs:*

* lightBar: 26-bit output to the LED bar.
* blinkBit: Output which can be used by other modules to synchronize their blinking with the LED bar; high when the blinking LEDs are on and low when they are off. Note that when the blinkEnab input is low, this output will be high continuously.

**LFSR16:**

*Description:* Implements a maximum-period 16-bit Xorshift linear feedback shift register. For each sequence number from 0 to 2^16 – 1, there exists a unique output value for this shift register, so it can be used as a lower-overhead replacement for a normal counter, as long as the target counter value is known beforehand. This implementation has a hardcoded seed of FFFFh and a hardcoded triplet of [7, 9, 8].

*Inputs:*

* Restart: If this input is high, the LFSR returns to its starting value and holds there until this input goes low. Takes priority over the Run bit.
* Run: If this input is high and Restart is low, the LFSR will transition to the next value in the sequence on each rising clock edge.
* CLK: System clock signal.
* RST: Negative-logic reset.

*Outputs:*

* Value: Current value of the LFSR.

**RangeLFSR16:**

*Description:* Uses the LFSR16 module to generate random numbers within a range specified by the user.

*Inputs:*

* Restart: If this input is high, the LFSR returns to its starting value and holds there until this input goes low. Takes priority over the Run bit.
* Run: If this input is high and Restart is low, the LFSR will transition to the next value in the sequence on each rising clock edge.
* offset: Represents the smallest number the Ranged LFSR can generate.
* Limit: Represents the largest number the Ranged LFSR can generate.
* CLK: System clock signal.
* RST: Negative-logic reset.

*Outputs:*

* out: Current value of the ranged LFSR.

**difficultySelector:**

*Description:* This module is mostly combinational logic that sets the load values for the two timers in the gameController module (see below) based on the current score and difficulty setting when the game starts, and sets the LED bounds for gameplay based on the initial difficulty setting alone. The only sequential part of the module is a load register which latches the difficulty setting so that it cannot be changed during gameplay.

*Inputs/Parameters:*

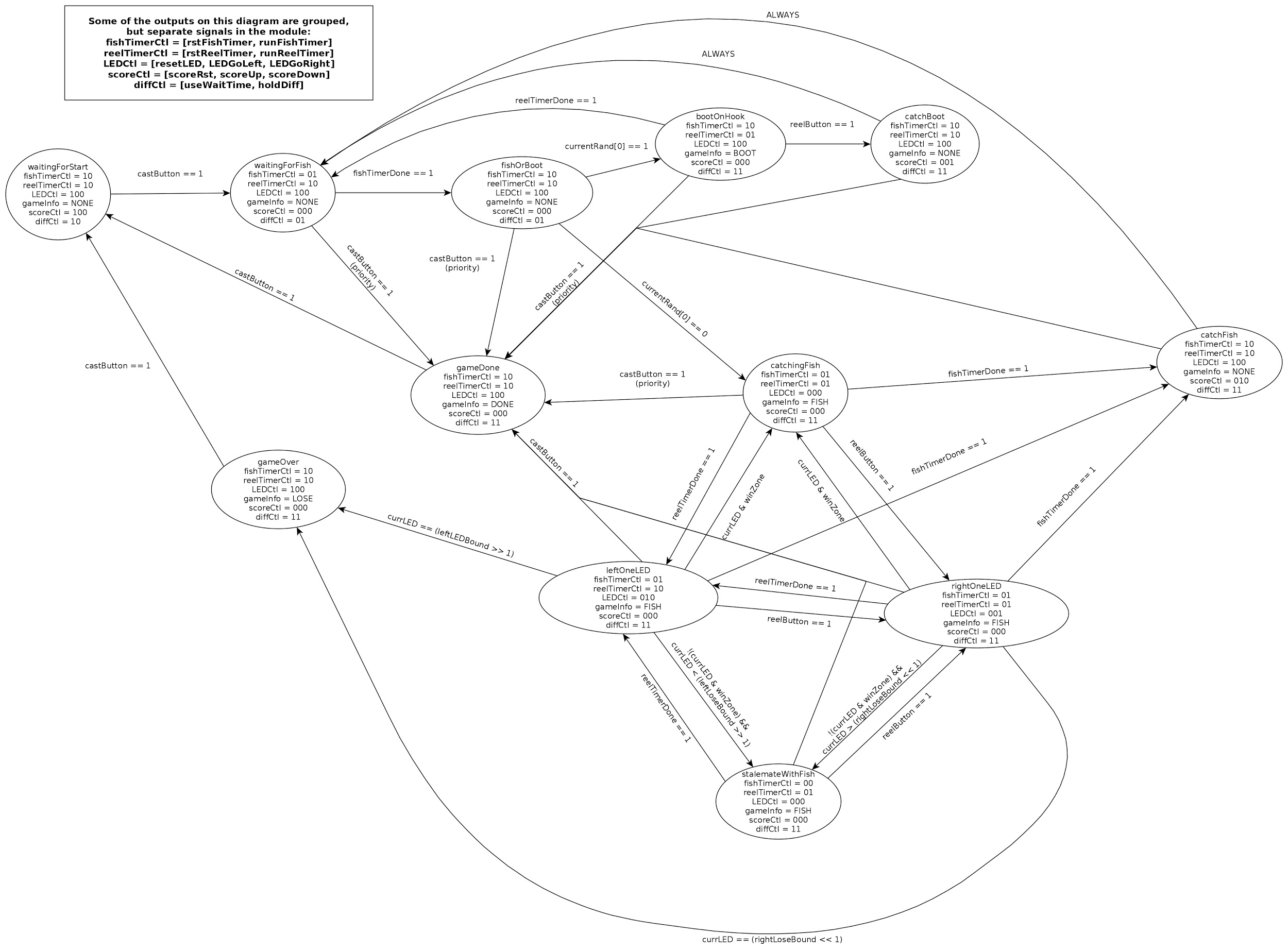
* WAIT\_TIME (parameter): The fixed waiting time selected by the useWaitTime input (see below), represented as 5 4-bit hex digits representing a time in seconds a tens, ones, tenths, hundredths, and thousandths places.
* currentDiff: Current difficulty (set by toggle switches): any valid 3-bit value 0-7.
* currentScore: Current score (from score keeper): two decimal digits represented by two 4-bit hex numbers.
* holdDiff: If this input is high across a rising clock edge, the value of currentDiff will be latched into the internal startingDiff register, which will not change until holdDiff is low again.
* useWaitTime: If this input is high, the fishTime output will be equal to the WAIT\_TIME parameter rather than based on difficulty. This is here because the same timer is used for both waiting periods and fishing periods, but the waiting period should not increase with difficulty like the fishing period.
* CLK: System clock signal.
* RST: Negative-logic reset.

*Outputs:*

* reelTime: 12-bit load value for the reelTimer module, representing a time .xxx seconds. Decreases as difficulty and score increase.
* fishTime: 20-bit load value for the fishTimer module, representing a time xx.xxx seconds. While the useWaitTime input is high, this output is equal to the WAIT\_TIME parameter; when useWaitTime is low, this output increases as difficulty and score increase.
* leftLoseBound: Maximum (leftmost) LED that currentLED can be equal to; decreases (moves right) as difficulty increases. 26-bit output with exactly one nonzero bit.
* rightLoseBound: Minimum (rightmost) LED that currentLED can be equal to; increases (moves left) as difficulty increases. 26-bit output with exactly one nonzero bit.
* winZone: Area of LEDs that currentLED must be in for the timer to count down, represented as a 26-bit output with a contiguous subset of nonzero bits. This output will always be greater than rightLoseBound and less than leftLoseBound.

**gameController:**

*Description:* This module implements the state machine and necessary submodules to run the game. Figure 6 below shows the finite state diagram for the state machine.



*Figure 6. gameController Finite State Machine diagram*

*Inputs:*

* gameControl: This input activates and deactivates the module; if it is not asserted, the module is held in a reset state until it goes high again.
* startingDiff: 3-bit user difficulty selection 0-7, from toggle switches. Once game is started, changes to this input will not be reflected until the current game is ended and a new one begins.
* reelButton: This input, from a shaped button signal, moves the current LED to the right during gameplay.
* castButton: This input, from a shaped button signal, starts and ends the game.
* currentScore: Current score, a decimal number represented as two 4-bit hex digits, from the score controller module.
* CLK: System clock signal.
* RST: Negative-logic reset.

*Outputs:*

* scoreUp: This output is pulsed whenever the score should be incremented.
* scoreDown: This output is pulsed whenever the score should be decremented.
* scoreRst: This output is held high whenever the game is in the reset state (waitingForStart), to indicate that the score should be reset.
* lightBar: 26-bit output to control the LED bar; should be tied directly to LEDR0-LEDG7 (see LEDBlinker description for details).
* gameInfo: 4-bit code telling the game info decoder to display various messages.
* fishTimeDisp: This outputs the tens and ones digits of the current fishTimer value for display to the user as two 4-bit hex digits 0-9.
* blinkBit: This bit is used to synchronize blinking displays with the blinking LEDs controlled by this module (see LEDBlinker description for details).

**scorekeeper:**

*Description:* This module keeps score for the game. It does this by using a 1-byte number. The first 4 bits represent the tens digit, and the second 4 bits represent the ones digit. The module can receive 1-bit signals which indicate if the score should be increased or decreased.

*Inputs/ Parameters:*

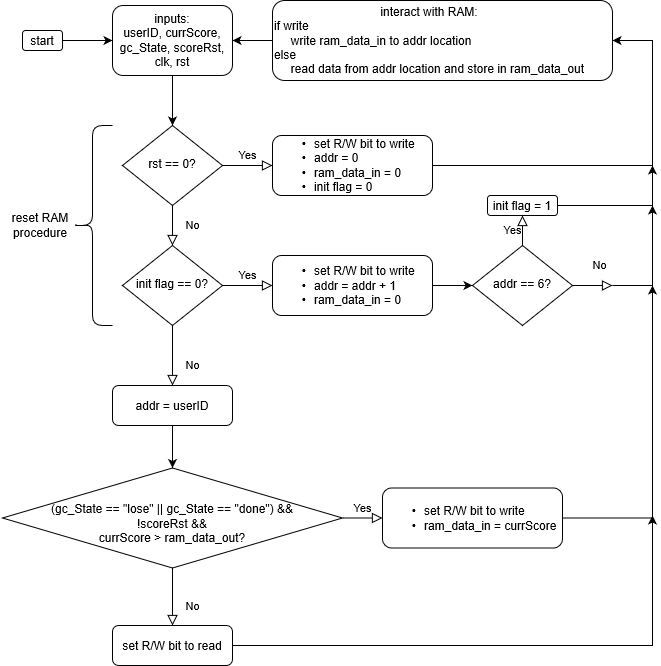
* clk: system clock signal
* rst: system reset signal
* resetScore: 1-bit signal that pulses if user wishes to reset their score, which sets output currentScore to 00000000
* incrementScore: 1-bit signal that indicates the user’s score should increment by 1
* decrementScore: 1-bit signal that indicates the user’s score should decrement by 1

*Outputs:*

* currentScore: 8-bit signal whose first 4 bits represent the 10s place, and whose second 4 bits represent the 1s place of the current user’s score

**ramController:**

*Description:* This module controls the RAM. When reset is pressed, it initializes the RAM to all 0s. Then, if the game is over and the score isn't being reset, it writes the current score to the current user's address in RAM. It also outputs the scores of all users. Figure 7 shows the flowchart for the logic in ramController.



*Figure 7. Flowchart for ramController.*

*Inputs/ Parameters:*

* currScore: the current score for the current user, as described in the scorekeeper module
* gc\_State: the current state of the game, encoded as a 4-bit number
* scoreRst: if high, then the score is reset to 0
* userID: the ID of the current user, encoded as a 3-bit number
* clk: system clock signal
* rst: system reset signal

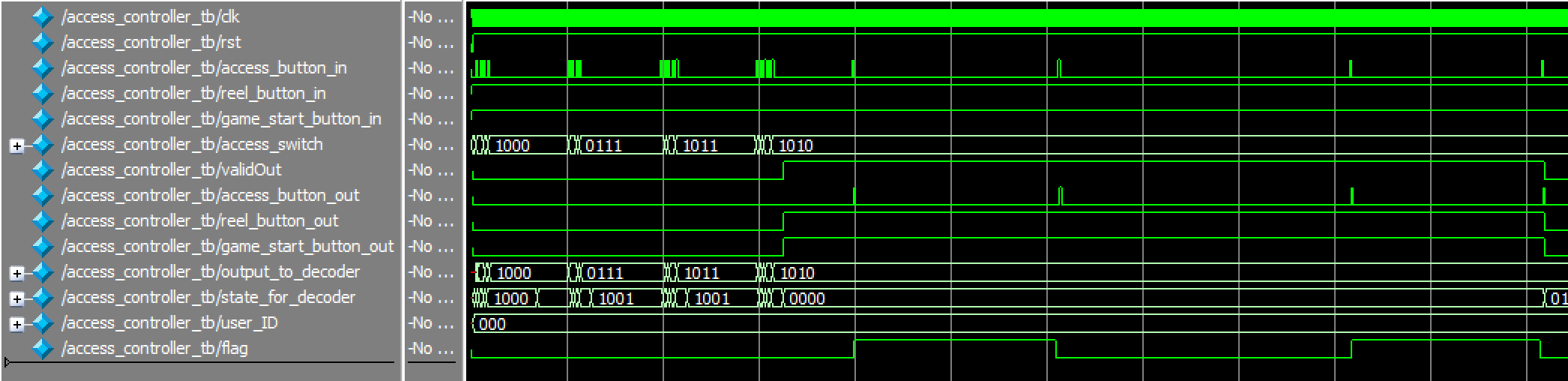
*Outputs:*

* scoreUserAddr0: the score for player 0
* scoreUserAddr1: the score for player 1
* scoreUserAddr2: the score for player 2
* scoreUserAddr3: the score for player 3
* scoreUserAddr4: the score for player 4
* scoreUserAddr5: the score for player 5

**Simulation Results:**

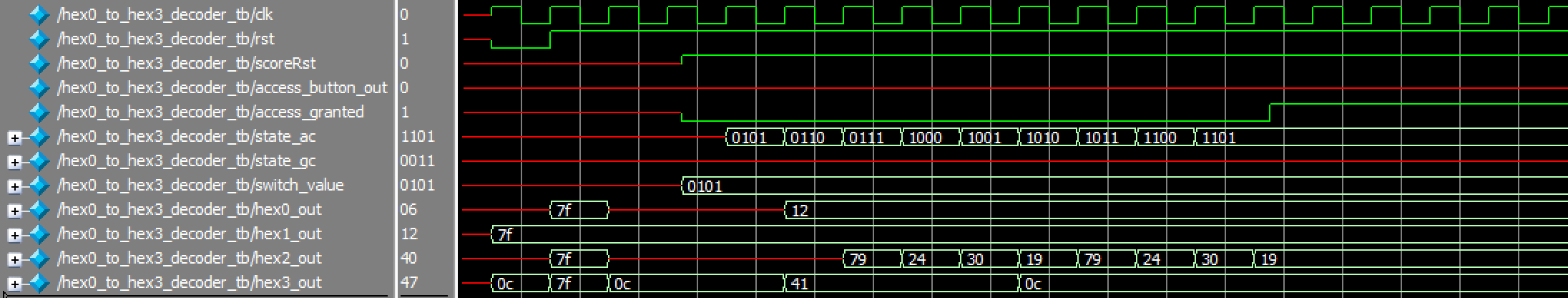
**access\_controller Simulation:**

This testbench inputs the incorrect User ID, correct User ID, incorrect Password, and finally the correct Password. This testbench also ended by testing a long wait between buttons, and shorter wait, when testing a logout. The access\_controller module passed by looking for the password input only when the correct User ID was inputted, only granting access when the correct Password was inputted, and only logging the user out when the period between button presses was 100 clock cycles (100 clock cycles instead of 12.5M was used for testing). Figure 8 below shows these simulation results.

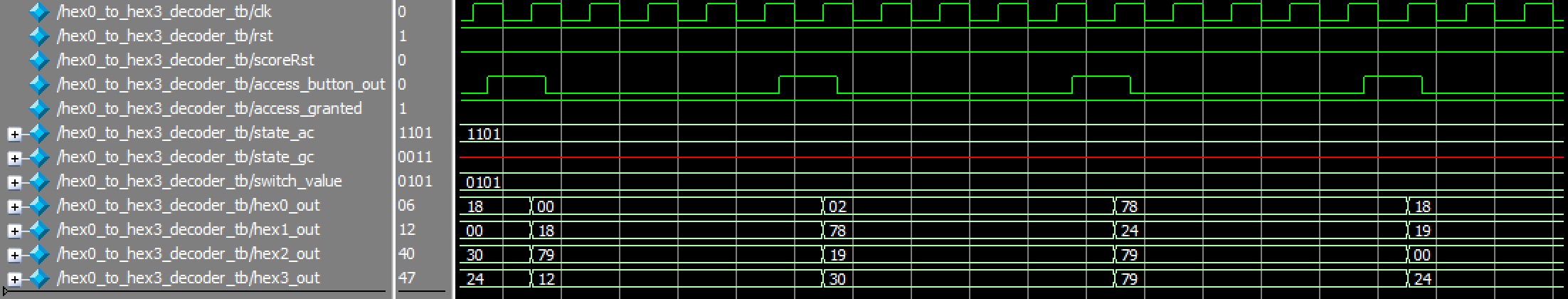


*Figure 8. access\_controller simulation results*

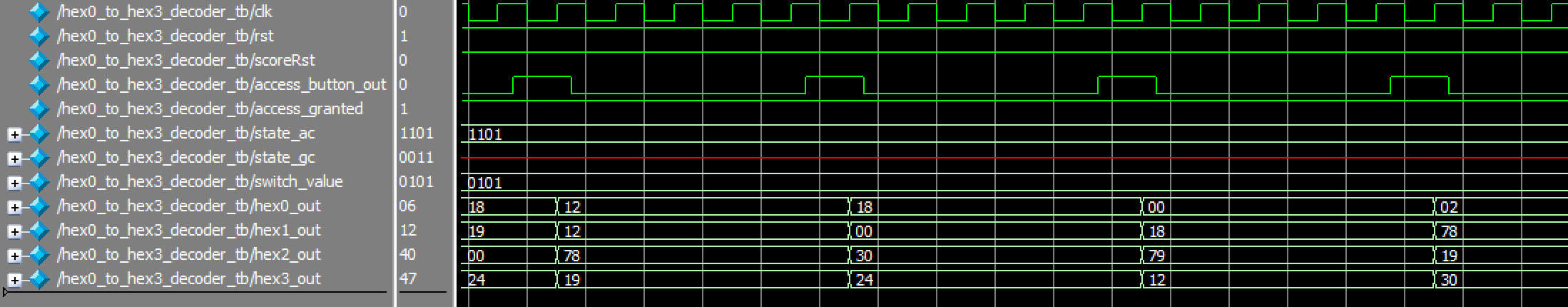
**hex0\_to\_hex3\_decoder Simulation:**

This testbench forces the hex0\_to\_hex3\_decoder module to go through each state of the access\_controller, then round robin’s through all the User ID’s twice, followed by displaying all the gameController states. The hex0\_to\_hex3\_decoder passed the test by properly outputting to the 7-segment displays. Figures 9, 10, 11, 12 below illustrates these simulation results.

