HARDWARE-BASED TRUE RANDOM NUMBER GENERATOR AND TIMER CONTROL ON FPGA

Author: Sriram Pandiyan

Course: ECE 6370

Introduction

FPGA-Based Pattern Matching Game implements peripheral functionalities to make the system more robust and thus present a more enjoyable play experience. This is achieved through the incorporation of individual gameplay features such as the addition of a random number generator to allow for uniquely randomized matches and a game timer mechanism to add a layer of challenge and an incentive to replay the game.

The complete system operates on a DE2-115 FPGA Board. Initially, the system will not react to any input changes other than user input switches and button presses for setting a 4-digit password combination. If the combination is incorrect, the system will cycle getting a new combination until a correct password is authenticated. If the password is correct, the user will have the option to select a desired game time of up to 99 seconds. Once read, the user will press a button to commence the game.

The 1-player game consists of having the system generate a random 4-bit number, the player will then try to come up with a second 4-bit number that when added to the first one results in the value 1111 (binary). These numbers are all displayed through on-board 7-segment displays in their equivalent hexadecimal form, adding a level of complexity to the math by having to first transform a hexadecimal value into its binary representation before attempting to solve the addition. Every time a correct addition is obtained, a green LED will light up to notify the player, at this point he/she is allowed to press a button to generate a different random number and try again. This procedure will repeat for as many rounds as possible until the game timer reaches 0. Once the match is complete, the player is free to modify the game timer and restart the game with the goal of finishing as many rounds as possible in a designated time and setting new personal records. Finally, when the game session is complete, the user is free to turn off or reset the system.

System Architecture Design

Game system functionality was developed using Verilog (a Hardware Description Language) through ModelSim (a Verilog development and simulation environment). The system's major components were designed as separate modules. These modules are for a 4-Bit Adder, 7-Segment Display, LED Verification, Load Register, Button Shaper, Access Controller, 4-Bit Random Number Generator, and Game Timer. Using these components, the overall system architecture was developed as a Verilog top module, interconnecting various iterations of these objects to create a mental binary math game with access control and individual gameplay mechanisms.

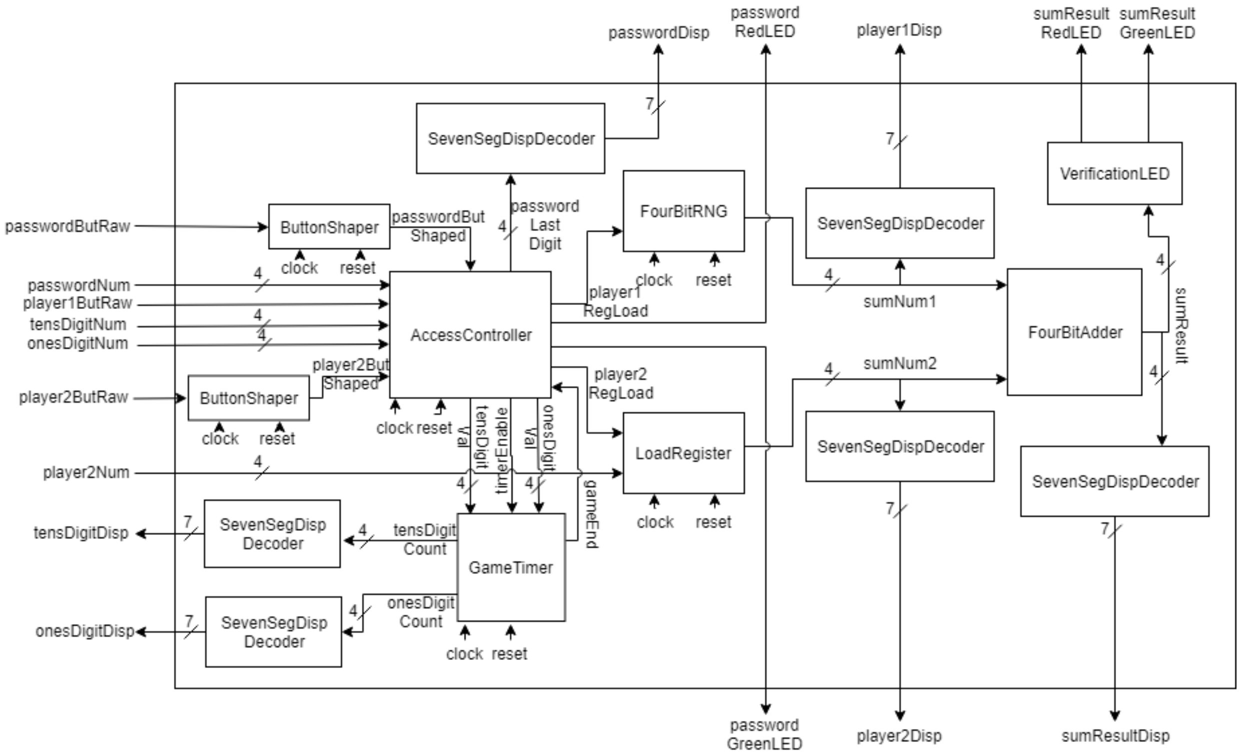

Figure 1: Top-Module System Architecture Diagram

As shown in the top module diagram (Figure 1), input and outputs are available to the users but there are a substantial number of necessary connections inside the system, not visible from the outside. Each new/updated component module operates as follow:

- The FourBitRNG module takes in a button press input signal (buttonPress) which is used to count a number of clock cycles and come up with a 4-bit random number output (randomNumber). If the reset signal is activated low, output is set to the number 0.
- The GameTimer module takes in a countdown enabling input signal (enableCount) and two 4-bit inputs for the initial timer value (digitTensInitValue, digitOnesInitValue). These numbers are used to commence a 1s count from this initial value down to 0. During this process, two 4-bit outputs are used to display the current time left (digitTensDisp, digitOnesDisp) and once the final value is reached, an output flag (timeout) is used to stop the game match. Furthermore, when the reset signal is activated low, outputs are set to their respective initial state.
- The AccessController module (modified) takes in a 4-bit input (passwordDigit) and a 1-bit input signal (nextButton) four times to verify a correct password for user authentication. It also takes in two 1-bit input signals (player1LoadSig, player2LoadSig) allowing the two operand values to be loaded to the game. Every time the clock reaches a rising-edge, the module checks for a password digit and button press (passwordDigit = ####, nextButton = 1). After 4 presses the password is verified, if it is correct the selected operand values are allowed to be displayed, accomplished through 1-bit outputs

(player1Transmit, player2Transmit), and verification LED signals (redLED, greenLED) are updated accordingly. Otherwise, for incorrect passwords, these outputs will always remain in their respective off state. After password verification, two 4-bit timer input signals (digitTensValue, digitOnesValue) are transferred to respective 4-bit outputs (digitTensTransmit, digitOnesTransmit) and after one more button press, the timer enable output (timerEnable) is activated high, allowing the game match to commence. Finally, if the reset signal is activated low, regardless of other inputs, all output signals are set to their equivalent off state.