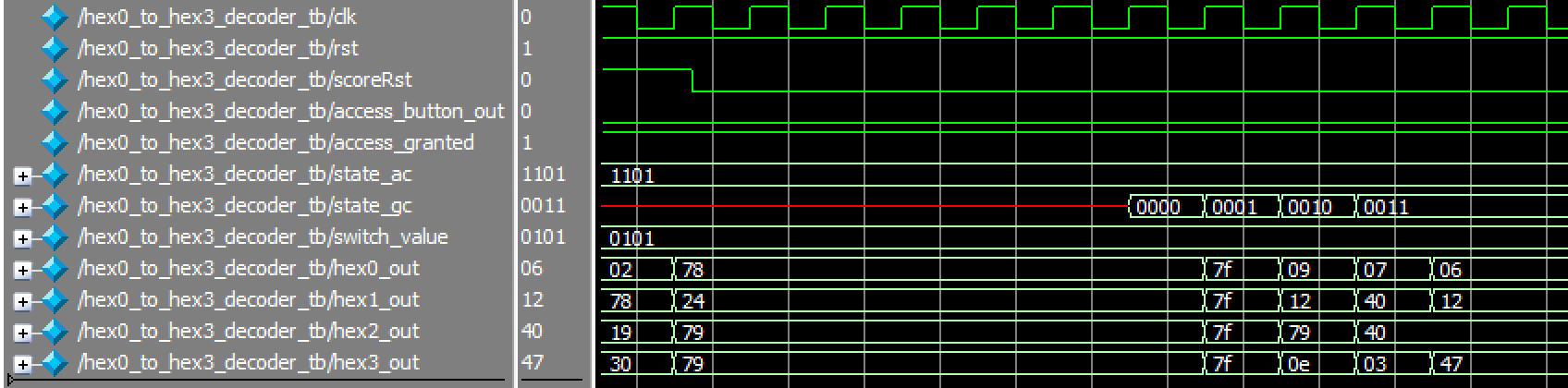
*Figure 9. hex0\_to\_hex3\_decoder simulation results 1*



*Figure 10. hex0\_to\_hex3\_decoder simulation results 2*



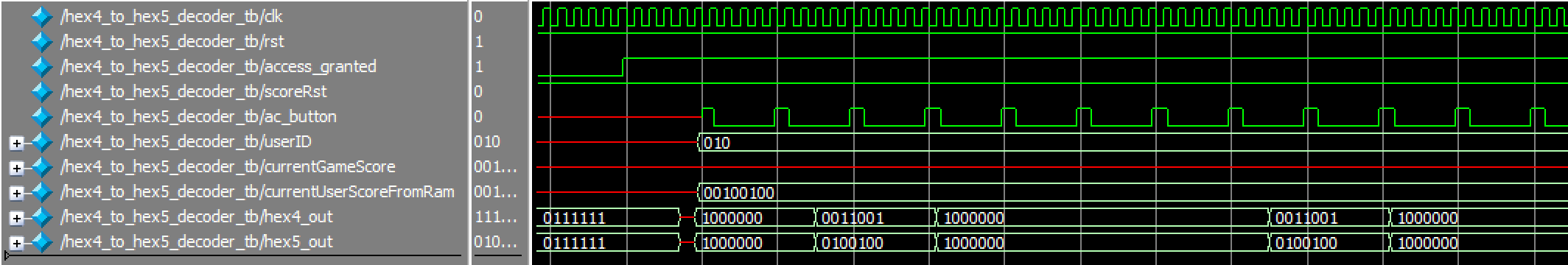
*Figure 11. hex0\_to\_hex3\_decoder simulation results 3*



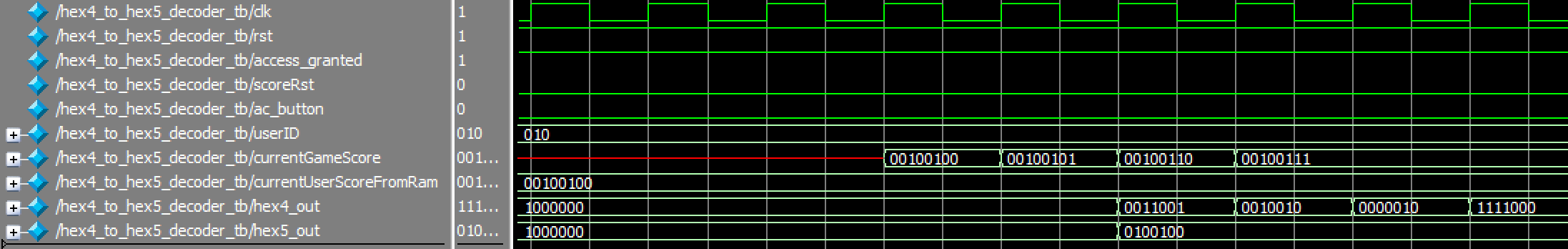
*Figure 12. hex0\_to\_hex3\_decoder simulation results 4*

**hex4\_to\_hex5\_decoder Simulation:**

This testbench forces the hex4\_to\_hex5\_decoder module to go through each state of the access\_controller, then round robin’s through all the User’s scores twice, followed by displaying all the gameController states. The hex4\_to\_hex5\_decoder passed the test by properly outputting to the 7-segment displays. Figures 13 and 14 below illustrate these simulation results.



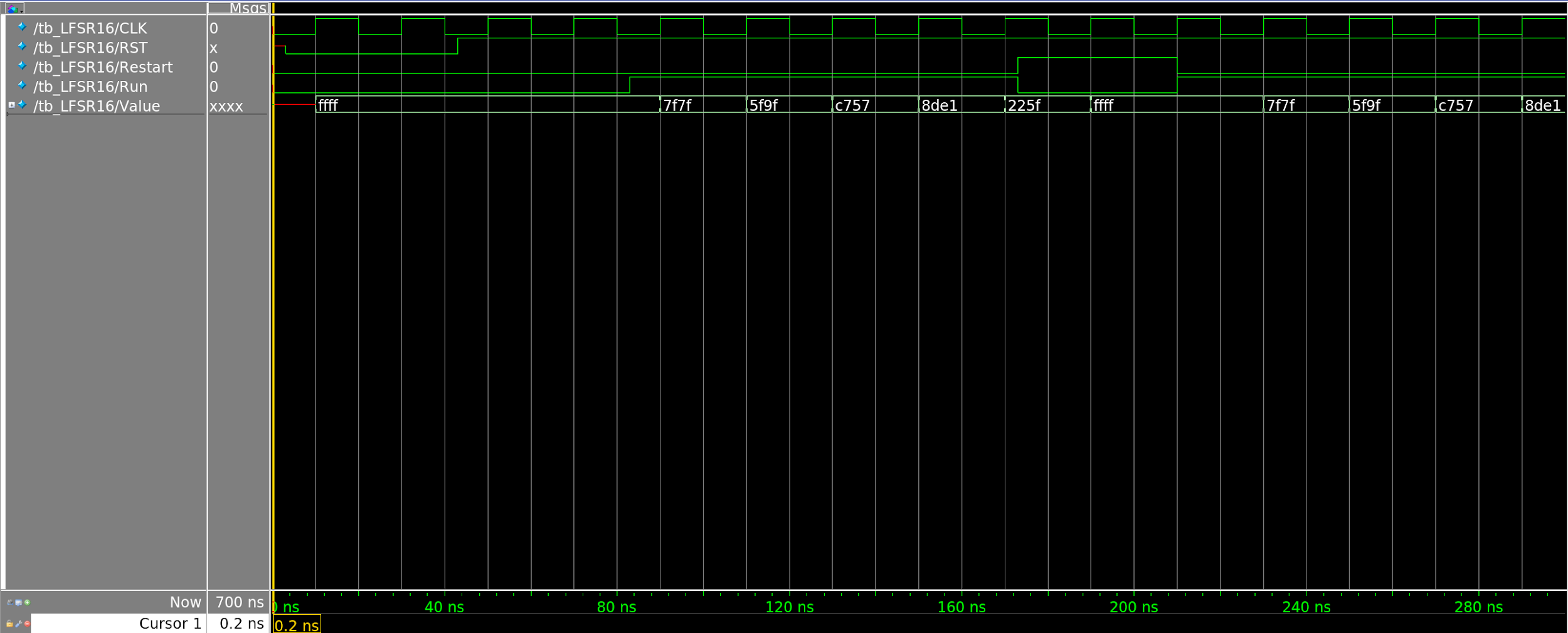
*Figure 13. hex4\_to\_hex5\_decoder simulation results 1*



*Figure 14. hex4\_to\_hex5\_decoder simulation results 2*

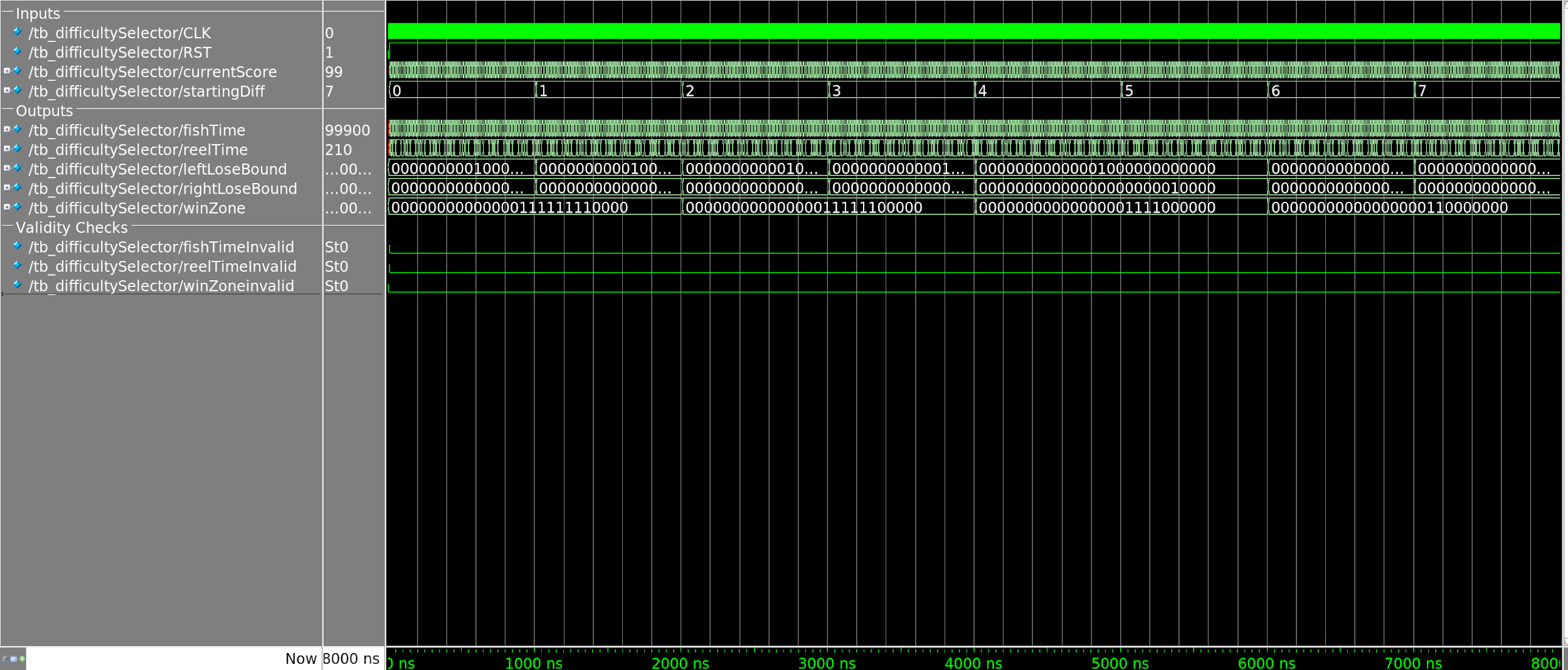
**LFSR16 Simulation:**

This testbench instantiates an LFSR16 module and runs it through the beginning of its output sequence, restarting it in the middle of the test. The output values match those that would be expected for a Xorshift LFSR with the shift triplet <7, 9, 8> and the starting seed 0xFFFF. The waveforms from simulation are shown in Figure 15 below.

*Figure 15. LFSR16 simulation results*

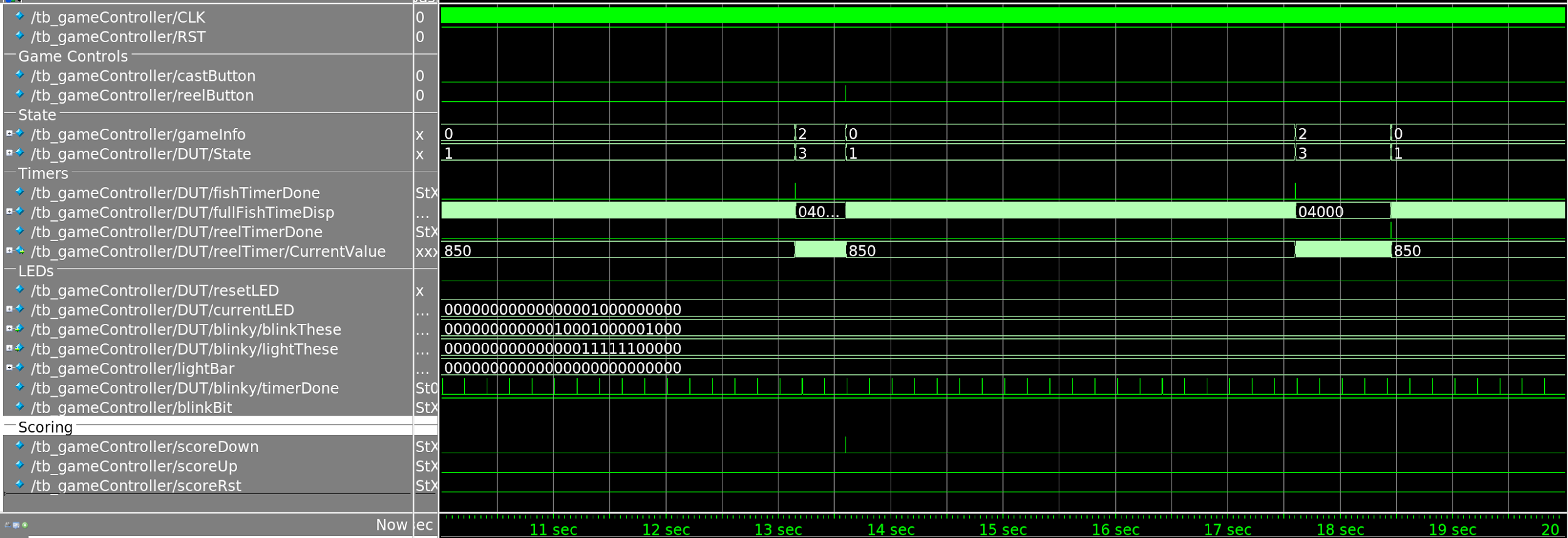
**difficultySelector Simulation:**

This testbench instantiates a difficultySelector module and runs through all possible combinations of score (0-99) and starting difficulty (0-7) to ensure that the output timer values are always valid decimal numbers, which is checked by the fishTimeInvalid and reelTimeInvalid indicators, and that the win zone is between the leftLoseBound and rightLoseBound, which is checked by the winZoneInvalid indicator. The waveforms from simulation are shown in Figure 16 below.

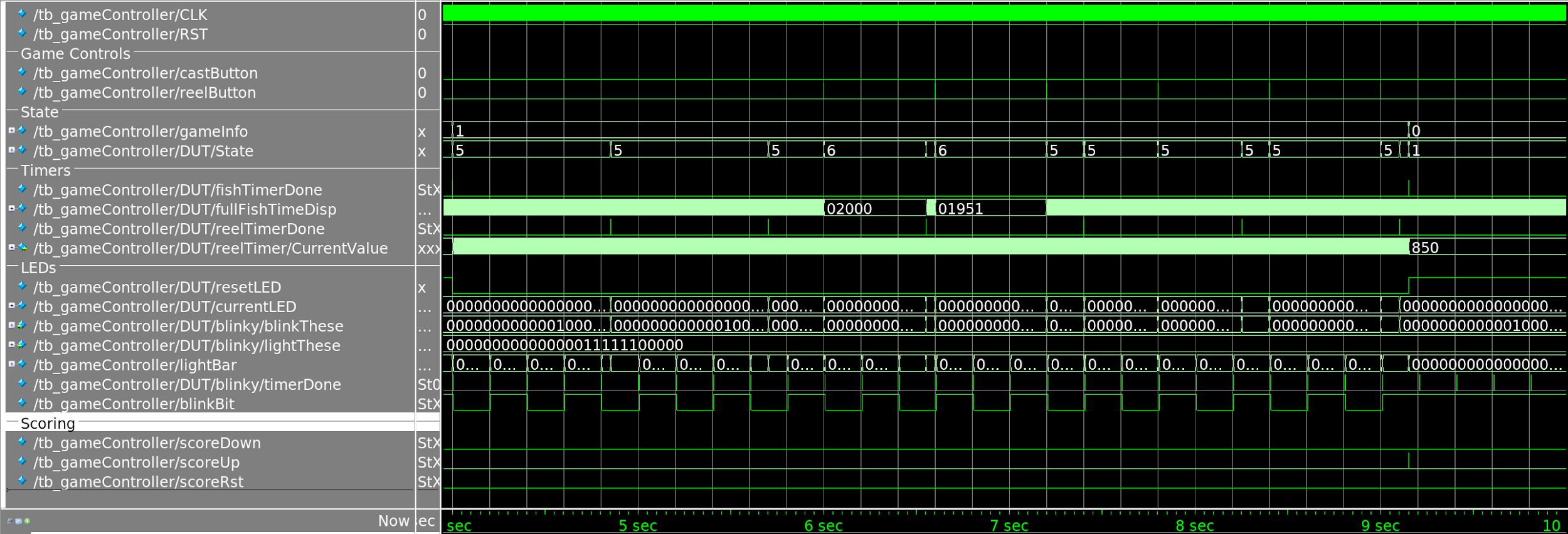
*Figure 16. difficultySelector simulation results*

**gameController Simulation:**

This testbench instantiates a gameController module and checks the functionality of basic gameplay. Figure 17 below shows the process of catching a fish, and Figure 18 shows the process of both catching a boot, and not catching one.



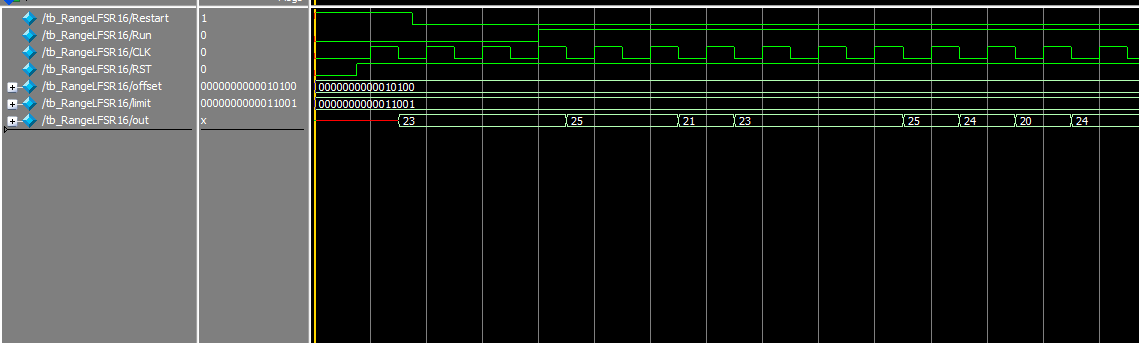
*Figure 17. gameController simulation results 1*



*Figure 18. gameController simulation results 2*

**RangeLFSR16 Simulation:**

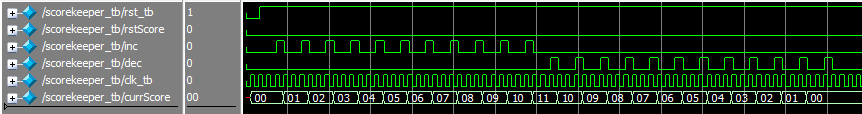
This module is used to test the RangeLFSR16 module, which generates pseudo-random numbers within an inclusive interval. Figure 19, demonstrates the modules ability to generate pseudo-random numbers between a maximum and minimum.



*Figure 19. RangeLFSR16.v simulation results.*

**scorekeeper Simulation:**

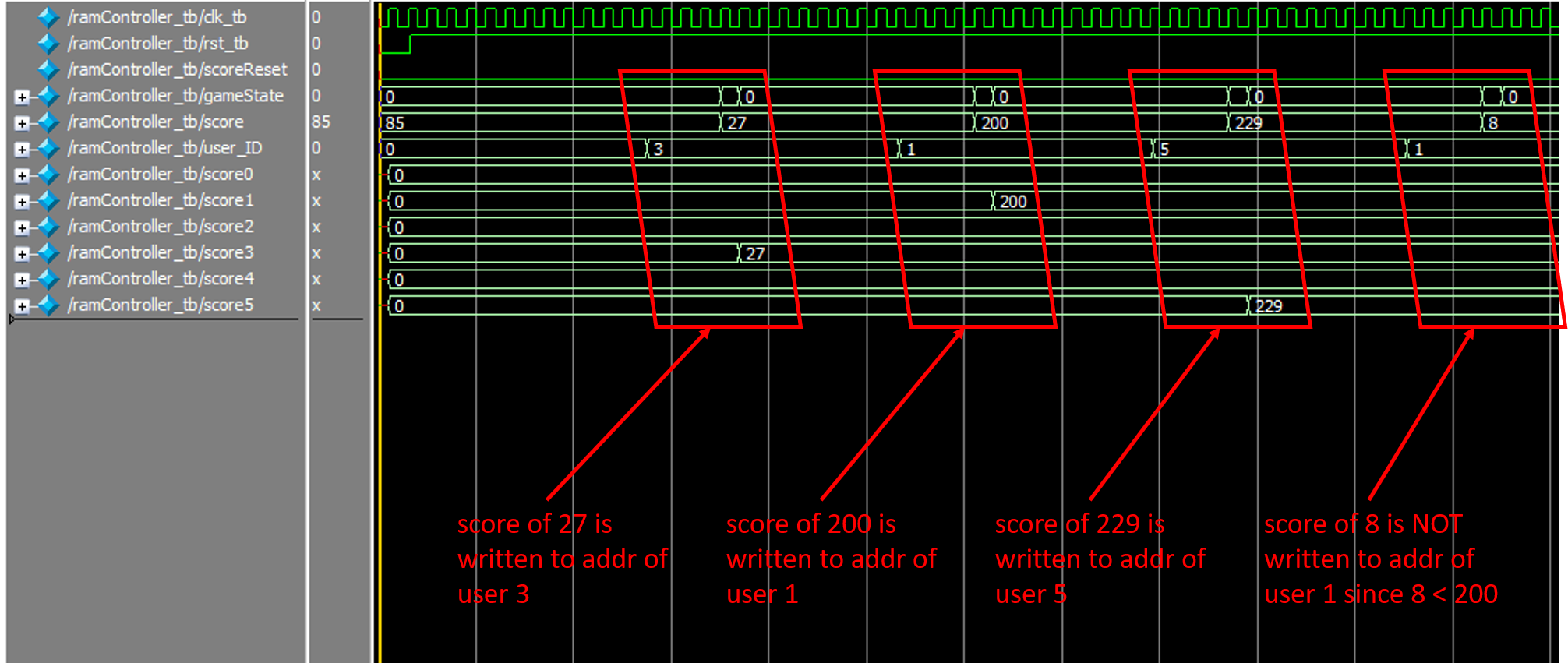
This testbench tests the counting ability of the scorekeeper. It increments the score from 0 to 11, and then decrements the score back down to 0. The results are shown below in Figure 20. Note that currScore is an 8-bit number whose first 4 bits represent the 10s place, and whose second 4 bits represent the 1s place.



*Figure 20. Simulation results for scorekeeper.*

**ramController Simulation:**

This testbench tests the ramController’s ability to save scores when the game state is “done” or “lose.” This is exemplified below in Figure 21. Note that scores are only saved if the new score is higher than the score that is already saved.



*Figure 21. Simulation results for ramController.*

**FPGA Board Testing Results:**

To demonstrate a working solution of the project described in the above documentation, we have created a video demo. This video features the following: multi-user login, all user high-score viewing and updating, fishing game-play at different difficulties, demonstration of logging into another user account and still being able to view other player’s high scores. The link to view this demonstration can be found at: <https://drive.google.com/open?id=1eS9bvTOoNO-AdYisZt4_KHrMWIcWhkp6>.

**Conclusion:**

As shown in the demo video, we were successfully able to complete the proposed game design. We did not complete the bonus feature of password reset, but were able to include the following features: LFSR random number generator, multi-user authentication, game-level difficulty control, LFSR based timer, and RAM storage of user highscores.

**Appendix:**

**FinalProject\_GoneFishIn.v:**

// ECE 5440

// Group: Gone Fishin'

// Module: Top Module

module FinalProject\_GoneFishIn (

// hardware inputs

clk, rst, access\_switch, access\_button\_unshapped, reel\_button\_unshapped, cast\_button\_unshapped,

startingDiff,

// hardware outpts

access\_green\_led, hex0, hex1, hex2, hex3, hex4, hex5, hex6, hex7, lightBar);

input clk, rst, access\_button\_unshapped, reel\_button\_unshapped, cast\_button\_unshapped;

input [3:0] access\_switch;

input [2:0] startingDiff;

output access\_green\_led;

output [6:0] hex0, hex1, hex2, hex3, hex4, hex5, hex6, hex7;

output [25:0] lightBar;

wire access\_button\_shapped, reel\_button\_shapped, cast\_button\_shapped;

wire [3:0] state\_for\_decoder, output\_to\_decoder;

wire access\_button\_out, reel\_button\_out, cast\_button\_out;

wire [3:0] user\_ID;

wire [7:0] currentScore, fishTimeDisp;

wire [3:0] gameInfo;

wire scoreDown, scoreUp, blinkBit, scoreRst;

wire [7:0] scoreUserAddr0, scoreUserAddr1, scoreUserAddr2, scoreUserAddr3, scoreUserAddr4, scoreUserAddr5;

button\_shaper bs1(access\_button\_unshapped, clk, rst, access\_button\_shapped);

button\_shaper bs2(reel\_button\_unshapped, clk, rst, reel\_button\_shapped);

button\_shaper bs3(cast\_button\_unshapped, clk, rst, cast\_button\_shapped);

access\_controller ac(clk, rst, access\_switch, access\_button\_shapped, reel\_button\_shapped, cast\_button\_shapped,

access\_green\_led, state\_for\_decoder, output\_to\_decoder, access\_button\_out, reel\_button\_out, cast\_button\_out, user\_ID);

scorekeeper sk(clk, rst, scoreRst, scoreUp, scoreDown, currentScore);

ramController rC(currentScore,gameInfo,scoreRst,user\_ID,clk,rst,scoreUserAddr0, scoreUserAddr1,

scoreUserAddr2, scoreUserAddr3, scoreUserAddr4, scoreUserAddr5);

gameController gc(access\_green\_led, startingDiff, reel\_button\_out, cast\_button\_out, currentScore,

scoreUp, scoreDown, scoreRst, lightBar, gameInfo, fishTimeDisp, blinkBit, clk, rst);

hex0\_to\_hex3\_decoder decoder(clk, rst, output\_to\_decoder, access\_button\_out, state\_for\_decoder, gameInfo, access\_green\_led, scoreRst, hex0, hex1, hex2, hex3);

hex4\_to\_hex5\_decoder decoder2(clk, rst, access\_button\_out, user\_ID, currentScore, access\_green\_led, scoreRst,

scoreUserAddr0, scoreUserAddr1, scoreUserAddr2, scoreUserAddr3, scoreUserAddr4, scoreUserAddr5, hex4, hex5);

SevenSeg ss3(fishTimeDisp[3:0], hex6);

SevenSeg ss4(fishTimeDisp[7:4], hex7);

endmodule

**access\_controller.v:**

// ECE 5440

// Group: Gone Fishin'

// Module: access\_controller

// This module will block inputs from all buttons except RST to ensure the game can not be triggered untill access is granted.

// To login, the player must enter a user id and password combination from the list below. This module will output the address

// in ROM so the score keeper can update the high score for the user that is logged in. This module will also interact with

// decoder modules to enhance the user experience. If user ID is not valid, the player has unlimited attempts, as well if the

// password does not match the user ID, the player as unlimited attempts at the password without having to enter the user ID

// again.

// u/p setup:

// Fazal: u=1-1-2-7, p=7-2-1-1-A, addr=0

// Jon: u=2-8-4-9, p=9-4-8-2-B, addr=1

// George: u=4-7-5-5, p=5-5-7-4-C, addr=2

// Nick: u=2-3-8-9, p=9-8-3-2-D, addr=3

// Nathan: u=5-1-9-8, p=8-9-1-5-E, addr=4

// Guest: u=3-4-7-6, p=6-7-4-3-F, addr=5

module access\_controller(

// sequential logic variables

clk, rst,

// password digit switch input

access\_switch,

// user input buttons

access\_button\_in, reel\_button\_in, game\_start\_button\_in,

// output LED/ game controller trigger variable

validOut,

// output to 7-seg decoder

state\_for\_decoder, output\_to\_decoder,

// output button's based on access state

access\_button\_out, reel\_button\_out, game\_start\_button\_out,

// output user ID to score keeper

user\_ID);

// input variables

input clk, rst, access\_button\_in, reel\_button\_in, game\_start\_button\_in;

input [3:0] access\_switch;

// output variables

output reg validOut, access\_button\_out, reel\_button\_out, game\_start\_button\_out;

output reg [3:0] output\_to\_decoder, state\_for\_decoder;

output reg [2:0] user\_ID;

// accessing User ROM

reg [5:0] addr\_user; // 6-bit wide

wire [15:0] data\_user; // 16-bit wide

reg [15:0] user\_ID\_raw; // 16-bit wide

// access Pass ROM

reg [5:0] addr\_pass; // 6-bit wide

wire [19:0] data\_pass; // 20-bit wide

reg [19:0] pass\_raw; // 20-bit wide

// 1 instances of ROM user retrieval module

User\_ROM romU(addr\_user, clk, data\_user);

// 1 instances of ROM password retrieval module

Password\_ROM romP(addr\_pass, clk, data\_pass);

// helper variables

reg [4:0] state, state\_callback;

reg check\_rom;

integer clk\_counter;

// read 4 digits of user ID input

parameter state\_user\_0 = 5'b00000, state\_user\_1 = 5'b00001;

parameter state\_user\_2 = 5'b00010, state\_user\_3 = 5'b00011;

// round robin 6 ROM address to find match for User ID

parameter state\_user\_rom\_0 = 5'b00100, state\_user\_rom\_1 = 5'b00101;

parameter state\_user\_rom\_2 = 5'b00110, state\_user\_rom\_3 = 5'b00111;

parameter state\_user\_rom\_4 = 5'b01000, state\_user\_rom\_5 = 5'b01001;

// read 5 digits of password input

parameter state\_pass\_0 = 5'b01010, state\_pass\_1 = 5'b01011;

parameter state\_pass\_2 = 5'b01100, state\_pass\_3 = 5'b01101;

parameter state\_pass\_4 = 5'b01110;

// no need to round robin, simiply check same addres that user\_data was a match

parameter state\_pass\_rom = 5'b01111;

// wait for ROM to update Data with current addr value

parameter state\_wait\_1 = 5'b10000, state\_wait\_2 = 5'b10001;

// access granted

parameter state\_access\_granted = 5'b10010;

always @(posedge clk) begin

if(rst == 0) begin

validOut <= 0;

user\_ID <= 0;

user\_ID\_raw <= 0;

pass\_raw <= 0;

check\_rom <= 0;

access\_button\_out <= 0;

reel\_button\_out <= 0;

game\_start\_button\_out <= 0;

state\_for\_decoder <= 0;

state <= state\_user\_0;

end

else begin

case (state)

/\*----Waiting on ROM to update Data---\*/

state\_wait\_1: begin

state <= state\_wait\_2;

end

state\_wait\_2: begin

state <= state\_callback;

check\_rom <= 1;

end

/\*----Reading in User ID Input---\*/

state\_user\_0: begin

validOut <= 0;

user\_ID <= 0;

user\_ID\_raw = 0; // need to make this blocking

pass\_raw <= 0;

check\_rom <= 0;

access\_button\_out <= 0;

reel\_button\_out <= 0;

game\_start\_button\_out <= 0;

state\_for\_decoder <= 4'b0101;

output\_to\_decoder <= access\_switch;

if(access\_button\_in == 1) begin

user\_ID\_raw = user\_ID\_raw + access\_switch;

user\_ID\_raw = user\_ID\_raw << 4;

state = state\_user\_1;

end

end

state\_user\_1: begin

state\_for\_decoder <= 4'b0110;

output\_to\_decoder <= access\_switch;

if(access\_button\_in == 1) begin

user\_ID\_raw = user\_ID\_raw + access\_switch;

user\_ID\_raw = user\_ID\_raw << 4;

state = state\_user\_2;

end

end

state\_user\_2: begin

state\_for\_decoder <= 4'b0111;

output\_to\_decoder <= access\_switch;

if(access\_button\_in == 1) begin

user\_ID\_raw = user\_ID\_raw + access\_switch;

user\_ID\_raw = user\_ID\_raw << 4;

state = state\_user\_3;

end

end

state\_user\_3: begin

state\_for\_decoder <= 4'b1000;

output\_to\_decoder <= access\_switch;

if(access\_button\_in == 1) begin

user\_ID\_raw = user\_ID\_raw + access\_switch;