Simulation Results

To ensure correct operation in the design phase of this laboratory assignment, necessary testbench modules for new mechanisms and lower-level components were created. During all module testing a simulated clock runs with a period of 20 [ns] and different input combinations are sequentially generated to observe the output of the modules in ModelSim. All waveforms should line up with their respective operation described in each individual module's comments

For the FourBitRNG module, input combinations were generated to observe the counting of clock cycles during a button press that ends in the output of a random value. Successful simulations are shown in Figures 2 and 3.

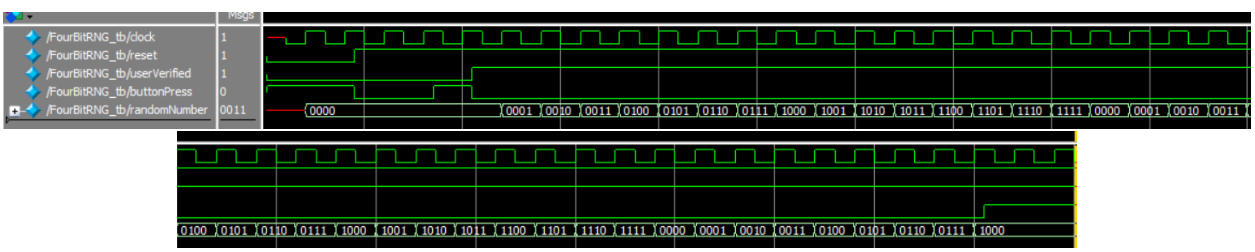

Figure 2: Simulation Results for 4-Bit Random Number Generator Module (Press 1)

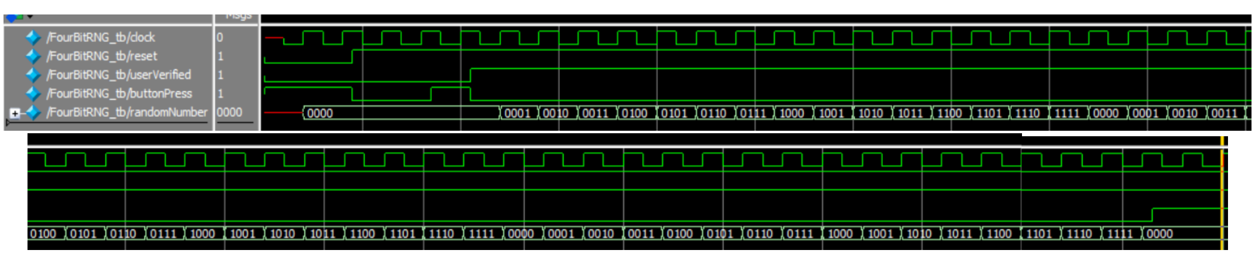

Figure 3: Simulation Results for 4-Bit Random Number Generator Module (Press 2)

For the GameTimer module, tests were generated for several building block component modules. First, for the Timer1ms module a timeout signal was visualized to assert high after a specific number of clock cycles passed, in this case 50,000. Successful simulations are shown in Figures 4 and 5.

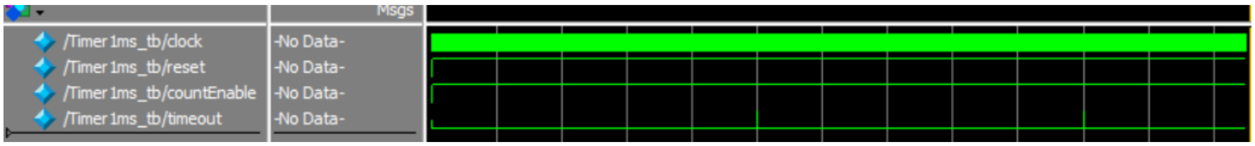

Figure 4: Simulation Results for 1ms Timer Module

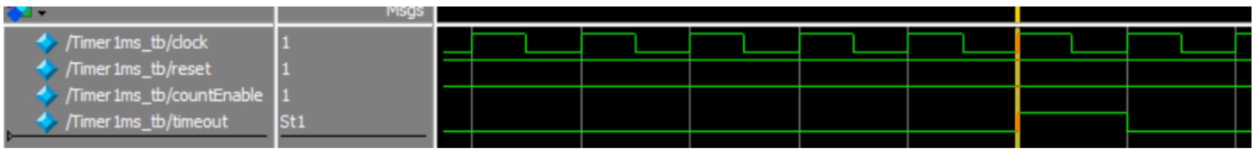

Figure 5: Simulation Results for 1ms Timer Module (Pulse Visualization)

Similarly, for the Count100 module, a timeout signal was visualized to assert high after a specific number of count cycles passed, in this case 100. Successful simulations are shown in Figures 6 and 7.

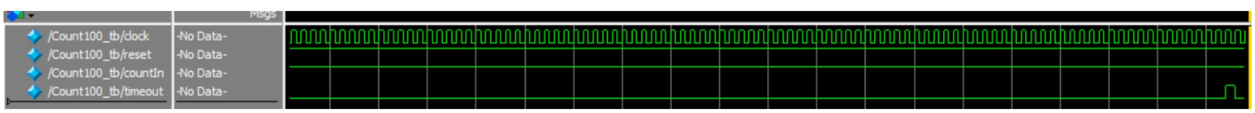

Figure 6: Simulation Results for 100 Value Counter Module

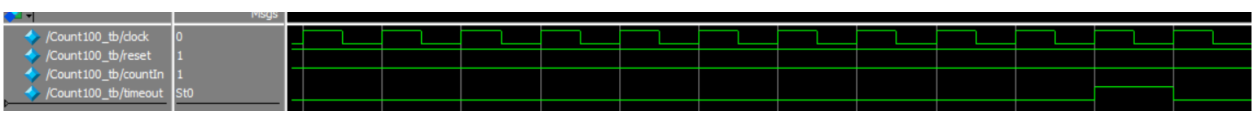

Figure 7: Simulation Results for 100 Value Counter Module (Pulse Visualization)

Other timer and counter modules were created similarly and then used to construct the Timer1ms module hierarchically. Because these other building block modules utilize the same logic as the ones above (there's only a slight change in one parameter), testing for these was not required.

Next, for the TimerDigit module, several inputs are generated to allow for a countdown from the number 13 to 0. This operation follows the procedure for subtraction by hand of 2-digit decimal numbers, i.e. borrowing digits from the left neighbor. Successful simulation is shown in Figure 8.