// no need to shift left since thats the final 4-bits in user ID

state = state\_user\_rom\_0;

end

end

/\*----Locating User ID in ROM---\*/

state\_user\_rom\_0: begin

state\_callback <= state\_user\_rom\_0;

state <= state\_wait\_1;

addr\_user <= 0;

if(check\_rom == 1) begin

check\_rom <= 0;

if(user\_ID\_raw == data\_user) begin

// then we can go straight to reading password

user\_ID <= addr\_user;

state <= state\_pass\_0;

end

else begin

state <= state\_user\_rom\_1;

end

end

end

state\_user\_rom\_1: begin

state\_callback <= state\_user\_rom\_1;

state <= state\_wait\_1;

addr\_user <= 1;

if(check\_rom == 1) begin

check\_rom <= 0;

if(user\_ID\_raw == data\_user) begin

// then we can go straight to reading password

user\_ID <= addr\_user;

state <= state\_pass\_0;

end

else begin

state <= state\_user\_rom\_2;

end

end

end

state\_user\_rom\_2: begin

state\_callback <= state\_user\_rom\_2;

state <= state\_wait\_1;

addr\_user <= 2;

if(check\_rom == 1) begin

check\_rom <= 0;

if(user\_ID\_raw == data\_user) begin

// then we can go straight to reading password

user\_ID <= addr\_user;

state <= state\_pass\_0;

end

else begin

state <= state\_user\_rom\_3;

end

end

end

state\_user\_rom\_3: begin

state\_callback <= state\_user\_rom\_3;

state <= state\_wait\_1;

addr\_user <= 3;

if(check\_rom == 1) begin

check\_rom <= 0;

if(user\_ID\_raw == data\_user) begin

// then we can go straight to reading password

user\_ID <= addr\_user;

state <= state\_pass\_0;

end

else begin

state <= state\_user\_rom\_4;

end

end

end

state\_user\_rom\_4: begin

state\_callback <= state\_user\_rom\_4;

state <= state\_wait\_1;

addr\_user <= 4;

if(check\_rom == 1) begin

check\_rom <= 0;

if(user\_ID\_raw == data\_user) begin

// then we can go straight to reading password

user\_ID <= addr\_user;

state <= state\_pass\_0;

end

else begin

state <= state\_user\_rom\_5;

end

end

end

state\_user\_rom\_5: begin

state\_callback <= state\_user\_rom\_5;

state <= state\_wait\_1;

addr\_user <= 5;

if(check\_rom == 1) begin

check\_rom <= 0;

if(user\_ID\_raw == data\_user) begin

// then we can go straight to reading password

user\_ID <= addr\_user;

state <= state\_pass\_0;

end

else begin

// user ID does not match any ROM registers, go back to first user state

state <= state\_user\_0;

end

end

end

/\*----Reading in Password Input---\*/

state\_pass\_0: begin

pass\_raw = 0; // reset in case multiple password entries

state\_for\_decoder <= 4'b1001;

output\_to\_decoder <= access\_switch;

if(access\_button\_in == 1) begin

pass\_raw = pass\_raw + access\_switch;

pass\_raw = pass\_raw << 4;

state = state\_pass\_1;

end

end

state\_pass\_1: begin

state\_for\_decoder <= 4'b1010;

output\_to\_decoder <= access\_switch;

if(access\_button\_in == 1) begin

pass\_raw = pass\_raw + access\_switch;

pass\_raw = pass\_raw << 4;

state = state\_pass\_2;

end

end

state\_pass\_2: begin

state\_for\_decoder <= 4'b1011;

output\_to\_decoder <= access\_switch;

if(access\_button\_in == 1) begin

pass\_raw = pass\_raw + access\_switch;

pass\_raw = pass\_raw << 4;

state = state\_pass\_3;

end

end

state\_pass\_3: begin

state\_for\_decoder <= 4'b1100;

output\_to\_decoder <= access\_switch;

if(access\_button\_in == 1) begin

pass\_raw = pass\_raw + access\_switch;

pass\_raw = pass\_raw << 4;

state = state\_pass\_4;

end

end

state\_pass\_4: begin

state\_for\_decoder <= 4'b1101;

output\_to\_decoder <= access\_switch;

if(access\_button\_in == 1) begin

pass\_raw = pass\_raw + access\_switch;

// no need to shift left since thats the final 4-bits in pass

state = state\_pass\_rom;

end

end

/\*----Determining if Password inputted matched in same ROM address as User ID---\*/

state\_pass\_rom: begin

state\_callback <= state\_pass\_rom;

state <= state\_wait\_1;

addr\_pass <= user\_ID;

if(check\_rom == 1) begin

check\_rom <= 0;

if(pass\_raw == data\_pass) begin

// pass matched, grant access

state <= state\_access\_granted;

end

else begin

// pass did not match ROM, allow re-entering password

state <= state\_pass\_0;

end

end

end

/\*----Granting access to remainder of system---\*/

state\_access\_granted: begin

validOut <= 1;

access\_button\_out <= access\_button\_in;

reel\_button\_out <= reel\_button\_in;

game\_start\_button\_out <= game\_start\_button\_in;

state\_for\_decoder <= 0;

/\*----Looking for logout---\*/

if(access\_button\_in == 1 && clk\_counter > 0) begin

// logout!

clk\_counter <= 0;

state <= state\_user\_0;

end

else if(access\_button\_in == 1 && clk\_counter == 0) begin

// first button click

clk\_counter <= clk\_counter + 1;

end

else if(clk\_counter == 12500000) begin //12.5M = 1/4 second

// if its been longer than .25 seconds, dont try to log out user

clk\_counter <= 0;

end

else if(clk\_counter > 0) begin

// if counter is increased (by first button click), keep increasing

clk\_counter <= clk\_counter + 1;

end

end

endcase

end

end

endmodule

**User\_ROM.v:**

// megafunction wizard: %ROM: 1-PORT%

// GENERATION: STANDARD

// VERSION: WM1.0

// MODULE: altsyncram

// ============================================================

// File Name: User\_ROM.v

// Megafunction Name(s):

// altsyncram

//

// Simulation Library Files(s):

// altera\_mf

// ============================================================

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// THIS IS A WIZARD-GENERATED FILE. DO NOT EDIT THIS FILE!

//

// 18.1.0 Build 625 09/12/2018 SJ Lite Edition

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//Copyright (C) 2018 Intel Corporation. All rights reserved.

//Your use of Intel Corporation's design tools, logic functions

//and other software and tools, and its AMPP partner logic

//functions, and any output files from any of the foregoing

//(including device programming or simulation files), and any

//associated documentation or information are expressly subject

//to the terms and conditions of the Intel Program License

//Subscription Agreement, the Intel Quartus Prime License Agreement,

//the Intel FPGA IP License Agreement, or other applicable license

//agreement, including, without limitation, that your use is for

//the sole purpose of programming logic devices manufactured by

//Intel and sold by Intel or its authorized distributors. Please

//refer to the applicable agreement for further details.

// synopsys translate\_off

`timescale 1 ps / 1 ps

// synopsys translate\_on

module User\_ROM (

address,

clock,

q);

input [5:0] address;

input clock;

output [15:0] q;

`ifndef ALTERA\_RESERVED\_QIS

// synopsys translate\_off

`endif

tri1 clock;

`ifndef ALTERA\_RESERVED\_QIS

// synopsys translate\_on

`endif

wire [15:0] sub\_wire0;

wire [15:0] q = sub\_wire0[15:0];

altsyncram altsyncram\_component (

.address\_a (address),

.clock0 (clock),

.q\_a (sub\_wire0),

.aclr0 (1'b0),

.aclr1 (1'b0),

.address\_b (1'b1),

.addressstall\_a (1'b0),

.addressstall\_b (1'b0),

.byteena\_a (1'b1),

.byteena\_b (1'b1),

.clock1 (1'b1),

.clocken0 (1'b1),

.clocken1 (1'b1),

.clocken2 (1'b1),

.clocken3 (1'b1),

.data\_a ({16{1'b1}}),

.data\_b (1'b1),

.eccstatus (),

.q\_b (),

.rden\_a (1'b1),

.rden\_b (1'b1),

.wren\_a (1'b0),

.wren\_b (1'b0));

defparam

altsyncram\_component.address\_aclr\_a = "NONE",

altsyncram\_component.clock\_enable\_input\_a = "BYPASS",

altsyncram\_component.clock\_enable\_output\_a = "BYPASS",

altsyncram\_component.init\_file = "User\_ROM.mif",

altsyncram\_component.intended\_device\_family = "Cyclone IV E",

altsyncram\_component.lpm\_hint = "ENABLE\_RUNTIME\_MOD=NO",

altsyncram\_component.lpm\_type = "altsyncram",

altsyncram\_component.numwords\_a = 64,

altsyncram\_component.operation\_mode = "ROM",

altsyncram\_component.outdata\_aclr\_a = "NONE",

altsyncram\_component.outdata\_reg\_a = "CLOCK0",

altsyncram\_component.widthad\_a = 6,

altsyncram\_component.width\_a = 16,

altsyncram\_component.width\_byteena\_a = 1;

endmodule

// ============================================================

// CNX file retrieval info

// ============================================================

// Retrieval info: PRIVATE: ADDRESSSTALL\_A NUMERIC "0"

// Retrieval info: PRIVATE: AclrAddr NUMERIC "0"

// Retrieval info: PRIVATE: AclrByte NUMERIC "0"

// Retrieval info: PRIVATE: AclrOutput NUMERIC "0"

// Retrieval info: PRIVATE: BYTE\_ENABLE NUMERIC "0"

// Retrieval info: PRIVATE: BYTE\_SIZE NUMERIC "8"

// Retrieval info: PRIVATE: BlankMemory NUMERIC "0"

// Retrieval info: PRIVATE: CLOCK\_ENABLE\_INPUT\_A NUMERIC "0"

// Retrieval info: PRIVATE: CLOCK\_ENABLE\_OUTPUT\_A NUMERIC "0"

// Retrieval info: PRIVATE: Clken NUMERIC "0"

// Retrieval info: PRIVATE: IMPLEMENT\_IN\_LES NUMERIC "0"

// Retrieval info: PRIVATE: INIT\_FILE\_LAYOUT STRING "PORT\_A"

// Retrieval info: PRIVATE: INIT\_TO\_SIM\_X NUMERIC "0"

// Retrieval info: PRIVATE: INTENDED\_DEVICE\_FAMILY STRING "Cyclone IV E"

// Retrieval info: PRIVATE: JTAG\_ENABLED NUMERIC "0"

// Retrieval info: PRIVATE: JTAG\_ID STRING "NONE"

// Retrieval info: PRIVATE: MAXIMUM\_DEPTH NUMERIC "0"

// Retrieval info: PRIVATE: MIFfilename STRING "../src/User\_ROM.mif"

// Retrieval info: PRIVATE: NUMWORDS\_A NUMERIC "64"

// Retrieval info: PRIVATE: RAM\_BLOCK\_TYPE NUMERIC "0"

// Retrieval info: PRIVATE: RegAddr NUMERIC "1"

// Retrieval info: PRIVATE: RegOutput NUMERIC "1"

// Retrieval info: PRIVATE: SYNTH\_WRAPPER\_GEN\_POSTFIX STRING "0"

// Retrieval info: PRIVATE: SingleClock NUMERIC "1"

// Retrieval info: PRIVATE: UseDQRAM NUMERIC "0"

// Retrieval info: PRIVATE: WidthAddr NUMERIC "6"

// Retrieval info: PRIVATE: WidthData NUMERIC "16"

// Retrieval info: PRIVATE: rden NUMERIC "0"

// Retrieval info: LIBRARY: altera\_mf altera\_mf.altera\_mf\_components.all

// Retrieval info: CONSTANT: ADDRESS\_ACLR\_A STRING "NONE"

// Retrieval info: CONSTANT: CLOCK\_ENABLE\_INPUT\_A STRING "BYPASS"

// Retrieval info: CONSTANT: CLOCK\_ENABLE\_OUTPUT\_A STRING "BYPASS"

// Retrieval info: CONSTANT: INIT\_FILE STRING "../src/User\_ROM.mif"

// Retrieval info: CONSTANT: INTENDED\_DEVICE\_FAMILY STRING "Cyclone IV E"

// Retrieval info: CONSTANT: LPM\_HINT STRING "ENABLE\_RUNTIME\_MOD=NO"

// Retrieval info: CONSTANT: LPM\_TYPE STRING "altsyncram"

// Retrieval info: CONSTANT: NUMWORDS\_A NUMERIC "64"

// Retrieval info: CONSTANT: OPERATION\_MODE STRING "ROM"

// Retrieval info: CONSTANT: OUTDATA\_ACLR\_A STRING "NONE"

// Retrieval info: CONSTANT: OUTDATA\_REG\_A STRING "CLOCK0"

// Retrieval info: CONSTANT: WIDTHAD\_A NUMERIC "6"

// Retrieval info: CONSTANT: WIDTH\_A NUMERIC "16"

// Retrieval info: CONSTANT: WIDTH\_BYTEENA\_A NUMERIC "1"

// Retrieval info: USED\_PORT: address 0 0 6 0 INPUT NODEFVAL "address[5..0]"

// Retrieval info: USED\_PORT: clock 0 0 0 0 INPUT VCC "clock"

// Retrieval info: USED\_PORT: q 0 0 16 0 OUTPUT NODEFVAL "q[15..0]"

// Retrieval info: CONNECT: @address\_a 0 0 6 0 address 0 0 6 0

// Retrieval info: CONNECT: @clock0 0 0 0 0 clock 0 0 0 0

// Retrieval info: CONNECT: q 0 0 16 0 @q\_a 0 0 16 0

// Retrieval info: GEN\_FILE: TYPE\_NORMAL User\_ROM.v TRUE

// Retrieval info: GEN\_FILE: TYPE\_NORMAL User\_ROM.inc FALSE

// Retrieval info: GEN\_FILE: TYPE\_NORMAL User\_ROM.cmp FALSE

// Retrieval info: GEN\_FILE: TYPE\_NORMAL User\_ROM.bsf FALSE

// Retrieval info: GEN\_FILE: TYPE\_NORMAL User\_ROM\_inst.v FALSE

// Retrieval info: GEN\_FILE: TYPE\_NORMAL User\_ROM\_bb.v TRUE

// Retrieval info: LIB\_FILE: altera\_mf

**Password\_ROM.v:**

// megafunction wizard: %ROM: 1-PORT%

// GENERATION: STANDARD

// VERSION: WM1.0

// MODULE: altsyncram

// ============================================================

// File Name: Password\_ROM.v

// Megafunction Name(s):

// altsyncram

//

// Simulation Library Files(s):

// altera\_mf

// ============================================================

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// THIS IS A WIZARD-GENERATED FILE. DO NOT EDIT THIS FILE!

//

// 18.1.0 Build 625 09/12/2018 SJ Lite Edition

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//Copyright (C) 2018 Intel Corporation. All rights reserved.

//Your use of Intel Corporation's design tools, logic functions

//and other software and tools, and its AMPP partner logic

//functions, and any output files from any of the foregoing

//(including device programming or simulation files), and any

//associated documentation or information are expressly subject

//to the terms and conditions of the Intel Program License

//Subscription Agreement, the Intel Quartus Prime License Agreement,

//the Intel FPGA IP License Agreement, or other applicable license

//agreement, including, without limitation, that your use is for

//the sole purpose of programming logic devices manufactured by

//Intel and sold by Intel or its authorized distributors. Please

//refer to the applicable agreement for further details.

// synopsys translate\_off

`timescale 1 ps / 1 ps

// synopsys translate\_on

module Password\_ROM (

address,

clock,

q);

input [5:0] address;

input clock;

output [19:0] q;

`ifndef ALTERA\_RESERVED\_QIS

// synopsys translate\_off

`endif

tri1 clock;

`ifndef ALTERA\_RESERVED\_QIS

// synopsys translate\_on

`endif

wire [19:0] sub\_wire0;

wire [19:0] q = sub\_wire0[19:0];

altsyncram altsyncram\_component (

.address\_a (address),

.clock0 (clock),

.q\_a (sub\_wire0),

.aclr0 (1'b0),

.aclr1 (1'b0),

.address\_b (1'b1),

.addressstall\_a (1'b0),

.addressstall\_b (1'b0),

.byteena\_a (1'b1),

.byteena\_b (1'b1),

.clock1 (1'b1),

.clocken0 (1'b1),

.clocken1 (1'b1),

.clocken2 (1'b1),

.clocken3 (1'b1),

.data\_a ({20{1'b1}}),

.data\_b (1'b1),

.eccstatus (),

.q\_b (),

.rden\_a (1'b1),

.rden\_b (1'b1),

.wren\_a (1'b0),

.wren\_b (1'b0));

defparam

altsyncram\_component.address\_aclr\_a = "NONE",

altsyncram\_component.clock\_enable\_input\_a = "BYPASS",

altsyncram\_component.clock\_enable\_output\_a = "BYPASS",

altsyncram\_component.init\_file = "Password\_ROM.mif",

altsyncram\_component.intended\_device\_family = "Cyclone IV E",

altsyncram\_component.lpm\_hint = "ENABLE\_RUNTIME\_MOD=NO",

altsyncram\_component.lpm\_type = "altsyncram",

altsyncram\_component.numwords\_a = 64,

altsyncram\_component.operation\_mode = "ROM",

altsyncram\_component.outdata\_aclr\_a = "NONE",

altsyncram\_component.outdata\_reg\_a = "CLOCK0",

altsyncram\_component.widthad\_a = 6,

altsyncram\_component.width\_a = 20,

altsyncram\_component.width\_byteena\_a = 1;

endmodule

// ============================================================

// CNX file retrieval info

// ============================================================

// Retrieval info: PRIVATE: ADDRESSSTALL\_A NUMERIC "0"

// Retrieval info: PRIVATE: AclrAddr NUMERIC "0"

// Retrieval info: PRIVATE: AclrByte NUMERIC "0"

// Retrieval info: PRIVATE: AclrOutput NUMERIC "0"

// Retrieval info: PRIVATE: BYTE\_ENABLE NUMERIC "0"

// Retrieval info: PRIVATE: BYTE\_SIZE NUMERIC "8"

// Retrieval info: PRIVATE: BlankMemory NUMERIC "0"

// Retrieval info: PRIVATE: CLOCK\_ENABLE\_INPUT\_A NUMERIC "0"

// Retrieval info: PRIVATE: CLOCK\_ENABLE\_OUTPUT\_A NUMERIC "0"

// Retrieval info: PRIVATE: Clken NUMERIC "0"

// Retrieval info: PRIVATE: IMPLEMENT\_IN\_LES NUMERIC "0"

// Retrieval info: PRIVATE: INIT\_FILE\_LAYOUT STRING "PORT\_A"

// Retrieval info: PRIVATE: INIT\_TO\_SIM\_X NUMERIC "0"

// Retrieval info: PRIVATE: INTENDED\_DEVICE\_FAMILY STRING "Cyclone IV E"

// Retrieval info: PRIVATE: JTAG\_ENABLED NUMERIC "0"

// Retrieval info: PRIVATE: JTAG\_ID STRING "NONE"

// Retrieval info: PRIVATE: MAXIMUM\_DEPTH NUMERIC "0"

// Retrieval info: PRIVATE: MIFfilename STRING "../src/Password\_ROM.mif"

// Retrieval info: PRIVATE: NUMWORDS\_A NUMERIC "64"

// Retrieval info: PRIVATE: RAM\_BLOCK\_TYPE NUMERIC "0"

// Retrieval info: PRIVATE: RegAddr NUMERIC "1"

// Retrieval info: PRIVATE: RegOutput NUMERIC "1"

// Retrieval info: PRIVATE: SYNTH\_WRAPPER\_GEN\_POSTFIX STRING "0"

// Retrieval info: PRIVATE: SingleClock NUMERIC "1"

// Retrieval info: PRIVATE: UseDQRAM NUMERIC "0"

// Retrieval info: PRIVATE: WidthAddr NUMERIC "6"

// Retrieval info: PRIVATE: WidthData NUMERIC "20"

// Retrieval info: PRIVATE: rden NUMERIC "0"

// Retrieval info: LIBRARY: altera\_mf altera\_mf.altera\_mf\_components.all

// Retrieval info: CONSTANT: ADDRESS\_ACLR\_A STRING "NONE"

// Retrieval info: CONSTANT: CLOCK\_ENABLE\_INPUT\_A STRING "BYPASS"