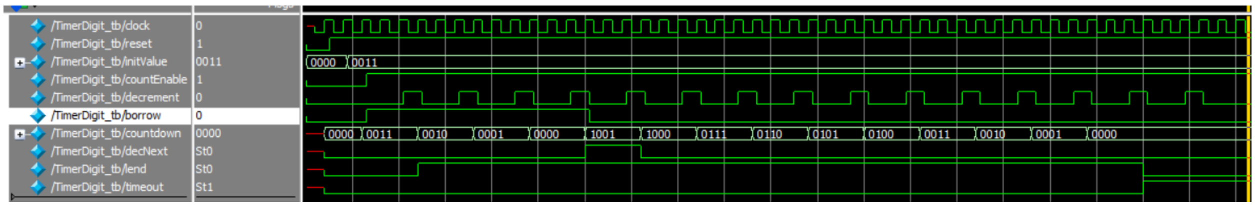

Figure 8: Simulation Results for Digit Timer Module (Countdown from 13)

FPGA Board Testing Results

After successful simulation results in software, the system was implemented physically on a DE2-115 FPGA board. To do this, a Quartus project was created with all necessary component modules and the top module design. The software pin planner was used to connect input and output bits to respective on-board 7-segment displays, input switches, input buttons, and colored LEDs. Finally, the system design was flashed into the board for final testing.

To verify correct operation a complete game session was tested. This test consisted of starting up the system, logging in with a correct password, setting a game timer and playing a match. Once completed, a different time setting was selected, and another match was played. Throughout the whole process, correct inputs and outputs were verified while ensuring that incorrect inputs and outputs were ignored in accordance to design. In the end, the reset function was utilized to restart the system.

Note that displays are aligned from left to right in the following order: random number, player number, summation result, timer digits, and password digit. Similarly, switches are aligned from left to right in the order: Player 2, timer digits, and password digit (skipping 1 switch in-between each set). Button alignment is as follows: player, random number generator, system control, and reset. Finally, verification LEDs for matching case summation are located between the game number and game configuration displays, those for password authentication are located above the rightmost switch.