// Retrieval info: CONSTANT: CLOCK\_ENABLE\_OUTPUT\_A STRING "BYPASS"

// Retrieval info: CONSTANT: INIT\_FILE STRING "../src/Password\_ROM.mif"

// Retrieval info: CONSTANT: INTENDED\_DEVICE\_FAMILY STRING "Cyclone IV E"

// Retrieval info: CONSTANT: LPM\_HINT STRING "ENABLE\_RUNTIME\_MOD=NO"

// Retrieval info: CONSTANT: LPM\_TYPE STRING "altsyncram"

// Retrieval info: CONSTANT: NUMWORDS\_A NUMERIC "64"

// Retrieval info: CONSTANT: OPERATION\_MODE STRING "ROM"

// Retrieval info: CONSTANT: OUTDATA\_ACLR\_A STRING "NONE"

// Retrieval info: CONSTANT: OUTDATA\_REG\_A STRING "CLOCK0"

// Retrieval info: CONSTANT: WIDTHAD\_A NUMERIC "6"

// Retrieval info: CONSTANT: WIDTH\_A NUMERIC "20"

// Retrieval info: CONSTANT: WIDTH\_BYTEENA\_A NUMERIC "1"

// Retrieval info: USED\_PORT: address 0 0 6 0 INPUT NODEFVAL "address[5..0]"

// Retrieval info: USED\_PORT: clock 0 0 0 0 INPUT VCC "clock"

// Retrieval info: USED\_PORT: q 0 0 20 0 OUTPUT NODEFVAL "q[19..0]"

// Retrieval info: CONNECT: @address\_a 0 0 6 0 address 0 0 6 0

// Retrieval info: CONNECT: @clock0 0 0 0 0 clock 0 0 0 0

// Retrieval info: CONNECT: q 0 0 20 0 @q\_a 0 0 20 0

// Retrieval info: GEN\_FILE: TYPE\_NORMAL Password\_ROM.v TRUE

// Retrieval info: GEN\_FILE: TYPE\_NORMAL Password\_ROM.inc FALSE

// Retrieval info: GEN\_FILE: TYPE\_NORMAL Password\_ROM.cmp FALSE

// Retrieval info: GEN\_FILE: TYPE\_NORMAL Password\_ROM.bsf FALSE

// Retrieval info: GEN\_FILE: TYPE\_NORMAL Password\_ROM\_inst.v FALSE

// Retrieval info: GEN\_FILE: TYPE\_NORMAL Password\_ROM\_bb.v TRUE

// Retrieval info: LIB\_FILE: altera\_mf

**hex0\_to\_hex3\_decoder.v :**

// ECE 5440

// Group: Gone Fishin'

// Module: hex0\_to\_hex3\_decoder

// This module will control what is outputted to Hex 0 - Hex 3

// of the FPGA Board. When access is not granted, this module

// will take input from access controller to display which order of the

// user ID or password is being inputted along with the actual switch value.

// When access is granted and game has not started, this module will display the User IDs

// in a round robin fashion while a seperate module will display the high score for the specific user ID

// shown by this module. When the game has started and access is granted, this module will

// display output from the game controller guiding the user of the game

module hex0\_to\_hex3\_decoder(

// sequential logic variables

clk, rst,

// switch\_value for user ID and Password

switch\_val\_in,

// access controller button for roundrobin user id's

ac\_button,

// FSM variables

state\_ac, state\_gc, access\_granted, scoreRst,

// 7-seg outputs

hex0\_out, hex1\_out, hex2\_out, hex3\_out);

// input variables

input clk, rst, access\_granted, ac\_button, scoreRst;

input [3:0] switch\_val\_in, state\_ac, state\_gc;

// output variables

output reg [6:0] hex0\_out, hex1\_out, hex2\_out, hex3\_out;

// helper variables

parameter state\_display\_nothing = 4'b0000, state\_display\_fish = 4'b0001;

parameter state\_display\_boot = 4'b0010, state\_display\_lose = 4'b0011, state\_display\_done = 4'b0100;

parameter state\_user\_1 = 4'b0101, state\_user\_2 = 4'b0110, state\_user\_3 = 4'b0111, state\_user\_4 = 4'b1000;

parameter state\_pass\_1 = 4'b1001, state\_pass\_2 = 4'b1010, state\_pass\_3 = 4'b1011, state\_pass\_4 = 4'b1100, state\_pass\_5 = 4'b1101;

// 4 instance of 7-Seg module

reg [3:0] in\_7\_seg\_1, in\_7\_seg\_2, in\_7\_seg\_3, in\_7\_seg\_4;

wire [6:0] out\_7\_seg\_1, out\_7\_seg\_2, out\_7\_seg\_3, out\_7\_seg\_4;

SevenSeg ss1(in\_7\_seg\_1, out\_7\_seg\_1);

SevenSeg ss2(in\_7\_seg\_2, out\_7\_seg\_2);

SevenSeg ss3(in\_7\_seg\_3, out\_7\_seg\_3);

SevenSeg ss4(in\_7\_seg\_4, out\_7\_seg\_4);

// accessing User ROM

reg [5:0] addr\_user; // 6-bit wide

wire [15:0] data\_user; // 16-bit wide

reg [15:0] user\_ID\_raw; // 16-bit wide

// 1 instances of ROM user retrieval module

User\_ROM romU(addr\_user, clk, data\_user);

// read 4 digits of user ID input

reg [2:0] state\_disp\_user\_id, state\_callback;

reg check\_rom;

parameter state\_disp\_user\_id\_0 = 3'b000, state\_disp\_user\_id\_1 = 3'b001;

parameter state\_disp\_user\_id\_2 = 3'b010, state\_disp\_user\_id\_3 = 3'b011;

parameter state\_disp\_user\_id\_4 = 3'b100, state\_disp\_user\_id\_5 = 3'b101;

// wait for ROM to update Data with current addr value

parameter state\_wait\_1 = 3'b110, state\_wait\_2 = 3'b111;

always @(posedge clk) begin

if(rst == 1'b0) begin

/\* if rst pressed, do not display anything and set internal state to 1st user ID \*/

hex0\_out <= 7'b1111111; // off

hex1\_out <= 7'b1111111; // off

hex2\_out <= 7'b1111111; // off

hex3\_out <= 7'b1111111; // off

state\_disp\_user\_id <= state\_disp\_user\_id\_0;

end

else begin

if(access\_granted == 1'b1 && scoreRst == 1'b1) begin

/\* game not started yet and user authenticated so round robin thru user ID's \*/

case (state\_disp\_user\_id)

/\*----Waiting on ROM to update Data---\*/

state\_wait\_1: begin

state\_disp\_user\_id <= state\_wait\_2;

end

state\_wait\_2: begin

state\_disp\_user\_id <= state\_callback;

check\_rom <= 1;

end

/\*----Locating User ID in ROM---\*/

state\_disp\_user\_id\_0: begin

if (check\_rom == 0) begin

// since we stay in state while user is viewing, only run this if check\_rom is low

state\_callback <= state\_disp\_user\_id\_0;

state\_disp\_user\_id <= state\_wait\_1;

addr\_user <= 0;

end

if(ac\_button == 1) begin

// display next user ID

state\_disp\_user\_id <= state\_disp\_user\_id\_1;

check\_rom <= 0;

end

if(check\_rom == 1) begin

// then we set the displays to the values from ROM

in\_7\_seg\_1 = data\_user[3:0];

in\_7\_seg\_2 = data\_user[7:4];

in\_7\_seg\_3 = data\_user[11:8];

in\_7\_seg\_4 = data\_user[15:12];

hex0\_out = out\_7\_seg\_1;

hex1\_out = out\_7\_seg\_2;

hex2\_out = out\_7\_seg\_3;

hex3\_out = out\_7\_seg\_4;

end

end

state\_disp\_user\_id\_1: begin

if (check\_rom == 0) begin

// since we stay in state while user is viewing, only run this if check\_rom is low

state\_callback <= state\_disp\_user\_id\_1;

state\_disp\_user\_id <= state\_wait\_1;

addr\_user <= 1;

end

if(ac\_button == 1) begin

// display next user ID

state\_disp\_user\_id <= state\_disp\_user\_id\_2;

check\_rom <= 0;

end

if(check\_rom == 1) begin

// then we set the displays to the values from ROM

in\_7\_seg\_1 = data\_user[3:0];

in\_7\_seg\_2 = data\_user[7:4];

in\_7\_seg\_3 = data\_user[11:8];

in\_7\_seg\_4 = data\_user[15:12];

hex0\_out = out\_7\_seg\_1;

hex1\_out = out\_7\_seg\_2;

hex2\_out = out\_7\_seg\_3;

hex3\_out = out\_7\_seg\_4;

end

end

state\_disp\_user\_id\_2: begin

if (check\_rom == 0) begin

// since we stay in state while user is viewing, only run this if check\_rom is low

state\_callback <= state\_disp\_user\_id\_2;

state\_disp\_user\_id <= state\_wait\_1;

addr\_user <= 2;

end

if(ac\_button == 1) begin

// display next user ID

state\_disp\_user\_id <= state\_disp\_user\_id\_3;

check\_rom <= 0;

end

if(check\_rom == 1) begin

// then we set the displays to the values from ROM

in\_7\_seg\_1 = data\_user[3:0];

in\_7\_seg\_2 = data\_user[7:4];

in\_7\_seg\_3 = data\_user[11:8];

in\_7\_seg\_4 = data\_user[15:12];

hex0\_out = out\_7\_seg\_1;

hex1\_out = out\_7\_seg\_2;

hex2\_out = out\_7\_seg\_3;

hex3\_out = out\_7\_seg\_4;

end

end

state\_disp\_user\_id\_3: begin

if (check\_rom == 0) begin

// since we stay in state while user is viewing, only run this if check\_rom is low

state\_callback <= state\_disp\_user\_id\_3;

state\_disp\_user\_id <= state\_wait\_1;

addr\_user <= 3;

end

if(ac\_button == 1) begin

// display next user ID

state\_disp\_user\_id <= state\_disp\_user\_id\_4;

check\_rom <= 0;

end

if(check\_rom == 1) begin

// then we set the displays to the values from ROM

in\_7\_seg\_1 = data\_user[3:0];

in\_7\_seg\_2 = data\_user[7:4];

in\_7\_seg\_3 = data\_user[11:8];

in\_7\_seg\_4 = data\_user[15:12];

hex0\_out = out\_7\_seg\_1;

hex1\_out = out\_7\_seg\_2;

hex2\_out = out\_7\_seg\_3;

hex3\_out = out\_7\_seg\_4;

end

end

state\_disp\_user\_id\_4: begin

if (check\_rom == 0) begin

// since we stay in state while user is viewing, only run this if check\_rom is low

state\_callback <= state\_disp\_user\_id\_4;

state\_disp\_user\_id <= state\_wait\_1;

addr\_user <= 4;

end

if(ac\_button == 1) begin

// display next user ID

state\_disp\_user\_id <= state\_disp\_user\_id\_5;

check\_rom <= 0;

end

if(check\_rom == 1) begin

// then we set the displays to the values from ROM

in\_7\_seg\_1 = data\_user[3:0];

in\_7\_seg\_2 = data\_user[7:4];

in\_7\_seg\_3 = data\_user[11:8];

in\_7\_seg\_4 = data\_user[15:12];

hex0\_out = out\_7\_seg\_1;

hex1\_out = out\_7\_seg\_2;

hex2\_out = out\_7\_seg\_3;

hex3\_out = out\_7\_seg\_4;

end

end

state\_disp\_user\_id\_5: begin

if (check\_rom == 0) begin

// since we stay in state while user is viewing, only run this if check\_rom is low

state\_callback <= state\_disp\_user\_id\_5;

state\_disp\_user\_id <= state\_wait\_1;

addr\_user <= 5;

end

if(ac\_button == 1) begin

// display next user ID

state\_disp\_user\_id <= state\_disp\_user\_id\_0;

check\_rom <= 0;

end

if(check\_rom == 1) begin

// then we set the displays to the values from ROM

in\_7\_seg\_1 = data\_user[3:0];

in\_7\_seg\_2 = data\_user[7:4];

in\_7\_seg\_3 = data\_user[11:8];

in\_7\_seg\_4 = data\_user[15:12];

hex0\_out = out\_7\_seg\_1;

hex1\_out = out\_7\_seg\_2;

hex2\_out = out\_7\_seg\_3;

hex3\_out = out\_7\_seg\_4;

end

end

endcase

end

else if(access\_granted == 1'b1) begin

state\_disp\_user\_id <= state\_disp\_user\_id\_0;

check\_rom <= 0;

case (state\_gc)

state\_display\_nothing: begin

hex0\_out <= 7'b1111111; // off

hex1\_out <= 7'b1111111; // off

hex2\_out <= 7'b1111111; // off

hex3\_out <= 7'b1111111; // off

end

state\_display\_fish: begin

hex0\_out <= 7'b0001001; // "H"

hex1\_out <= 7'b0010010; // "S"

hex2\_out <= 7'b1111001; // "I"

hex3\_out <= 7'b0001110; // "F"

end

state\_display\_boot: begin

hex0\_out <= 7'b0000111; // "t"

hex1\_out <= 7'b1000000; // "O"

hex2\_out <= 7'b1000000; // "O"

hex3\_out <= 7'b0000011; // "b"

end

state\_display\_lose: begin

hex0\_out <= 7'b0000110; // "E"

hex1\_out <= 7'b0010010; // "S"

hex2\_out <= 7'b1000000; // "O"

hex3\_out <= 7'b1000111; // "L"

end

state\_display\_done: begin

hex0\_out <= 7'b0000110; // "E"

hex1\_out <= 7'b0101011; // "n"

hex2\_out <= 7'b1000000; // "O"

hex3\_out <= 7'b0100001; // "d"

end

endcase

end

else begin

/\* Access is not granted \*/

state\_disp\_user\_id <= state\_disp\_user\_id\_0;

check\_rom <= 0;

// the switch value will always be displayed on hex0 when access is not granted

in\_7\_seg\_2 = switch\_val\_in;

hex0\_out = out\_7\_seg\_2;

// hex1 will be blank when access is not granted

hex1\_out <= 7'b1111111; // off

if(state\_ac < 4'b1001) begin

// if access controller is loading user values

hex3\_out <= 7'b1000001; // "U"

in\_7\_seg\_1 = state\_ac - 4'b0100;

hex2\_out = out\_7\_seg\_1;

end

else begin

// if access controller is loading password values

hex3\_out <= 7'b0001100; // "P"

in\_7\_seg\_1 = state\_ac - 4'b1000;

hex2\_out = out\_7\_seg\_1;

end

end

end

end

endmodule

**hex4\_to\_hex5\_decoder.v:**

// ECE 5440

// Group: Gone Fishin'

// Module: hex4\_to\_hex5\_decoder

// This module will control what is outputted to Hex 4 - Hex 5

// of the FPGA Board. When access is not granted, this module

// will output "--".

// When access is granted and game has not started, this module

// will display the high score for each user in a round robin fashion while

// another module will display the associated user ID. When

// the game has started and access is granted, this module will

// simply display the current game score.

module hex4\_to\_hex5\_decoder(

// sequential logic variables

clk, rst,

// access button

ac\_button,

// FSM variables

userID, currentGameScore, access\_granted, scoreRst,

// Inputs from RAM Controller

scoreUserAddr0, scoreUserAddr1, scoreUserAddr2, scoreUserAddr3, scoreUserAddr4, scoreUserAddr5,

// 7-seg outputs

hex4\_out, hex5\_out);

// input variables

input clk, rst, access\_granted, scoreRst, ac\_button;

input [2:0] userID;

input [7:0] currentGameScore, scoreUserAddr0, scoreUserAddr1, scoreUserAddr2, scoreUserAddr3, scoreUserAddr4, scoreUserAddr5;

// output variables

output reg [6:0] hex4\_out, hex5\_out;

// 2 instance of 7-Seg module

reg [3:0] in\_7\_seg\_1, in\_7\_seg\_2;

wire [6:0] out\_7\_seg\_1, out\_7\_seg\_2;

SevenSeg ss1(in\_7\_seg\_1, out\_7\_seg\_1);

SevenSeg ss2(in\_7\_seg\_2, out\_7\_seg\_2);

// read 4 digits of user ID input

reg [2:0] state\_disp\_user\_ram\_score;

parameter state\_disp\_user\_ram\_score\_0 = 3'b000, state\_disp\_user\_ram\_score\_1 = 3'b001;

parameter state\_disp\_user\_ram\_score\_2 = 3'b010, state\_disp\_user\_ram\_score\_3 = 3'b011;

parameter state\_disp\_user\_ram\_score\_4 = 3'b100, state\_disp\_user\_ram\_score\_5 = 3'b101;

always @(posedge clk) begin

if(rst == 1'b0) begin

/\* if rst pressed, do not display anything and set internal state to 1st user ID \*/

hex4\_out <= 7'b1111111; // off

hex5\_out <= 7'b1111111; // off

state\_disp\_user\_ram\_score <= state\_disp\_user\_ram\_score\_0;

end

else begin

if(access\_granted == 1'b1 && scoreRst == 1'b1) begin

/\* game not started yet and user authenticated so round robin thru user ram score's \*/

case (state\_disp\_user\_ram\_score)

state\_disp\_user\_ram\_score\_0: begin

if(ac\_button == 1) begin

state\_disp\_user\_ram\_score <= state\_disp\_user\_ram\_score\_1;

end

else begin

in\_7\_seg\_1 = scoreUserAddr0[3:0];

in\_7\_seg\_2 = scoreUserAddr0[7:4];

hex4\_out = out\_7\_seg\_1;

hex5\_out = out\_7\_seg\_2;

end

end

state\_disp\_user\_ram\_score\_1: begin

if(ac\_button == 1) begin

state\_disp\_user\_ram\_score <= state\_disp\_user\_ram\_score\_2;

end

else begin

in\_7\_seg\_1 = scoreUserAddr1[3:0];

in\_7\_seg\_2 = scoreUserAddr1[7:4];

hex4\_out = out\_7\_seg\_1;

hex5\_out = out\_7\_seg\_2;

end

end

state\_disp\_user\_ram\_score\_2: begin

if(ac\_button == 1) begin

state\_disp\_user\_ram\_score <= state\_disp\_user\_ram\_score\_3;

end

else begin

in\_7\_seg\_1 = scoreUserAddr2[3:0];

in\_7\_seg\_2 = scoreUserAddr2[7:4];

hex4\_out = out\_7\_seg\_1;

hex5\_out = out\_7\_seg\_2;

end

end

state\_disp\_user\_ram\_score\_3: begin

if(ac\_button == 1) begin

state\_disp\_user\_ram\_score <= state\_disp\_user\_ram\_score\_4;

end

else begin

in\_7\_seg\_1 = scoreUserAddr3[3:0];

in\_7\_seg\_2 = scoreUserAddr3[7:4];

hex4\_out = out\_7\_seg\_1;

hex5\_out = out\_7\_seg\_2;

end

end

state\_disp\_user\_ram\_score\_4: begin

if(ac\_button == 1) begin

state\_disp\_user\_ram\_score <= state\_disp\_user\_ram\_score\_5;

end

else begin

in\_7\_seg\_1 = scoreUserAddr4[3:0];

in\_7\_seg\_2 = scoreUserAddr4[7:4];

hex4\_out = out\_7\_seg\_1;

hex5\_out = out\_7\_seg\_2;

end

end

state\_disp\_user\_ram\_score\_5: begin

if(ac\_button == 1) begin

state\_disp\_user\_ram\_score <= state\_disp\_user\_ram\_score\_0;

end

else begin

in\_7\_seg\_1 = scoreUserAddr5[3:0];

in\_7\_seg\_2 = scoreUserAddr5[7:4];

hex4\_out = out\_7\_seg\_1;

hex5\_out = out\_7\_seg\_2;

end

end

endcase

end

else if(access\_granted == 1'b1) begin

/\* Game Started \*/

// set display state to first user score

state\_disp\_user\_ram\_score <= state\_disp\_user\_ram\_score\_0;

in\_7\_seg\_1 = currentGameScore[3:0];

in\_7\_seg\_2 = currentGameScore[7:4];

hex4\_out = out\_7\_seg\_1;

hex5\_out = out\_7\_seg\_2;

end

else begin

/\* Access is not granted \*/

hex4\_out <= 7'b0111111; // dash

hex5\_out <= 7'b0111111; // dash

state\_disp\_user\_ram\_score <= state\_disp\_user\_ram\_score\_0;

end

end

end

endmodule

**gameController.v:**

// ECE 5440

// Group: Gone Fishin'