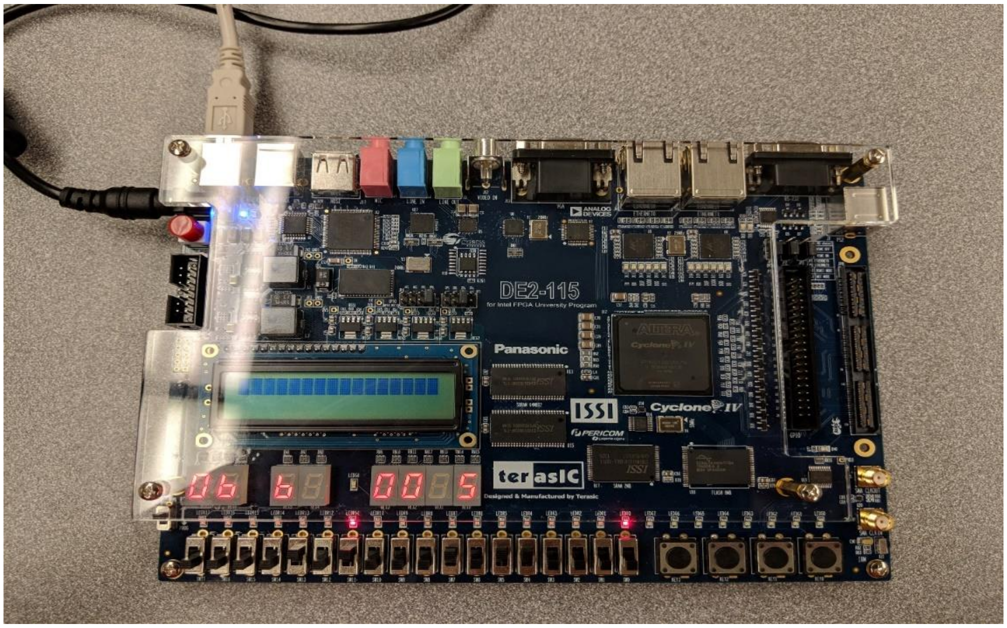

Figure 9: Generated Random Number 1

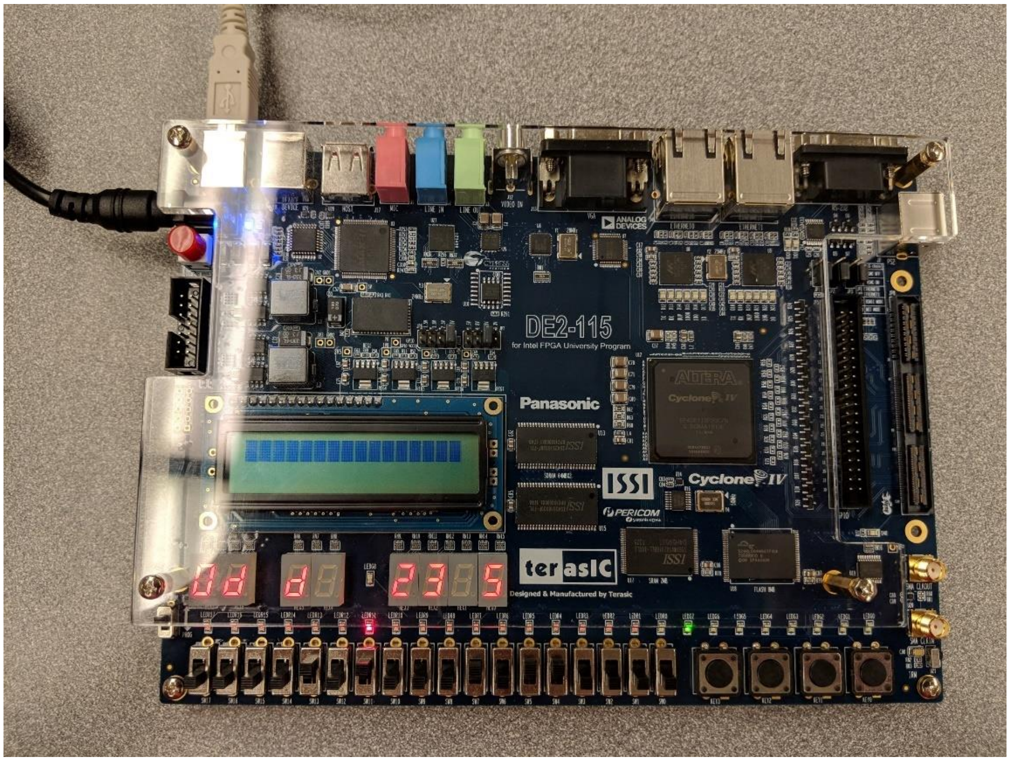

Figure 10: Generated Random Number 2

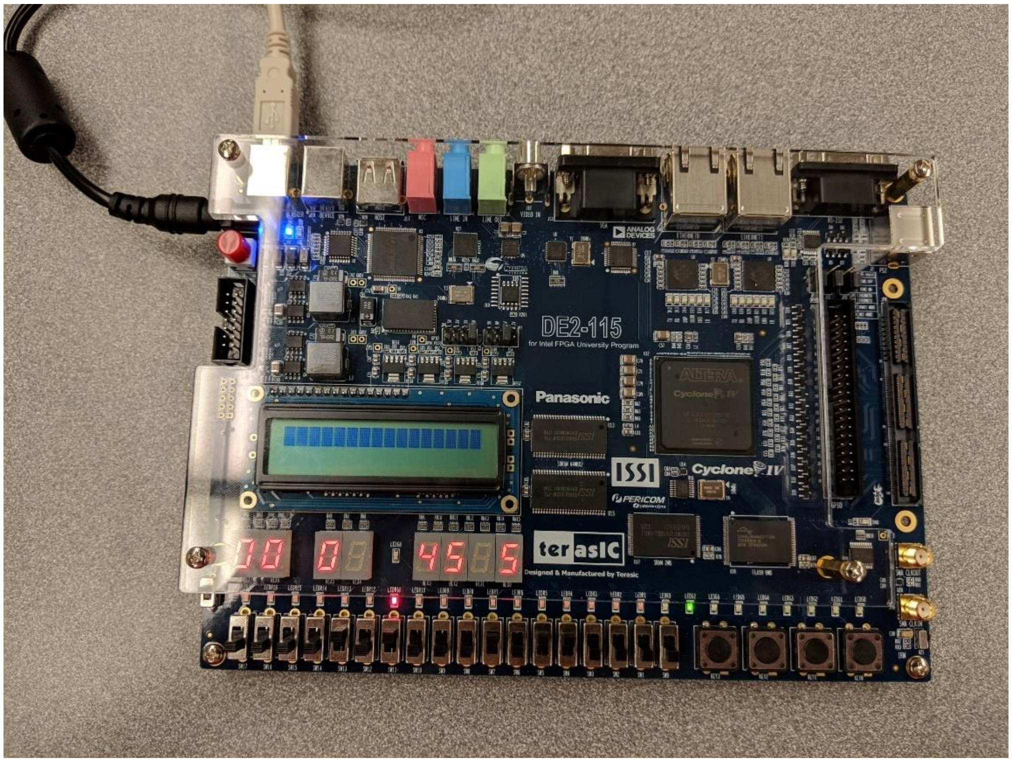

Figure 11: Game Time Initialized

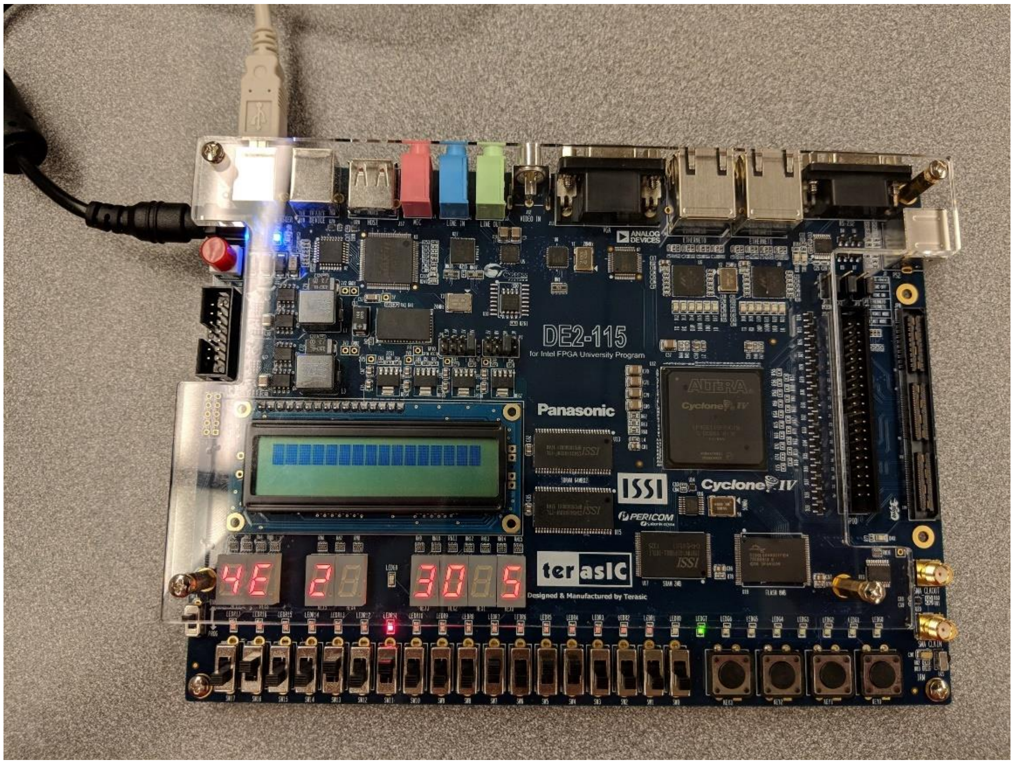

Figure 12: Game Time Running

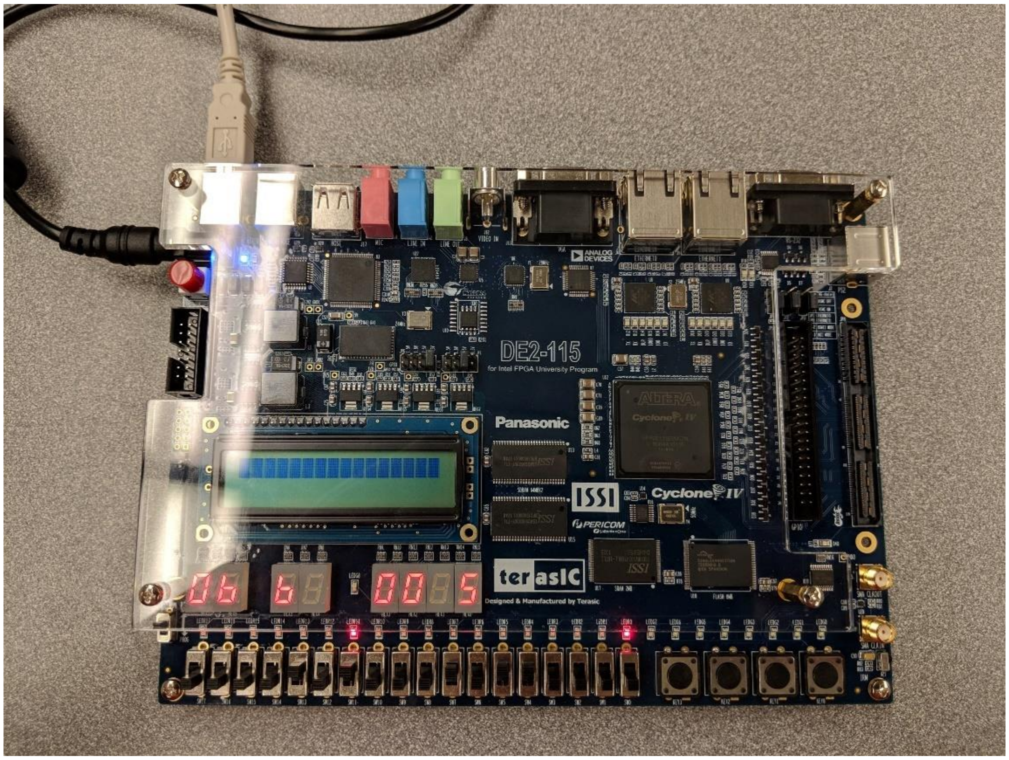

Figure 13: Game Time Finalized

Although not shown here, many other cases were tested to verify correct random number generation and game timer configuration performance.

Conclusion

Game system improvements based on new and updated component modules, as well as a redesigned top-level architecture resulted in well-coded Verilog modules that allow for an efficient and enjoyable individual gameplay experience. Because of good documentation, all utilized Verilog modules are easy to understand and modify for future applications and improvements.

Finally, the developed Hardware-Based True Random Number Generator and Timer Control on FPGS was a success, making this laboratory assignment an effective introduction hardware design language hierarchical system design and complex architecture.

Appendix

FourBitRNG.v (4-Bit Random Number Generator Module)

```
// ECE 6370
// Author: Sriram Pandiyan, 6101
// FourBitRNG
// This module operates as a random number generator. It uses a counter working on a hardware
based timer
// to create a random number based on the length of time a button press is held down. Since the
clock runs
// at a very fast speed compared to human operation, it is impossible to replicate values when
implemeted.


module FourBitRNG(buttonPress, randomNumber, clock, reset);

	input clock, reset;   // System clock and reset signals
	input buttonPress;   // Button press down and correct user verification signal
	output[3:0] randomNumber;   // Random number generated

	assign buttonPressInv = ~buttonPress;   // Inversion of button press to active high
operationg

	Counter Counter_RNG(buttonPressInv, randomNumber, clock, reset);   // RNG Counter
object instantiation

endmodule
```

FourBitRNG_tb.v (4-Bit Random Number Generator Testbench Module)

```
// ECE 6370
// Author: Sriram Pandiyan, 6101
// FourBitRNG_tb
// This module acts as a testbench for the FourBitRNG. It generates different input combinations
and operational signals
// to verify correct output generation from the random number generator implementation.


`timescale 1ns/100ps

module FourBitRNG_tb();
```

```
	reg clock, reset;
	reg buttonPress, userVerified;
	wire[3:0] randomNumber;

	FourBitRNG DUT_FourBitRNG(buttonPress, userVerified, randomNumber, clock,
reset);

	always   // Procedure to create 20ns clock
	begin
	  #10 clock = 1'b0;   // Low for 10ns
	  #10 clock = 1'b1;   // High for 10ns
	end

	initial   // Procedure to generate tests
	begin
	  reset = 0; buttonPress = 1; userVerified = 0;   // Initially reset all signals
	  @(posedge clock); @(posedge clock);
	  #5 reset = 1; buttonPress = 0;   // No user verification, counting does not take place
	  @(posedge clock); @(posedge clock);
	  #5 buttonPress = 1;
	  @(posedge clock);
	  #5 userVerified = 1; buttonPress = 0;   // User verification and button press, counting
takes place
	  @(posedge clock); @(posedge clock); @(posedge clock); @(posedge clock);
@(posedge clock);
	  @(posedge clock); @(posedge clock); @(posedge clock); @(posedge clock);
@(posedge clock);
	  @(posedge clock); @(posedge clock); @(posedge clock); @(posedge clock);
@(posedge clock);
	  @(posedge clock); @(posedge clock); @(posedge clock); @(posedge clock);
@(posedge clock);
	  @(posedge clock); @(posedge clock); @(posedge clock); @(posedge clock);
@(posedge clock);
	  @(posedge clock); @(posedge clock); @(posedge clock); @(posedge clock);
@(posedge clock);
	  @(posedge clock); @(posedge clock); @(posedge clock); @(posedge clock);
@(posedge clock);
	  @(posedge clock); @(posedge clock); @(posedge clock); @(posedge clock);
@(posedge clock);
	  #5 buttonPress = 1;
	  @(posedge clock); @(posedge clock);
	end
```

endmodule

Timer1ms.v (1ms Timer Module)

```
// ECE 6370
// Author: Sriram Pandiyan, 6101
// Timer1ms
// This module operates as a 1ms timer. It utilizes a 16-bit variable to count values from 0 to
50000 necessary for a
// 50MHz clock. Additionally, it incorporates reset functionality.


module Timer1ms(countEnable, timeout, clock, reset);

    input clock, reset;   // System clock and reset signals
    input countEnable;   // Counting enable
    output reg timeout;   // Timeout when 1ms is reached
    reg[15:0] count;   // Counting variable

    // Sequential logic
    always @(posedge clock)
    begin
      if (reset == 0)   // Reset operation
      begin
        count <= 0;
        timeout <= 0;
      end
      else
      begin
        timeout <= 0;
        if (countEnable == 1)   // Count enable signal activated
        begin                          // Count 1ms (value range: 0 - 50000)
          if (count == 50000 || count > 50000)
          begin
                count <= 0;
                timeout <= 1;
              end
              else
              begin
                count <= count + 1;
```

```
            end
        end
      end
    end

endmodule
```

Timer1ms_tb.v (1ms Timer Testbench Module)

```
// ECE 6370
// Author: Sriram Pandiyan, 6101
// Timer1ms_tb
// This module acts as a testbench for the Timer1ms. It generates different input combinations
and operational signals
// to verify correct output generation from the 1ms timer implementation.


`timescale 1ns/100ps

module Timer1ms_tb();

    reg clock, reset;
    reg countEnable;
    wire timeout;

    Timer1ms DUT_Timer1ms(countEnable, timeout, clock, reset);   // Object Instantiation

    always   // Procedure to create 20ns clock
    begin
      #10 clock = 1'b0;   // Low for 10ns
      #10 clock = 1'b1;   // High for 10ns
    end

    initial   // Procedure to generate tests
    begin
      reset = 0; countEnable = 0;   // Initially reset all signals
      @(posedge clock);
      #5 reset = 1; countEnable = 1;
    end

endmodule
```