// Module: Game controller

//

// This module is the primary controller for the Gone Fishin' game. It

// implements the state machine for gameplay and instantiates several related

// modules, including the various timers used and the random number generator.

module gameController (

// game control inputs

gameControl, startingDiff, reelButton, castButton, currentScore,

// score control outputs

scoreUp, scoreDown, scoreRst,

// display outputs

lightBar, gameInfo, fishTimeDisp, blinkBit,

// system clock and reset

CLK, RST);

input CLK, RST;

//-----Control inputs-----

// Bit indicating if the game should be operational or suspended

input gameControl;

// this is the starting difficulty (0-7, higher more difficult)

input [2:0] startingDiff;

// this shaped button input starts and ends the game

input castButton;

// this button increases the LED bar by one during gameplay

input reelButton;

// this is the current score (from the score keeper) represented as

// 2 4-bit decimal digits

input [7:0] currentScore;

//-----Display & score control outputs-----

// These are controlled by the state machine unless otherwise indicated

//

// this controls the LED bar that indicates the fishing status

// controlled by LED blinker module

output [25:0] lightBar;

// this selects various things to be displayed on the info display

// passed through a decoder

output [3:0] gameInfo;

reg [3:0] gameInfo;

// these outputs increment, decrement, and reset the score, respectively

output scoreUp, scoreDown, scoreRst;

reg scoreUp, scoreDown, scoreRst;

// this outputs the remaining time until the fish is caught as two

// 4-bit hex digits, but representing decimal numbers

// tied to a timer module output below

output [7:0] fishTimeDisp;

// this bit is used to blink the info display

// controlled by LED blinker module

output blinkBit;

//-----RNG-----

// Random numbers will be generated from a free-spinning 16-bit LFSR

wire [15:0] currentRand; // current value of the LFSR

// this LFSR will shift every clock cycle, always

LFSR16 randomSource (1'b0, 1'b1, currentRand, CLK, RST);

//-----Difficulty selector-----

// The difficulty selector takes a difficulty setting and the current

// score as inputs and produces timer starting values and win/lose bounds

// as outputs. As the difficulty/score increase, reel time gets

// shorter and fish time gets longer. Since the fish timer is also

// used for the waiting periods, but we don't want to wait longer on

// higher difficulties, the useWaitTime bit tells the selector to

// output a shorter load time for the fish timer to use during waiting periods.

reg holdDiff; // tells selector to keep the current difficulty until the end of the game

reg useWaitTime; // tells selector to use a shorter time for waiting-for-fish periods

defparam diffSel.WAIT\_TIME = {4'h0, 4'h4, 4'h0, 4'h0, 4'h0}; // 04.000 sec

wire [11:0] reelTimerLoad; // reel timer load value

wire [19:0] fishTimerLoad; // fish timer load value

// win/lose bounds (inputs to LEDBlinker below)

wire [25:0] leftLoseBound, rightLoseBound, winZone;

difficultySelector diffSel (startingDiff, currentScore, holdDiff, useWaitTime,

reelTimerLoad, fishTimerLoad, leftLoseBound, rightLoseBound, winZone,

CLK, RST);

// TODO: add interval randomizers

//-----Gameplay timers-----

// There are two timers involved in gameplay:

// fishTimer - time until an object is hooked or a fish is caught

// reelTimer - time until LED decrements or boot is off hook

reg runFishTimer, runReelTimer; // bits telling the timer to run

reg rstFishTimer, rstReelTimer; // bits telling the timer to reset and load new values

wire fishTimerDone, reelTimerDone; // bits indicating the timer has timed out

wire [19:0] fullFishTimeDisp; // full time has three sub-second digits not used

assign fishTimeDisp = fullFishTimeDisp[19:12];

parameter LFSRTarget = 16'h9CED; // target for 50 MHz

defparam fishTimer.LFSRTarget = LFSRTarget;

defparam reelTimer.LFSRTarget = LFSRTarget;

defparam fishTimer.Digits = 5; //xx.xxx

countdownTimer fishTimer (

.StartValue(fishTimerLoad),

.Restart(rstFishTimer),

.Run(runFishTimer),

.CurrentValue(fullFishTimeDisp), // tied to a module output

.TimerDone(fishTimerDone),

.CLK(CLK),

.RST(RST));

defparam reelTimer.Digits = 3; //.xxx

countdownTimer reelTimer (

.StartValue(reelTimerLoad),

.Restart(rstReelTimer),

.Run(runReelTimer),

.CurrentValue(), // intentionally not connected

.TimerDone(reelTimerDone),

.CLK(CLK),

.RST(RST));

//-----LED blinker----

// This module controls the LED bar output, selecting some LEDs to

// turn on and some to blink (using an internal timer).

// It also outputs a blink bit to control blinking of the info

// display. LED bar is represented as a 26-bit binary number where

// LEDG7 is the LSB and LEDR0 is the MSB.

//

// Three LEDs will blink: the right and left "lose bounds", and the current

// LED (controlled by the reel button/timer). If the current LED

// hits either boundary, you lose the game.

//

// The LEDs between the "win bounds," inclusive, will be lit (solid),

// except the current LED if it is in that range. The current LED has

// to stay in the win bounds for the fish timer to count down; when

// the timer reaches zero, you catch a fish and gain a point.

//

// For reference, the green-red boundary that these bounds should be centered

// around is bit 16 (LEDR16) and 17 (LEDG0). Bounds are set by the difficulty

// selector above.

// set up a shift/load register to control the current LED

// resetLED also acts as an inverted enable/disable for the LED

// blinker output since those happen to be synonymous

reg LEDGoRight, LEDGoLeft, resetLED;

localparam startingLED = 26'b1 << 9;

wire [25:0] currentLED;

defparam LEDMultiReg.Bits = 26;

multiReg LEDMultiReg (startingLED, resetLED, LEDGoLeft, LEDGoRight, currentLED, CLK, RST);

// these are the LEDs that will blink

wire [25:0] blinkThese = leftLoseBound | rightLoseBound | currentLED;

// instantiate blinker module

defparam blinky.LFSRTarget = LFSRTarget;

LEDBlinker blinky (

.lightThese(winZone),

.blinkThese(blinkThese),

.blinkEnab(!resetLED),

.lightBar(lightBar),

.blinkBit(blinkBit),

.CLK(CLK),

.RST(RST));

//-----Game state machine-----

// Since this thing is so big, it's divided into three procedures:

// - sequential logic to advance state

// - combinational logic to set outputs from current state

// - combinational logic to set next state from inputs and current state

// in that order

// 13 states

reg [3:0] State, nextState;

localparam waitingForStart = 0, waitingForFish = 1, fishOrBoot = 2, bootOnHook = 3,

catchBoot = 4, catchingFish = 5, stalemateWithFish = 6, leftOneLED = 7,

rightOneLED = 8, catchFish = 9, gameOver = 10, gameDone = 11;

// possible info outputs, named for clarity

localparam NONE = 0, FISH = 1, BOOT = 2, LOSE = 3, DONE = 4;

// SeqLogic to advance or reset state

always @(posedge CLK) begin

if (!RST || !gameControl ) begin // negative-logic reset ORed with game control bit

State <= waitingForStart;

end

else begin

State <= nextState;

end

end

// CombLogic to set outputs based on current state

always @(State) begin

//-----default output values-----

// reset fish timer

rstFishTimer = 1;

runFishTimer = 0;

// reset reel timer

rstReelTimer = 1;

runReelTimer = 0;

// reset current LED to default (and do not show)

resetLED = 1;

LEDGoLeft = 0;

LEDGoRight = 0;

// do not show game info

gameInfo = NONE;

// score stays constant

scoreUp = 0;

scoreDown = 0;

scoreRst = 0;

// difficulty is latched (not changing)

holdDiff = 1;

// wait time is used by default for fish timer

useWaitTime = 1;

case (State)

waitingForStart: begin

// reset score in starting state

scoreRst = 1;

// load in new difficulty

holdDiff = 0;

end

waitingForFish: begin

// run fish timer

runFishTimer = 1;

rstFishTimer = 0;

// use fishing time, not wait time

//useWaitTime = 0;

end

fishOrBoot: begin

// use fishing time, not wait time

useWaitTime = 0;

end

bootOnHook: begin

// run reel timer

runReelTimer = 1;

rstReelTimer = 0;

// show BOOT on game info

gameInfo = BOOT;

end

catchBoot: begin

// score goes down because they caught a boot

scoreDown = 1;

end

catchingFish: begin

// pull LED reset low

resetLED = 0;

// run fish timer

runFishTimer = 1;

rstFishTimer = 0;

// run reel timer

runReelTimer = 1;

rstReelTimer = 0;

// show FISH on game info

gameInfo = FISH;

end

stalemateWithFish: begin

// pull LED reset low

resetLED = 0;

// pause but do not reset fish timer

runFishTimer = 0;

rstFishTimer = 0;

// run reel timer

runReelTimer = 1;

rstReelTimer = 0;

// show FISH on game info

gameInfo = FISH;

end

// NB: I am aware that the fish timer runs in the two states

// below even if you're coming from the paused state, which

// is technically incorrect. However, I think if you

// are able to win the game using this bug, you

// deserve to win, so I'm not changing it.

leftOneLED: begin

// pull LED reset low, shift LED left

resetLED = 0;

LEDGoLeft = 1;

// run fish timer

runFishTimer = 1;

rstFishTimer = 0;

// reset reel timer

runReelTimer = 0;

rstReelTimer = 1;

// show FISH on game info

gameInfo = FISH;

end

rightOneLED: begin

// pull LED reset low, shift LED right

resetLED = 0;

LEDGoRight = 1;

// run fish timer

runFishTimer = 1;

rstFishTimer = 0;

// run reel timer

runReelTimer = 1;

rstReelTimer = 0;

// show FISH on game info

gameInfo = FISH;

end

catchFish: begin

// score goes up because you caught a fish

scoreUp = 1;

end

gameOver: begin

// show LOSE on game info

gameInfo = LOSE;

end

gameDone: begin

// show DONE on game info

gameInfo = DONE;

end

endcase

end

// CombLogic to define next state from inputs and current state

always @\* begin

case (State)

waitingForStart: begin

// start game if cast button is pressed

if (castButton) begin

nextState = waitingForFish;

end

else begin

nextState = waitingForStart;

end

end

waitingForFish: begin

// end game if cast button is pressed

if (castButton) begin

nextState = gameDone;

end

// if fish timer is done, advance to fish/boot selection

else if (fishTimerDone) begin

nextState = fishOrBoot;

end

else begin

nextState = waitingForFish;

end

end

fishOrBoot: begin

// end game if cast button is pressed

if (castButton) begin

nextState = gameDone;

end

// select fish or boot based on LSB of random number

else if (currentRand[0]) begin

nextState = bootOnHook;

end

else begin // !currentRand[0]

nextState = catchingFish;

end

end

bootOnHook: begin

// end game if cast button is pressed

if (castButton) begin

nextState = gameDone;

end

// catch boot if reel button is pressed

else if (reelButton) begin

nextState = catchBoot;

end

// if reel timer is done, go back to waiting

else if (reelTimerDone) begin

nextState = waitingForFish;

end

else begin

nextState = bootOnHook;

end

end

catchBoot: begin

// end game if cast button is pressed

if (castButton) begin

nextState = gameDone;

end

else begin

nextState = waitingForFish;

end

end

catchingFish: begin

// end game if cast button is pressed

if (castButton) begin

nextState = gameDone;

end

// if fish timer is done, you catch fish

else if (fishTimerDone) begin

nextState = catchFish;

end

// in the unlikely event reel button and timer

// signal at the same time, nothing happens

else if (reelTimerDone && reelButton) begin

nextState = catchingFish;

end

// if reel button is pressed, LED goes right

else if (reelButton) begin

nextState = rightOneLED;

end

// if reel timer is done, LED goes left

else if (reelTimerDone) begin

nextState = leftOneLED;

end

else begin

nextState = catchingFish;

end

end

stalemateWithFish: begin

// end game if cast button is pressed

if (castButton) begin

nextState = gameDone;

end

// in the unlikely event reel button and timer

// signal at the same time, nothing happens

else if (reelTimerDone && reelButton) begin

nextState = stalemateWithFish;

end

// if reel button is pressed, LED goes right

else if (reelButton) begin

nextState = rightOneLED;

end

// if reel timer is done, LED goes left

else if (reelTimerDone) begin

nextState = leftOneLED;

end

else begin

nextState = stalemateWithFish;

end

end

leftOneLED: begin

// end game if cast button is pressed

if (castButton) begin

nextState = gameDone;

end

// if reel button is pressed, LED goes right

else if (reelButton) begin

nextState = rightOneLED;

end

// if LED is going to hit the left lose bound

// (after shift), player loses

else if (currentLED == (leftLoseBound >> 1)) begin

nextState = gameOver;

end

// if LED is within win bounds, you are

// catching fish

else if (currentLED & winZone) begin

nextState = catchingFish;

end

else begin

nextState = stalemateWithFish;

end

end

rightOneLED: begin

// end game if cast button is pressed

if (castButton) begin

nextState = gameDone;

end

// if reel timer is done, LED goes left

else if (reelTimerDone) begin

nextState = leftOneLED;

end

// if LED is going to hit the right lose bound

// (after shift), player loses

else if (currentLED == (rightLoseBound << 1)) begin

nextState = gameOver;

end

// if LED is within win bounds, you are

// catching fish

else if (currentLED & winZone) begin

nextState = catchingFish;

end

else begin

nextState = stalemateWithFish;

end

end

catchFish: begin

// end game if cast button is pressed

if (castButton) begin

nextState = gameDone;

end

else begin

nextState = waitingForFish;

end

end

gameOver: begin

// return to start if cast button is pressed

if (castButton) begin

nextState = waitingForStart;

end

else begin

nextState = gameOver;

end

end

gameDone: begin

// return to start if cast button is pressed

if (castButton) begin

nextState = waitingForStart;

end

else begin

nextState = gameDone;

end

end

// return to start if in an illegal state

default: begin

nextState = waitingForStart;

end

endcase

end

endmodule

**difficultySelector.v:**

// ECE 5440

// Group: Gone Fishin'

// Module: Difficulty selector

//

// This module takes an initial difficulty selection and a current score and

// uses these to select times for the reel & catch timers, which

// decrease/increase respectively with initial difficulty and score, and win/lose

// bounds for the LED bar, which get tighter as difficulty increases (not affected

// by score). Mostly combinational, except for a latch. Not really a separate function from the game

// controller, but it's broken out because that module is already pretty long

module difficultySelector (currentDiff, currentScore, holdDiff, useWaitTime, // inputs

reelTime, fishTime, leftLoseBound, rightLoseBound, winZone, // outputs

CLK, RST);

// system clock and reset

// only used for latching difficulty

input CLK, RST;

// 8 possible starting difficulties

input [2:0] currentDiff; // actual current difficulty input

reg [2:0] startingDiff; // stored value for use during gameplay

// current score is a 2-digit decimal number

// represented as 2 4-bit hex digits

input [7:0] currentScore;

wire [3:0] scoreTens = currentScore[7:4];

wire [3:0] scoreOnes = currentScore[3:0];

// holdDiff bit controls latching of currentDiff->startingDiff

// (1=opaque)

input holdDiff;

// useWaitTime bit indicates that the fishTime should be set to

// a lower, fixed time (for wait periods) rather than longer varying

// time for catching fish

input useWaitTime;

// WAIT\_TIME is the fixed time mentioned above

parameter WAIT\_TIME = {4'h0, 4'h4, 4'h0, 4'h0, 4'h0}; // 4 sec

// reel time is time it takes one LED to decrement,

// or a boot to go away

// up to .999 sec, represented as 3 4-bit hex digits .xxx

output [11:0] reelTime; // assigned at end of module

// fish time is time it takes a fish to be hooked/caught

// up to 99.999 secs (theoretically)

// represented as 5 4-bit hex digits xx.xxx

output [19:0] fishTime; // assigned at end of module

// left and right lose bounds are the bounds you must keep the current

// LED within to not lose (closer on higher difficulty)

// defined in terms of the 26-LED LED bar on the board

output [25:0] leftLoseBound;

reg [25:0] leftLoseBound;

output [25:0] rightLoseBound;

reg [25:0] rightLoseBound;