Count100.v (100 Value Counter Module)

```
// ECE 6370
// Author: Sriram Pandiyan, 6101
// Count100
// This module operates as a basic counter that utilizes a 7-bit variable to count values from 0 to 100.
// Additionally, it incorporates reset functionality.


module Count100(countIn, timeout, clock, reset);

        input clock, reset;   // System clock and reset signals
        input countIn;   // Count input signal
        output reg timeout;   // Timeout when 10 is reached
        reg[6:0] count;   // Counting variable

        // Sequential Logic
        always @(posedge clock)
        begin
          if (reset == 0)   // Reset operation
          begin
            count <= 0;
            timeout <= 0;
          end
          else
          begin
            timeout <= 0;
            if (countIn == 1)   // Count input signal received
            begin                    // Increase count (value range: 0 - 100)
              if (count == 100 || count > 100)
                begin
                  count <= 0;
                  timeout <= 1;
                end
                else
                begin
                  count <= count + 1;
                end
            end
```

```
    end
  end

endmodule
```

Count100_tb.v (100 Value Counter Testbench Module)

```
// ECE 6370
// Author: Sriram Pandiyan, 6101
// Count100_tb
// This module acts as a testbench for the Count100. It generates different input combinations and operational signals
// to verify correct output generation from the 100 value counter implementation.


`timescale 1ns/100ps

module Count100_tb();

  reg clock, reset;
  reg countIn;
  wire timeout;

  Count100 DUT_Count100(countIn, timeout, clock, reset);   // Object Instantiation

  always   // Prcedure to create 20ns clock
  begin
    #10 clock = 1'b0;   // Low for 10ns
    #10 clock = 1'b1;   // High for 10ns
  end

  initial   // Procedure to generate tests
  begin
    reset = 0; countIn = 0;   // Initially reset all signals
    @(posedge clock);
    #5 reset = 1; countIn = 1;
  end

endmodule
```

Count10.v (10 Value Counter Module)

```
// ECE 6370
// Author: Sriram Pandiyan, 6101
// Count10
// This module operates as a basic counter that utilizes a 4-bit variable to count values from 0 to
10.
// Additionally, it incorporates reset functionality.


module Count10(countIn, timeout, clock, reset);

        input clock, reset;   // System clock and reset signals
        input countIn;   // Count input signal
        output reg timeout;   // Timeout when 10 is reached
        reg[3:0] count;   // Counting variable

        // Sequential Logic
        always @(posedge clock)
        begin
          if (reset == 0)   // Reset operation
          begin
            count <= 0;
            timeout <= 0;
          end
          else
          begin
            timeout <= 0;
            if (countIn == 1)   // Count input signal received
            begin                        // Increase count (value range: 0 - 10)
              if (count == 10 || count > 10)
                  begin
                    count <= 0;
                    timeout <= 1;
                  end
                  else
                  begin
                    count <= count + 1;
                  end
            end
          end
        end

endmodule
```

Timer100ms.v (100ms Timer Module)

```
// ECE 6370
// Author: Sriram Pandiyan, 6101
// Timer100ms
// This module operates as a 100ms timer. It utilizes a 1ms timer and a 100 value counter to
achieve 100ms timer functionality.
// Additionally, it incorporates reset functionality.


module Timer100ms(countEnable, timeout, clock, reset);

	input clock, reset;   // System clock and reset signals
	input countEnable;   // Counting enable
	output timeout;   // Timeout when 100ms is reached
	wire partialTimeout;   // 1ms timeout driving the 100 value counter

	Timer1ms Timer1ms_1(countEnable, partialTimeout, clock, reset);   // 1ms Timer object
instantiation
	Count100 Count100_1(partialTimeout, timeout, clock, reset);   // 100 Value Counter
object instantiation

endmodule
```

Timer1s.v (1s Timer Module)

```
// ECE 6370
// Author: Sriram Pandiyan, 6101
// Timer1s
// This module operates as a 1s timer. It utilizes a 100ms timer and a 10 value counter to achieve
1s timer functionality.
// Additionally, it incorporates reset functionality.


module Timer1s(countEnable, timeout, clock, reset);

	input clock, reset;   // System clock and reset signals
	input countEnable;   // Counting enable
	output timeout;   // Timeout when 1s is reached
```

```
    wire partialTimeout;   // 100ms timeout driving the 10 value counter

    Timer100ms Timer100ms_1(countEnable, partialTimeout, clock, reset);   // 100ms Timer
object instantiation
    Count10 Count10_1(partialTimeout, timeout, clock, reset);   // 10 Value Counter object
instantiation

endmodule
```

TimerDigit.v (Digit Timer Module)

```
// Author: Sriram Pandiyan, 6101
// TimerDigit
// This module operates as a timer digit. It utilizes a logic to sequentially decrease the timer while
borrowing and lending
// values to other similar timer digit modules. When the end value of zero is reached, it generates
a timeout signal.

module TimerDigit(initValue, decrement, countEnable, timeout, countdown, borrow, lend,
decNext, clock, reset);

    input clock, reset;   // System clock and reset signals
    input[3:0] initValue;   // Initial digit value
    input decrement;   // Decrement signal
    input countEnable;   // Enable digit countdown
    input borrow;   // Can borrow?
    output reg[3:0] countdown;   // Count value to display
    output reg timeout;   // Timeout when digit reaches 0
    output reg lend;   // Can lend
    output reg decNext;   // Decrement next digit place

    // Sequential logic
    always @(posedge clock)
    begin
      if (reset == 0 || countEnable == 0)
      begin
        countdown <= initValue;
        timeout <= 0;
        lend <= 0;
        decNext <= 0;
      end
```

```
          else
          begin
            if (decrement == 1)
            begin
              if (countdown > 0)
                begin
                  countdown = countdown - 1;
                  timeout <= 0;
                  lend <= 1;
                  decNext <= 0;
                end
                else
                begin
                  if (borrow == 1)
                  begin
                    countdown <= 9;
                    timeout <= 0;
                    lend <= 1;
                    decNext <= 1;
                  end
                  else
                  begin
                    timeout <= 1;
                    lend <= 0;
                    decNext <= 0;
                  end
                end
            end
          end
        end

endmodule
```