// win zone is the zone current LED must be in for the timer to count down

output [25:0] winZone;

reg [25:0] winZone;

// set up difficulty latching

always @(posedge CLK) begin

if (!RST) begin // negative-logic reset

startingDiff <= 0;

end

else begin

// latch is transparent if holdDiff is low

if (holdDiff) begin

startingDiff <= startingDiff;

end

else begin

startingDiff <= currentDiff;

end

end

end

// first, preliminary values are set from starting diff

reg [11:0] reelTimePrelim;

reg [19:0] fishTimePrelim;

always @(startingDiff) begin

// tens, hundths, and thouths digits of prelim fish time will not be nonzero

fishTimePrelim[19:16] = 4'h0;

fishTimePrelim[7:0] = {4'h0, 4'h0};

// millisecond digit of prelim reel time will not be nonzero

reelTimePrelim[3:0] = {4'h0};

case (startingDiff)

'h0: begin

// .99 sec

reelTimePrelim[11:4] = {4'h9, 4'h9};

// 3.0 sec

fishTimePrelim[15:8] = {4'h3, 4'h0};

// LEDG7

rightLoseBound = 26'b1 << 0;

// LEDR10

leftLoseBound = 26'b1 << 16;

// LEDR14 - LEDG3

winZone = 26'b00000000000000111111110000;

end

'h1: begin

// .95 sec

reelTimePrelim[11:4] = {4'h9, 4'h5};

// 3.5 sec

fishTimePrelim[15:8] = {4'h3, 4'h5};

// LEDG6

rightLoseBound = 26'b1 << 1;

// LEDR11

leftLoseBound = 26'b1 << 15;

// LEDR14 - LEDG3

winZone = 26'b00000000000000111111110000;

end

'h2: begin

// .89 sec

reelTimePrelim[11:4] = {4'h8, 4'h9};

// 4.0 sec

fishTimePrelim[15:8] = {4'h4, 4'h0};

// LEDG5

rightLoseBound = 26'b1 << 2;

// LEDR12

leftLoseBound = 26'b1 << 14;

// LEDR15 - LEDG2

winZone = 26'b00000000000000011111100000;

end

'h3: begin

// .85 sec

reelTimePrelim[11:4] = {4'h8, 4'h5};

// 4.5 sec

fishTimePrelim[15:8] = {4'h4, 4'h5};

// LEDG4

rightLoseBound = 26'b1 << 3;

// LEDR13

leftLoseBound = 26'b1 << 13;

// LEDR15 - LEDG2

winZone = 26'b00000000000000011111100000;

end

'h4: begin

// .79 sec

reelTimePrelim[11:4] = {4'h7, 4'h9};

// 5.0 sec

fishTimePrelim[15:8] = {4'h5, 4'h0};

// LEDG3

rightLoseBound = 26'b1 << 4;

// LEDR14

leftLoseBound = 26'b1 << 12;

// LEDR16 - LEDG1

winZone = 26'b00000000000000001111000000;

end

'h5: begin

// .75 sec

reelTimePrelim[11:4] = {4'h7, 4'h5};

// 5.5 sec

fishTimePrelim[15:8] = {4'h5, 4'h5};

// note that the bounds here are the same as the

// previous state because otherwise we run out

// of space to decrease

// LEDG3

rightLoseBound = 26'b1 << 4;

// LEDR14

leftLoseBound = 26'b1 << 12;

// LEDR16 - LEDG1

winZone = 26'b00000000000000001111000000;

end

'h6: begin

// .69 sec

reelTimePrelim[11:4] = {4'h6, 4'h9};

// 5.9 sec

// note that this stops increasing with difficulty

// because 5 is the highest valid ones value

fishTimePrelim[15:8] = {4'h5, 4'h9};

// LEDG2

rightLoseBound = 26'b1 << 5;

// LEDR15

leftLoseBound = 26'b1 << 11;

// LEDR17 - LEDG0

winZone = 26'b00000000000000000110000000;

end

'h7: begin

// .65 sec

reelTimePrelim[11:4] = {4'h6, 4'h5};

// 5.9 sec

fishTimePrelim[15:8] = {4'h5, 4'h9};

// LEDG1

rightLoseBound = 26'b1 << 6;

// LEDR16

leftLoseBound = 26'b1 << 10;

// LEDR17 - LEDG0

winZone = 26'b00000000000000000110000000;

end

endcase

end

// next, score selects an amount to add/subtract to/from the prelim values

reg [3:0] reelHundthsSub, reelTenthsSub;

reg [3:0] fishOnesAdd;

always @(scoreOnes, scoreTens) begin

// select values for fish timer ones/reel timer hundths

// based on highest nonzero bit of score ones

if (scoreOnes[3]) begin

reelHundthsSub = 'h4;

fishOnesAdd = 'h4;

end

else if (scoreOnes[2]) begin

reelHundthsSub = 'h3;

fishOnesAdd = 'h3;

end

else if (scoreOnes[1]) begin

reelHundthsSub = 'h2;

fishOnesAdd = 'h2;

end

else if (scoreOnes[0]) begin

reelHundthsSub = 'h1;

fishOnesAdd = 'h1;

end

else begin

reelHundthsSub = 'h0;

fishOnesAdd = 'h0;

end

// select value for reel timer tenths based on highest

// nonzero bit of score tens

if (scoreTens[3]) begin

reelTenthsSub = 'h4;

end

else if (scoreTens[2]) begin

reelTenthsSub = 'h3;

end

else if (scoreTens[1]) begin

reelTenthsSub = 'h2;

end

else if (scoreTens[0]) begin

reelTenthsSub = 'h1;

end

else begin

reelTenthsSub = 'h0;

end

// add value for fish timer tens is simply based on the tens

// digit of the score

end

// finally, implement the actual adding and subtracting and produce output

wire [7:0] reelTimeDiffPart = reelTimePrelim[11:4] - {reelTenthsSub, reelHundthsSub};

wire [7:0] fishTimeSumPart = fishTimePrelim[19:12] + {scoreTens, fishOnesAdd};

assign reelTime = {reelTimeDiffPart, reelTimePrelim[3:0]};

assign fishTime = (useWaitTime) ? WAIT\_TIME : {fishTimeSumPart, fishTimePrelim[11:0]};

endmodule

**RangeLFSR16.v:**

// ECE 5440

// Group: Gone Fishin'

// Module: RangLFSR16.v

//This module generates a random number within the range specified by the user.

module RangeLFSR16(Restart,Run,offset,limit,out,CLK,RST);

input Restart,Run,CLK,RST;

input [15:0] offset,limit;

output [15:0] out;

reg [15:0] out;

wire [15:0] Value;

LFSR16 LFSR16\_1(Restart,Run,Value,CLK,RST);

always @ (CLK)

begin

out=offset+(Value%(limit+1-offset));

end

endmodule

**LEDBlinker.v:**

// ECE 5440

// Group: Gone Fishin'

// Module: LED bar blinker

//

// This module controls the LED bar on the DE2-112 board (i.e. every LED except

// the one between the 7-segs) and selects LEDs to a) turn on and b) blink;

// the blinking is controlled by an internal timer. A blink bit is

// also made available as an output to control other blinking displays.

module LEDBlinker (lightThese, blinkThese, blinkEnab, lightBar, blinkBit, CLK, RST);

// these LEDs will be lit

input [25:0] lightThese;

// these LEDs will blink

// they will blink even if not selected above

input [25:0] blinkThese;

// LEDs will only be driven if this is high

input blinkEnab;

// light bar control output

// gated with enable bit

output [25:0] lightBar;

reg [25:0] lightBarUngated;

assign lightBar = blinkEnab ? lightBarUngated : 26'b0;

// system clock and reset

input CLK, RST;

// blink output bit

// tied to the (gated) state machine bit

output blinkBit;

reg blinkOn;

assign blinkBit = blinkOn || !blinkEnab;

// this timer controls the blinking rate

// runs continuously

wire timerDone; // timeout signal from timer

parameter blinkTime = {4'h2, 4'h0, 4'h0}; // .200 sec

parameter LFSRTarget = 16'h9CED; // target for 50 MHz clock

defparam blinkTimer.Digits = 3; // up to .999 sec

defparam blinkTimer.LFSRTarget = LFSRTarget;

countdownTimer blinkTimer (

.StartValue(blinkTime),

.Restart(1'b0), // never restart (except for a RST)

.Run(1'b1), // always run

.CurrentValue(), // intentionally not connected

.TimerDone(timerDone),

.CLK(CLK),

.RST(RST));

// this is basically just a two state state machine

// one state the blinking LEDs are off, the other they are on

// the non blinking are on all the time

always @(posedge CLK) begin

if (!RST) begin // negative-logic reset

blinkOn <= 0;

end

else begin

case (blinkOn)

// blinking LEDs off

1'b0: begin

lightBarUngated <= lightThese & ~blinkThese;

if (timerDone) begin

blinkOn <= 1'b1;

end

else begin

blinkOn <= 1'b0;

end

end

// blinking LEDs on

1'b1: begin

lightBarUngated <= lightThese | blinkThese;

if (timerDone) begin

blinkOn <= 1'b0;

end

else begin

blinkOn <= 1'b1;

end

end

endcase

end

end

endmodule

**LFSR16.v:**

// ECE 5440

// Group: Gone Fishin'

// Module: 16-bit LFSR

//

// This module implements a maximum-period (2^16 -1) 16-bit LFSR for use as the core

// of a timer module.

module LFSR16 (Restart, Run, Value, CLK, RST);

// this bit resets the LFSR (same as system RST)

// has priority over Run input

input Restart;

// the LFSR only counts if this input is high

input Run;

// current value of the LFSR state

output [15:0] Value;

reg [15:0] state;

assign Value = state;

// these wires implement the actual shifting,

// the sequential logic simply latches shift3->state every clock

// when Run is high

wire [15:0] shift1 = state ^ (state << 7);

wire [15:0] shift2 = shift1 ^ (shift1 >> 9);

wire [15:0] shift3 = shift2 ^ (shift2 << 8);

// system clock and reset signals

input CLK, RST;

always @(posedge CLK) begin

if (!RST || Restart) begin // negative-logic reset

state <= 16'hFFFF; // any nonzero seed works, this one is memorable

end

else begin

if (Run) begin

// see wire assignments above

state <= shift3;

end

else begin

state <= state;

end

end

end

endmodule

**countdownTimer.v:**

// ECE 5440

// Group: Gone Fishin'

// Module: Countdown timer

//

// This module implements a decimal-valued timer with a resolution of 1 ms and

// parameterized number of digits. Each digit is represented at both the input

// and the output as a 4-bit hex number. The core 1 ms counter is implemented

// using a 16-bit LFSR.

//

// Note: LSD = least sig. digit, MSD = most sig. digit

module countdownTimer (StartValue, Restart, Run, CurrentValue, TimerDone, CLK, RST);

// number of decimal digits (starting from .001) to instantiate

// minimum 2 for proper functioning

parameter Digits = 5; // 5 digits is decimal xx.xxx

// target value for LFSR, i.e. the LFSR value corresponding to the

// number of clock cycles in 1 ms for your system, minus two

// the LFSR is a 16-bit Xorshift with triplet [7, 9, 8] and seed 'hFFFF

parameter LFSRTarget = 16'h9CED; // counter value 49999998 (50 MHz clock)

// value to count down from

// each 4-bit chunk represents one decimal value 0-9

// MSD on the left, LSD (1 ms) on the right

input [Digits\*4-1:0] StartValue;

// this bit holds the timer in the waiting state while it is high

// takes priority over the run bit

input Restart;

// the timer will run when this bit is high

// if it is pulled low without a restart, the timer will pause

input Run;

// current value of the timer

// format is same as StartValue

output [Digits\*4-1:0] CurrentValue;

// we need to see the current value within the module

wire [Digits\*4-1:0] currentTime;

assign CurrentValue = currentTime;

// bit indicating timer is done

// tied to the highest overflow bit

output TimerDone;

reg TimerDone;

// internal nets controlling biDirBCD modules

reg countBit, setBit;

// internal nets controlling LFSR module

reg runLFSR, restartLFSR;

// current value of the LFSR

wire [15:0] shiftRegValue;

// system clock and reset signals

input CLK, RST;

// internal nets to link BCDs

// bottom carry in is tied to countBit

wire [Digits-1:0] counterCarry;

// generate block for BCD chain

genvar d;

generate

for (d = 0; d < Digits; d = d + 1) begin: counters

wire count;

if (d == 0) begin

// external control bit for LSD

assign count = countBit;

end

else begin

// internal linking net for all others

assign count = counterCarry[d-1];

end

// all counters counting down

defparam Digit.Direction = 0;

biDirBCD Digit (

.Count(count),

.SetValue(StartValue[(d+1)\*4-1:d\*4]),

.Set(setBit),

.OutValue(currentTime[(d+1)\*4-1:d\*4]),

.CarryOut(counterCarry[d]),

.CLK(CLK),

.RST(RST));

end

endgenerate

// instantiate LFSR module to do 1ms count

LFSR16 shiftReg (restartLFSR, runLFSR, shiftRegValue, CLK, RST);

// states for state machine

localparam waiting = 0, running = 1, counting = 2, paused = 3, timeout = 4;

reg [2:0] State;

reg [2:0] nextState;

// CombLogic to set outputs and next state from current one

always @(State, Run, Restart, shiftRegValue, currentTime) begin

case (State)

waiting: begin

TimerDone = 0;

// BCDs accept starting value inputs

countBit = 0;

setBit = 1;

// LFSR stopped

runLFSR = 0;

restartLFSR = 1;

// start timer if Run is asserted

if (Run) begin

nextState = running;

end

else begin

nextState = waiting;

end

end

running: begin

TimerDone = 0;

// BCDs do not change

countBit = 0;

setBit = 0;

// LFSR is counting

runLFSR = 1;

restartLFSR = 0;

if (Restart) begin

nextState = waiting;

end

// Check for 1 ms timeout

else if (shiftRegValue == LFSRTarget) begin

nextState = counting;

end

else if (!Run) begin

nextState = paused;

end

else begin

nextState = running;

end

end

counting: begin

TimerDone = 0;

// decrement BCD chain

countBit = 1;

setBit = 0;

// restart LFSR

runLFSR = 0;

restartLFSR = 1;

// check for overall timeout

if (currentTime[Digits\*4-1:4] == {Digits\*4-5 {1'h0}} &&

currentTime[3:0] == 4'h1) begin

// time out if LSD is one and all

// others are zero

nextState = timeout;

end

else if (Restart) begin

nextState = waiting;

end

else if (!Run) begin

nextState = paused;

end

else begin

nextState = running;

end

end

paused: begin

TimerDone = 0;

// BCDs do not change

countBit = 0;

setBit = 0;

// LFSR is paused but not restarted

runLFSR = 0;

restartLFSR = 0;

if (Restart) begin

nextState = waiting;

end

// continue if Run bit is asserted again

else if (Run) begin

nextState = running;

end

else begin

nextState = paused;

end

end

timeout: begin

TimerDone = 1;

// BCDs do not change

countBit = 0;

setBit = 0;

// LFSR stopped

runLFSR = 0;

restartLFSR = 1;

// return to wait state if Restart is asserted

nextState = waiting;

end

default: begin

// replicate default state outputs

TimerDone = 0;

countBit = 0;

setBit = 1;

runLFSR = 0;

restartLFSR = 1;

nextState = waiting;

end

endcase

end

// SeqLogic to set state

always @(posedge CLK) begin

if (!RST) begin // negative-logic reset asserted

State <= waiting;

end

else begin

State <= nextState;

end

end

endmodule

**biDirBCD.v:**

// ECE 5440

// Author: George Hodgkins

// Module: Bidirectional BCD counter

//

// This module is a BCD (decimal counter) that can count up or down depending

// on a control bit, and be connected to other of the same module for

// multi-digit operation using carry bits.

module biDirBCD (Count, SetValue, Set, OutValue, CarryOut, CLK, RST);

// parameter controlling count direction (0 = down, 1 = up)

parameter Direction = 1;

// increment/decrement bit (depending on direction)

// for a multi-counter setup, this would be connected to the CarryOut

// of another biDirBCD, unless it represents the decimal LSB

input Count;

// set value loaded in by the set bit

input [3:0] SetValue;

// set bit, loads in SetValue to the counter

input Set;

// internal counter

reg [3:0] Counter;

// value output (wired to internal counter)

output [3:0] OutValue;

assign OutValue = Counter;

// carry out bit (in a multi setup, connected to CountBit of another

// biDirBCD unless it represents the MSB)

output CarryOut;

reg CarryOut;

// system clock and reset signals

input CLK, RST;

// CombLogic to set CarryOut bit

// combinational is used so that carries can propagate through

// multiple counters in one clock cycle

always @(Count, Set, Counter) begin

// the carry bit should not be high if Set bit is asserted or

// Count bit is not

if (!Set && Count) begin

if (Direction == 1) begin // counting up

// counter about to roll over, carry out one

if (Counter == 4'd9) begin

CarryOut = 1;

end

else begin

CarryOut = 0;

end

end

else begin // Direction == 0, counting down

// counter about to roll over, carry out one

if (Counter == 4'd0) begin

CarryOut = 1;

end

else begin

CarryOut = 0;

end

end

end

else begin // no carry out if set bit is high or count bit is low

CarryOut = 0;

end

end

// sequential logic to run counter

always @(posedge CLK) begin

if (!RST) begin // negative-logic reset active

// counter reset value is dependent on Direction

if (Direction == 1) begin // counting up, reset to 0

Counter <= 4'd0;

end

else begin // counting down, reset to 9

Counter <= 4'd9;

end

end

else begin

// Set bit has priority over count bit

if (Set) begin

// ignore digit inputs greater than 9

if (SetValue <= 4'd9) begin

Counter <= SetValue;

end

end

else if (Count) begin

if (Direction == 1) begin // counting up

if (Counter == 4'd9) begin

// time to roll over (CarryOut = 1)

Counter <= 4'd0;

end

else begin

Counter <= Counter + 4'd1;

end

end

else begin // Direction == 0, counting down

if (Counter == 4'd0) begin

// time to roll over (CarryOut = 1)

Counter <= 4'd9;

end

else begin

Counter <= Counter - 4'd1;

end

end

end

end

end

endmodule

**multiReg.v:**

// ECE 5440

// Group: Gone Fishin'

// Module: Load/bidirectional shift register

//

// This module is a load/bidirectional shift register, meaning it can act as

// both a normal load register and a shift register. Arbitrary number of bits.

module multiReg (loadValue, loadBit, shiftLeft, shiftRight, currentValue, CLK, RST);

// number of bits in register

parameter Bits = 4;

// value to be loaded when load bit or reset is asserted

input [Bits-1:0] loadValue;

// bit controlling load (has priority over shifts)

input loadBit;

// bits controlling shifting (left has priority over right)

input shiftLeft, shiftRight;

// current value of the register

output [Bits-1:0] currentValue;

// we need to be able to see it inside the module

reg [Bits-1:0] storedValue;

assign currentValue = storedValue;

// system clock and reset

input CLK, RST;

always @(posedge CLK) begin

if (!RST || loadBit) begin // negative-logic reset

storedValue <= loadValue;

end

else if (shiftLeft) begin

storedValue <= storedValue << 1;

end

else if (shiftRight) begin

storedValue <= storedValue >> 1;

end

else begin

storedValue <= storedValue;

end

end

endmodule

**scorekeeper.v:**

// ECE 5440

// Group: Gone Fishin'

// Module: scorekeeper

/\*

This module keeps score for the game. It does this by using a 1-byte number. The first 4 bits

represent the tens digit, and the second 4 bits represent the ones digit. The module can recieve

1-bit signals which indicate if the score should be increased or decreased.