TimerDigit_tb.v (Timer Digit Testbench Module)

```
// Author: Sriram Pandiyan, 6101
// TimerDigit
// This module operates as a timer digit. It utilizes a logic to sequentially decrease the timer while
borrowing and lending
// values to other similar timer digit modules. When the end value of zero is reached, it generates
a timeout signal.
```

```
module TimerDigit(initValue, decrement, countEnable, countdown, borrowRes, borrowReq,
lend, clock, reset);

        input clock, reset;   // System clock and reset signals
        input[3:0] initValue;   // Initial digit value
        input decrement;   // Decrement signal
        input countEnable;   // Enable digit countdown
        input borrowRes;   // Response to borrow
        output reg[3:0] countdown;   // Count value to display
        output reg borrowReq;   // Request to borrow
        output reg lend;   // Can lend

        // Sequential logic
        always @(posedge clock)
        begin
          if (reset == 0 || countEnable == 0)
          begin
            countdown <= initValue;
            borrowReq <= 0;
            lend <= 1;
          end
          else
          begin
            if (countdown > 0)
            begin
              borrowReq <= 0;
              lend <= 1;
                 if (decrement == 1)
                 begin
                   countdown <= countdown - 1;
                 end
            end
            else
            begin
                 if (borrowRes == 1)
                 begin
                   lend <= 1;
                   if (decrement == 1)
                   begin
                     countdown <= 9;
                     borrowReq <= 1;
                   end
```

```
        end
        else
        begin
          lend <= 0;
        end
      end
    end
  end

endmodule
```

GameTimer.v (Game Timer Module)

```
// Author: Sriram Pandiyan, 6101
// GameTimer
// This module operates as a game timer. It uses two separate digit displays to count from a
selected configurable time down
// to 0 in 1 second intervals.


module GameTimer(countEnable, digitTensInitValue, digitOnesInitValue, timeout,
digitTensDisp, digitOnesDisp, clock, reset);

    input clock, reset;   // System clock and reset signals
    input countEnable;   // Counting enable
    input[3:0] digitTensInitValue, digitOnesInitValue;   // Initial timer values
    output timeout;   // Timeout after game timer finishes
    output[3:0] digitTensDisp, digitOnesDisp;   // Values to display
    wire partialTimeout;   // Intermediary timeouts
    wire borrowTen, lendTen;   // Intermediary variables

    Timer1s Timer1s_1(countEnable, partialTimeout, clock, reset);   // 1s Timer object
instantiation
    TimerDigit TimerDigit_Tens(digitTensInitValue, borrowTen, countEnable,
digitTensDisp,
                    0, , lendTen, clock, reset);   // Tens place Digit Timer object
instantiation
    TimerDigit TimerDigit_Ones(digitOnesInitValue, partialTimeout, countEnable,
digitOnesDisp,
                    lendTen, borrowTen, timeout, clock, reset);         // Ones place
Digit Timer object instantiation
```

endmodule

AccessController.v (Access Controller Module)

```
// Author: Sriram Pandiyan, 6101
// AccessController
// This module operates as an access controller authorization. The operation is that of a sequential state machine.
// There are five distinct states that each verify one digit in a 4-digit password, in the fifth and final state
// if the password was authenticated correctly, access is granted to continue with further system operation. In
// addition a set of red and green LEDs help visualize correct authentication.


module AccessController(passwordDigit, player1LoadSig, player2LoadSig, nextButton, player1Transmit, player2Transmit,
                        passwordTransmit, redLED, greenLED,
                        digitTensValue, digitOnesValue, digitTensTransmit, digitOnesTransmit, timeout, timerEnable,
                        clock, reset);
    // Correct password: 6101

    input[3:0] passwordDigit;   // Password digit
    input player1LoadSig, player2LoadSig, nextButton;   // Player load signals and button press for next digit
    input clock, reset;   // System clock and reset signals
    input[3:0] digitTensValue, digitOnesValue;   // Timer digit values
    input timeout;   // Timeout from game timer
    output reg player1Transmit, player2Transmit;   // Player load signals output
    output reg[3:0] passwordTransmit;   // Password digit output
    output reg redLED, greenLED;   // Control for red and green LED
    output reg[3:0] digitTensTransmit, digitOnesTransmit;   // Timer digit value transmit
    output reg timerEnable;

    reg mismatch = 1'b0;   // Mismatch to signal an incorrect digit
    reg[2:0] state;   // State register

    parameter DIGIT1 = 0, DIGIT2 = 1, DIGIT3 = 2, DIGIT4 = 3, CHECK = 4, GAME = 5, END = 6;   // Finite State Machine states
```

```
// One-Procedure Sequential FSM Logic
always @(posedge clock)
begin
  if (reset == 0)
  begin
    passwordTransmit <= 4'b0000;
    player1Transmit <= 1'b1; player2Transmit <= 1'b0;
    redLED <= 1'b1; greenLED <= 1'b0;
    digitTensTransmit <= 4'b0000; digitOnesTransmit <= 4'b0000;
    timerEnable <= 0;
    state <= DIGIT1;
  end
  else
  begin
    case (state)
      DIGIT1:   // Receive first digit
         begin
           player1Transmit <= 1'b1; player2Transmit <= 1'b0;
           redLED <= 1'b1; greenLED <= 1'b0;
           digitTensTransmit <= 4'b0000; digitOnesTransmit <= 4'b0000;
           timerEnable <= 0;
           mismatch <= 1'b0;   // First digit, mismatch not possible
           if (nextButton == 1'b1)   // If button is pressed, verify digit and proceed to next
state
           begin
             if (passwordDigit != 4'b0110)   // If incorrect digit, mark mismatch
             begin
               mismatch <= 1'b1;
             end
             passwordTransmit <= passwordDigit;
             state <= DIGIT2;
           end
           else   // If button is not pressed stay in this state
           begin
             state <= DIGIT1;
           end
         end
         DIGIT2:   // Receive second digit
         begin
           player1Transmit <= 1'b1; player2Transmit <= 1'b0;
           redLED <= 1'b1; greenLED <= 1'b0;
           digitTensTransmit <= 4'b0000; digitOnesTransmit <= 4'b0000;
           timerEnable <= 0;
```

```
  if (nextButton == 1'b1)   // If button is pressed, verify digit and proceed to next state
  begin
    if (passwordDigit != 4'b0001)   // If incorrect digit, mark mismatch
    begin
      mismatch <= 1'b1;
    end
    passwordTransmit <= passwordDigit;
    state <= DIGIT3;
  end
  else   // If button is not pressed stay in this state
  begin
    state <= DIGIT2;
  end
end
DIGIT3:   // Receive third digit
begin
  player1Transmit <= 1'b1; player2Transmit <= 1'b0;
  redLED <= 1'b1; greenLED <= 1'b0;
  digitTensTransmit <= 4'b0000; digitOnesTransmit <= 4'b0000;
  timerEnable <= 0;
  if (nextButton == 1'b1)   // If button is pressed, verify digit and proceed to next state
  begin
    if (passwordDigit != 4'b0000)   // If incorrect digit, mark mismatch
    begin
      mismatch <= 1'b1;
    end
    passwordTransmit <= passwordDigit;
    state <= DIGIT4;
  end
  else   // If button is not pressed stay in this state
  begin
    state <= DIGIT3;
  end
end
DIGIT4:   // Receive fourth digit
begin
  player1Transmit <= 1'b1; player2Transmit <= 1'b0;
  redLED <= 1'b1; greenLED <= 1'b0;
  digitTensTransmit <= 4'b0000; digitOnesTransmit <= 4'b0000;
  timerEnable <= 0;
```

```
                    if (nextButton == 1'b1)   // If button is pressed, verify digit and proceed to next
state
                    begin
                      if (passwordDigit != 4'b0001)   // If incorrect digit, mark mismatch
                      begin
                        mismatch <= 1'b1;
                      end
                      passwordTransmit <= passwordDigit;
                      state <= CHECK;
                    end
                    else   // If button is not pressed stay in this state
                    begin
                      state <= DIGIT4;
                    end
                  end
                  CHECK:   // Verify password
                  begin
                    player1Transmit <= 1'b1; player2Transmit <= 1'b0;
                    redLED <= 1'b1; greenLED <= 1'b0;
                    digitTensTransmit <= digitTensValue; digitOnesTransmit <= digitOnesValue;
// Transmit timer values
                    timerEnable <= 0;
                    if (mismatch == 1'b0)   // If a mismatch was not found, correct password
                    begin
                      redLED <= 1'b0; greenLED <= 1'b1;
                    end
                    else   // If a mismatch was found, incorrect password
                    begin
                      state <= DIGIT1;
                    end
                    if (nextButton == 1'b1)   // Remain in CHECK state until another button press
                    begin
                      state <= GAME;
                    end
                    else
                    begin
                      state <= CHECK;
                    end
                  end
                  GAME:
                  begin
                    player1Transmit <= player1LoadSig; player2Transmit <= player2LoadSig;
                    redLED <= 1'b0; greenLED <= 1'b1;
```

```
                digitTensTransmit <= 0; digitOnesTransmit <= 0;
                timerEnable <= 1;
                if (timeout == 1'b0)
                begin
                  state <= END;
                end
                else
                begin
                  state <= GAME;
                end
              end
              END:
              begin
                player1Transmit <= 1'b1; player2Transmit <= 1'b0;
                redLED <= 1'b1; greenLED <= 1'b0;
                digitTensTransmit <= 0; digitOnesTransmit <= 0;
                timerEnable <= 0;
                if (nextButton == 1'b1)
                begin
                  state <= CHECK;
                end
                else
                begin
                  state <= END;
                end
              end
              default:   // Default case is DIGIT1
              begin
                passwordTransmit <= 4'b0000;
                player1Transmit <= 1'b0; player2Transmit <= 1'b0;
                redLED <= 1'b1; greenLED <= 1'b0;
                state <= DIGIT1;
              end
          endcase
        end
      end

endmodule
```