\*/

module scorekeeper(clk, rst, resetScore, incrementScore, decrementScore, currentScore);

input clk, rst, resetScore; //initializes score

input incrementScore, decrementScore; //signals that change the score

output reg [7:0] currentScore; //score output which goes to other modules

always @(posedge clk) begin

if(rst == 1'b0 || resetScore == 1'b1) begin

currentScore = 8'b00000000; //initalize outputs

end

else if(incrementScore == 1'b1 && decrementScore == 1'b0 && currentScore != 8'b10011001) begin

//increment score

if(currentScore[3:0] == 4'b1001) begin //if ones place is 9

currentScore = currentScore - 8'b00001001; //set ones place to 0

currentScore = currentScore + 8'b00010000; //add one to tens place

end

else

currentScore = currentScore + 8'b00000001; //otherwise, add 1 to ones place

end

else if(decrementScore == 1'b1 && incrementScore == 1'b0 && currentScore != 8'b00000000) begin

//decrement score

if(currentScore[3:0] == 4'b0000) begin //if ones place is 0

currentScore = currentScore + 8'b00001001; //set ones place to 9

currentScore = currentScore - 8'b00010000; //subtract one from tens place

end

else

currentScore = currentScore - 8'b00000001; //otherwise, subtract 1 from ones place

end

end

endmodule

**ramController.v:**

// ECE 5440

// Group: Gone Fishin'

// Module: ramController

/\*

This module controlls the RAM. When reset is pressed, it initializes the RAM to all 0s.

Then, if the game is over and the score isn't being reset, it writes the current score

to the current user's address in RAM. Otherwise, it constantly outputs the score of the

current user.

\*/

module ramController(currScore, gc\_State, scoreRst, userID, clk, rst,

scoreUserAddr0, scoreUserAddr1, scoreUserAddr2, scoreUserAddr3, scoreUserAddr4, scoreUserAddr5);

input scoreRst,clk,rst;

input [7:0] currScore;

input [2:0] userID;

input [3:0] gc\_State;

output reg [7:0] scoreUserAddr0, scoreUserAddr1, scoreUserAddr2, scoreUserAddr3, scoreUserAddr4, scoreUserAddr5;

reg flag\_ram\_init,wr;

reg [4:0] addr;

reg [7:0] ram\_data\_in;

wire [7:0] ram\_data\_out;

ram scores(addr, clk, ram\_data\_in, wr, ram\_data\_out);

always@(posedge clk) begin

if(rst == 1'b0) begin

scoreUserAddr0 = 0;

scoreUserAddr1 = 0;

scoreUserAddr2 = 0;

scoreUserAddr3 = 0;

scoreUserAddr4 = 0;

scoreUserAddr5 = 0;

flag\_ram\_init = 1'b0;

addr = 5'b00000;

wr = 1'b1; //write = 1 and read = 0

ram\_data\_in = 8'b00000000;

end

else begin

if(flag\_ram\_init == 1'b0) begin //reset RAM

wr = 1'b1;

addr = addr + 5'b00001;

ram\_data\_in = 8'b00000000;

if(addr == 5'b00110) //initialize the first 6 addresses (which are the only ones that'll be used)

flag\_ram\_init = 1'b1; //done resetting

end

else begin

addr[0] = userID[0];

addr[1] = userID[1];

addr[2] = userID[2];

addr[3] = 0;

addr[4] = 0;

if( ((gc\_State == 4'b0100) || (gc\_State == 4'b0011)) && !scoreRst && currScore > ram\_data\_out) begin

//write score when game is over and the score wasn't reset

wr = 1'b1;

ram\_data\_in = currScore;

case (userID)

3'b000: scoreUserAddr0 = currScore;

3'b001: scoreUserAddr1 = currScore;

3'b010: scoreUserAddr2 = currScore;

3'b011: scoreUserAddr3 = currScore;

3'b100: scoreUserAddr4 = currScore;

3'b101: scoreUserAddr5 = currScore;

endcase

end

else begin

wr = 1'b0; //otherwise, output the score that corresponds with the current userID

end

end

end

end

endmodule

**ram.v:**

// megafunction wizard: %RAM: 1-PORT%

// GENERATION: STANDARD

// VERSION: WM1.0

// MODULE: altsyncram

// ============================================================

// File Name: ram.v

// Megafunction Name(s):

// altsyncram

//

// Simulation Library Files(s):

// altera\_mf

// ============================================================

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// THIS IS A WIZARD-GENERATED FILE. DO NOT EDIT THIS FILE!

//

// 18.1.0 Build 625 09/12/2018 SJ Lite Edition

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//Copyright (C) 2018 Intel Corporation. All rights reserved.

//Your use of Intel Corporation's design tools, logic functions

//and other software and tools, and its AMPP partner logic

//functions, and any output files from any of the foregoing

//(including device programming or simulation files), and any

//associated documentation or information are expressly subject

//to the terms and conditions of the Intel Program License

//Subscription Agreement, the Intel Quartus Prime License Agreement,

//the Intel FPGA IP License Agreement, or other applicable license

//agreement, including, without limitation, that your use is for

//the sole purpose of programming logic devices manufactured by

//Intel and sold by Intel or its authorized distributors. Please

//refer to the applicable agreement for further details.

// synopsys translate\_off

`timescale 1 ps / 1 ps

// synopsys translate\_on

module ram (

address,

clock,

data,

wren,

q);

input [4:0] address;

input clock;

input [7:0] data;

input wren;

output [7:0] q;

`ifndef ALTERA\_RESERVED\_QIS

// synopsys translate\_off

`endif

tri1 clock;

`ifndef ALTERA\_RESERVED\_QIS

// synopsys translate\_on

`endif

wire [7:0] sub\_wire0;

wire [7:0] q = sub\_wire0[7:0];

altsyncram altsyncram\_component (

.address\_a (address),

.clock0 (clock),

.data\_a (data),

.wren\_a (wren),

.q\_a (sub\_wire0),

.aclr0 (1'b0),

.aclr1 (1'b0),

.address\_b (1'b1),

.addressstall\_a (1'b0),

.addressstall\_b (1'b0),

.byteena\_a (1'b1),

.byteena\_b (1'b1),

.clock1 (1'b1),

.clocken0 (1'b1),

.clocken1 (1'b1),

.clocken2 (1'b1),

.clocken3 (1'b1),

.data\_b (1'b1),

.eccstatus (),

.q\_b (),

.rden\_a (1'b1),

.rden\_b (1'b1),

.wren\_b (1'b0));

defparam

altsyncram\_component.clock\_enable\_input\_a = "BYPASS",

altsyncram\_component.clock\_enable\_output\_a = "BYPASS",

altsyncram\_component.init\_file = "scoresRam.mif",

altsyncram\_component.intended\_device\_family = "Cyclone V",

altsyncram\_component.lpm\_hint = "ENABLE\_RUNTIME\_MOD=NO",

altsyncram\_component.lpm\_type = "altsyncram",

altsyncram\_component.numwords\_a = 32,

altsyncram\_component.operation\_mode = "SINGLE\_PORT",

altsyncram\_component.outdata\_aclr\_a = "NONE",

altsyncram\_component.outdata\_reg\_a = "CLOCK0",

altsyncram\_component.power\_up\_uninitialized = "FALSE",

altsyncram\_component.read\_during\_write\_mode\_port\_a = "NEW\_DATA\_NO\_NBE\_READ",

altsyncram\_component.widthad\_a = 5,

altsyncram\_component.width\_a = 8,

altsyncram\_component.width\_byteena\_a = 1;

endmodule

// ============================================================

// CNX file retrieval info

// ============================================================

// Retrieval info: PRIVATE: ADDRESSSTALL\_A NUMERIC "0"

// Retrieval info: PRIVATE: AclrAddr NUMERIC "0"

// Retrieval info: PRIVATE: AclrByte NUMERIC "0"

// Retrieval info: PRIVATE: AclrData NUMERIC "0"

// Retrieval info: PRIVATE: AclrOutput NUMERIC "0"

// Retrieval info: PRIVATE: BYTE\_ENABLE NUMERIC "0"

// Retrieval info: PRIVATE: BYTE\_SIZE NUMERIC "8"

// Retrieval info: PRIVATE: BlankMemory NUMERIC "0"

// Retrieval info: PRIVATE: CLOCK\_ENABLE\_INPUT\_A NUMERIC "0"

// Retrieval info: PRIVATE: CLOCK\_ENABLE\_OUTPUT\_A NUMERIC "0"

// Retrieval info: PRIVATE: Clken NUMERIC "0"

// Retrieval info: PRIVATE: DataBusSeparated NUMERIC "1"

// Retrieval info: PRIVATE: IMPLEMENT\_IN\_LES NUMERIC "0"

// Retrieval info: PRIVATE: INIT\_FILE\_LAYOUT STRING "PORT\_A"

// Retrieval info: PRIVATE: INIT\_TO\_SIM\_X NUMERIC "0"

// Retrieval info: PRIVATE: INTENDED\_DEVICE\_FAMILY STRING "Cyclone V"

// Retrieval info: PRIVATE: JTAG\_ENABLED NUMERIC "0"

// Retrieval info: PRIVATE: JTAG\_ID STRING "NONE"

// Retrieval info: PRIVATE: MAXIMUM\_DEPTH NUMERIC "0"

// Retrieval info: PRIVATE: MIFfilename STRING "../../Users/space/Documents/U of H/Year 4/Spring 2020/ECE 5440 - Advanced Digital Design/final project/src/scoresRam.mif"

// Retrieval info: PRIVATE: NUMWORDS\_A NUMERIC "32"

// Retrieval info: PRIVATE: RAM\_BLOCK\_TYPE NUMERIC "0"

// Retrieval info: PRIVATE: READ\_DURING\_WRITE\_MODE\_PORT\_A NUMERIC "3"

// Retrieval info: PRIVATE: RegAddr NUMERIC "1"

// Retrieval info: PRIVATE: RegData NUMERIC "1"

// Retrieval info: PRIVATE: RegOutput NUMERIC "1"

// Retrieval info: PRIVATE: SYNTH\_WRAPPER\_GEN\_POSTFIX STRING "0"

// Retrieval info: PRIVATE: SingleClock NUMERIC "1"

// Retrieval info: PRIVATE: UseDQRAM NUMERIC "1"

// Retrieval info: PRIVATE: WRCONTROL\_ACLR\_A NUMERIC "0"

// Retrieval info: PRIVATE: WidthAddr NUMERIC "5"

// Retrieval info: PRIVATE: WidthData NUMERIC "8"

// Retrieval info: PRIVATE: rden NUMERIC "0"

// Retrieval info: LIBRARY: altera\_mf altera\_mf.altera\_mf\_components.all

// Retrieval info: CONSTANT: CLOCK\_ENABLE\_INPUT\_A STRING "BYPASS"

// Retrieval info: CONSTANT: CLOCK\_ENABLE\_OUTPUT\_A STRING "BYPASS"

// Retrieval info: CONSTANT: INIT\_FILE STRING "../../Users/space/Documents/U of H/Year 4/Spring 2020/ECE 5440 - Advanced Digital Design/final project/src/scoresRam.mif"

// Retrieval info: CONSTANT: INTENDED\_DEVICE\_FAMILY STRING "Cyclone V"

// Retrieval info: CONSTANT: LPM\_HINT STRING "ENABLE\_RUNTIME\_MOD=NO"

// Retrieval info: CONSTANT: LPM\_TYPE STRING "altsyncram"

// Retrieval info: CONSTANT: NUMWORDS\_A NUMERIC "32"

// Retrieval info: CONSTANT: OPERATION\_MODE STRING "SINGLE\_PORT"

// Retrieval info: CONSTANT: OUTDATA\_ACLR\_A STRING "NONE"

// Retrieval info: CONSTANT: OUTDATA\_REG\_A STRING "CLOCK0"

// Retrieval info: CONSTANT: POWER\_UP\_UNINITIALIZED STRING "FALSE"

// Retrieval info: CONSTANT: READ\_DURING\_WRITE\_MODE\_PORT\_A STRING "NEW\_DATA\_NO\_NBE\_READ"

// Retrieval info: CONSTANT: WIDTHAD\_A NUMERIC "5"

// Retrieval info: CONSTANT: WIDTH\_A NUMERIC "8"

// Retrieval info: CONSTANT: WIDTH\_BYTEENA\_A NUMERIC "1"

// Retrieval info: USED\_PORT: address 0 0 5 0 INPUT NODEFVAL "address[4..0]"

// Retrieval info: USED\_PORT: clock 0 0 0 0 INPUT VCC "clock"

// Retrieval info: USED\_PORT: data 0 0 8 0 INPUT NODEFVAL "data[7..0]"

// Retrieval info: USED\_PORT: q 0 0 8 0 OUTPUT NODEFVAL "q[7..0]"

// Retrieval info: USED\_PORT: wren 0 0 0 0 INPUT NODEFVAL "wren"

// Retrieval info: CONNECT: @address\_a 0 0 5 0 address 0 0 5 0

// Retrieval info: CONNECT: @clock0 0 0 0 0 clock 0 0 0 0

// Retrieval info: CONNECT: @data\_a 0 0 8 0 data 0 0 8 0

// Retrieval info: CONNECT: @wren\_a 0 0 0 0 wren 0 0 0 0

// Retrieval info: CONNECT: q 0 0 8 0 @q\_a 0 0 8 0

// Retrieval info: GEN\_FILE: TYPE\_NORMAL ram.v TRUE

// Retrieval info: GEN\_FILE: TYPE\_NORMAL ram.inc FALSE

// Retrieval info: GEN\_FILE: TYPE\_NORMAL ram.cmp FALSE

// Retrieval info: GEN\_FILE: TYPE\_NORMAL ram.bsf FALSE

// Retrieval info: GEN\_FILE: TYPE\_NORMAL ram\_inst.v FALSE

// Retrieval info: GEN\_FILE: TYPE\_NORMAL ram\_bb.v TRUE

// Retrieval info: LIB\_FILE: altera\_mf

**SevenSeg.v:**

// ECE 5440

// Group: GoneFishin'

// Module: SevenSeg

// Provided a 4-bit binary value, between 0000 - 1111, this module will return the 7-bit output signal

// to properly display the number in Hexadecimal.

// Example:

// input:

// num\_bi: 1010

// output:

// num\_disp: 'A'

module SevenSeg (num\_bi, num\_disp);

input [3:0] num\_bi;

output [6:0] num\_disp;

reg [6:0] num\_disp;

always @ (num\_bi) begin

case(num\_bi)

4'b0000: num\_disp = 7'b1000000;//0

4'b0001: num\_disp = 7'b1111001;//1

4'b0010: num\_disp = 7'b0100100;//2

4'b0011: num\_disp = 7'b0110000;//3

4'b0100: num\_disp = 7'b0011001;//4

4'b0101: num\_disp = 7'b0010010;//5

4'b0110: num\_disp = 7'b0000010;//6

4'b0111: num\_disp = 7'b1111000;//7

4'b1000: num\_disp = 7'b0000000;//8

4'b1001: num\_disp = 7'b0011000;//9

4'b1010: num\_disp = 7'b0001000;//10 'A'

4'b1011: num\_disp = 7'b0000011;//11 'b'

4'b1100: num\_disp = 7'b1000110;//12 'C'

4'b1101: num\_disp = 7'b0100001;//13 'd'

4'b1110: num\_disp = 7'b0000110;//14 'E'

4'b1111: num\_disp = 7'b0001110;//15 'F'

endcase

end

endmodule

**button\_shaper.v:**

// ECE 5440

// Group: GoneFishin'

// Module: button\_shaper

// This module will convert a push button that is active-low to active-high and only replicate a one cycle pulse when the

// the button is pressed.

// Example:

// input:

// \_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_

// button\_in: |\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_|

// \_\_\_\_\_\_ \_\_\_\_\_\_

// clk:\_\_\_\_\_| |\_\_\_\_\_\_| |\_\_\_\_\_\_

// output:

// \_\_\_\_\_\_\_\_\_\_\_\_\_\_

// button\_out:\_\_\_\_\_| |\_\_\_\_\_\_\_\_\_\_

module button\_shaper (button\_in, clk, rst, button\_out);

input button\_in;

output reg button\_out;

input clk, rst;

parameter S\_Init = 0, S\_Pulse = 1, S\_Wait = 2;

reg [1:0] State, StateNext; // 2-bit

// Sequential Logic (Procedure 1)

always @(State, button\_in) begin

case (State)

S\_Init: begin

// stay in this state until button intially pressed

button\_out <= 0;

if(button\_in == 0) begin

StateNext <= S\_Pulse;

end

else begin

StateNext <= S\_Init;

end

end

S\_Pulse: begin

// stay in this state for 1 clk cycle

button\_out <= 1;

StateNext <= S\_Wait;

end

S\_Wait: begin

// stay in this state until button is released

button\_out <= 0;

if(button\_in == 0) begin

StateNext <= S\_Wait;

end

else begin

StateNext <= S\_Init;

end

end

endcase

end

// StateReg (Procedure 2)

always @(posedge clk) begin

if (rst == 0) begin

// if rst is click, go to intial state

State <= S\_Init;

end

else begin

State <= StateNext;

end

end

endmodule