(System Top Module)

```
// ECE 6370
// Author: Sriram Pandiyan, 6101
// This is the top module for Lab3. It defines the  objects and connections necessary for the
system to function.


module Lab3_Ram_D(player1ButRaw, player1Disp,
                        player2Num, player2ButRaw, player2Disp,
                        passwordNum, passwordButRaw, passwordDisp, passwordRedLED,
passwordGreenLED,
                        sumResultDisp, sumResultRedLED, sumResultGreenLED,
                        tensDigitNum, onesDigitNum, tensDigitDisp, onesDigitDisp,
                        clock, reset);

        input clock, reset;   // 50 [MHz] on-board clock and button press for reset functionality
        input[3:0] player2Num, passwordNum;   // User input number and a password digit
        input player1ButRaw, player2ButRaw, passwordButRaw;   // Button presses for loading
numbers into the system
        input[3:0] tensDigitNum, onesDigitNum;   // User input timer numbers
        output[6:0] player1Disp, player2Disp, passwordDisp, sumResultDisp;   // 7-segment
display signals to visualize values
        output passwordRedLED, passwordGreenLED;   // Control signals for password
verification LEDs
        output sumResultRedLED, sumResultGreenLED;   // Control signals for summation
verification LEDs
        output[6:0] tensDigitDisp, onesDigitDisp;   // 7-segment display signals to visualize timer
        wire player2ButShaped, passwordButShaped;   // Processed button press signals
        wire player1RegLoad, player2RegLoad;   // Authorized register load signals
        wire[3:0] passwordLastDigit;   // Last password digit input
        wire[3:0] sumNum1, sumNum2, sumResult;   // Summation operands and result
        wire[3:0] tensDigitVal, onesDigitVal;   // Timer initial digits
        wire timerEnable, gameEnd;   // Timer enable and end signal
        wire[3:0] tensDigitCount, onesDigitCount;   // Timer countdown digits for visualization

        FourBitAdder FourBitAdder_1(sumNum1, sumNum2, sumResult);   // 4-bit adder object

        SevenSegDispDecoder SevenSegDispDecoder_Player1(sumNum1, player1Disp);   //
Player 1's display decoder object
        SevenSegDispDecoder SevenSegDispDecoder_Player2(sumNum2, player2Disp);   //
Player 2's display decoder object
        SevenSegDispDecoder SevenSegDispDecoder_Result(sumResult,sumResultDisp);   //
Summation result display decoder object
```

```
	SevenSegDispDecoder SevenSegDispDecoder_Password(passwordLastDigit, passwordDisp);   // Password digit display decoder object
	SevenSegDispDecoder SevenSegDispDecoder_TensDigit(tensDigitCount, tensDigitDisp);   // Game timer's tens digit display decoder object
	SevenSegDispDecoder SevenSegDispDecoder_OnesDigit(onesDigitCount, onesDigitDisp);   // Game timer's ones digit display decoder object

	VerificationLED VerificationLED_1(sumResult, sumResultRedLED, sumResultGreenLED);   // Summation result verification LED object

	LoadRegister LoadRegister_Player2(player2Num, sumNum2, player2RegLoad, clock, reset);   // Player 2's load register object

	AccessController AccessController_1(passwordNum, player1ButRaw, player2ButShaped, passwordButShaped,
						player1RegLoad, player2RegLoad, passwordLastDigit, passwordRedLED, passwordGreenLED,
						tensDigitNum, onesDigitNum, tensDigitVal, onesDigitVal, gameEnd, timerEnable,
						clock, reset);   // Access controller object

	ButtonShaper ButtonShaper_Player2(player2ButRaw, player2ButShaped, clock, reset);   // Player 2's button shaper object
	ButtonShaper ButtonShaper_Password(passwordButRaw, passwordButShaped, clock, reset);   // Access controller button shaper object

	FourBitRNG(player1RegLoad, sumNum1, clock, reset);   // 4-bit randon number generator object

	GameTimer GameTimer_1(timerEnable, tensDigitVal, onesDigitVal, gameEnd,
						tensDigitCount, onesDigitCount, clock, reset);   //  Game timer object

endmodule
```