FINAL PROJECT DESIGN DOCUMENT

tEAM: DIGITAL FORTRESS

tEAM members:

1. Rangeet Pan
2. Nikhita Singh Satish
3. Krishna Priya Peddi
4. Justin K Augustine
5. Naveen Milind Chalawadi

**CONTENTS**

1. **About the system**
2. **Features**
3. **Description of each module**
4. **Final Project Architecture**
5. **Screen captures of simulation**
6. **Board Images**
7. **Inference**
8. **Code Listing**
9. **About the system:**

As a part of this project, we have designed a Pattern-matching game with a one-time authentication to validate the players. There are two modes for the game: One Player mode and Two Player mode. The player(s) can select the mode by using a mode push button. The game for each mode is described below:

1. **One Player Mode**

The goal of the game is to score as many points as possible by matching with the desired pattern before all the 7-segments get blocked.

Initially, the player has to enter his 4- digit user ID followed by his 7-digit authentication sequence using 4 toggle switches for both user ID and password. The user ID is the last 4-digits of the PeopleSoft ID and the password contain the characters 0-9 and A-F. If the password entered is correct then the game is unlocked and the game starts. If the password entered is wrong the player gets two more chances to enter the correct password if not, the player can play the game in Guest mode. However, in this mode the player cannot get his max score.

After the authentication process the game starts. The player has to change the given sequence such that it matches the target sequence. There are two Random Number Generators (RNG) that generate each of the sequences. The first RNG gives the target sequence and displays on the left most 7-segment display. The second RNG which gives the initial sequence displays on the rightmost 7-segment. The player has to change this sequence by using the 7 toggle switches. Each toggle switch is mapped to each segment of the 7-segment display. The sequence jumps from one 7-segment to the next from right left at a specific interval of time. The player has to match with the target pattern before the sequence reaches the 7th 7-segment from the left i.e., HEX6. If the player is successful in matching the given sequence to the target sequence before the sequence reaches HEX6, his score increments by 1 and a new target sequence and input sequence is displayed. If the player doesn’t match his sequence with the target sequence, the 7th 7-segment gets blocked. Now, a new target sequence on HEX7 and input sequence displays on HEX0. The player has to match the moving sequence with the target sequence before the sequence reaches HEX5. If the player fails to do so, HEX5 gets blocked. This way the 7-segments get blocked gradually if the player fails to match with target sequence. The game ends when all the 7-segments are blocked.

The scoring module is designed such that, every time the player replicates the target sequence he gets a point. Based on the score the level of the game changes. It is as below:

1. score < 5: Level 0, the sequence jumps from one 7-segment to the next with an interval of 6 seconds.
2. 5 ≤ score < 10: Level 1, the sequence jumps from one 7-segment to the next with an interval of 5 seconds.
3. 10 ≤ score < 15: Level 2, the sequence jumps from one 7-segment to the next with an interval of 4 seconds.
4. 15 ≤ score < 20: Level 3, the sequence jumps from one 7-segment to the next with an interval of 3 seconds.

The maximum points a player can score is 20. The game ends when the score is 20 or all the 7-segments get blocked. The player can request a new game once the game ends.

1. **Two Player Mode**

The goal of the game is to score as many points as possible by replicating the desired pattern before all the 7-segments get blocked.

Initially, the player 1 has to enter his 4- digit user ID followed by his 7-digit authentication sequence using 4 toggle switches for both user ID and password. Once he is authenticated, player 2 has to enter his credentials. The user ID is the last 4-digits of the PeopleSoft ID and the password contain the characters 0-9 and A-F. If the password entered is correct then the game is unlocked and the game starts. If the password entered is wrong the player gets two more chances to enter the correct password if not, the players can play the game in Guest mode. However, in this mode the player cannot get his max score.

After the authentication process the game starts. The players have to set their respective 7 toggle switches such that it matches the target sequence. There are two Random Number Generators (RNG), one for each of the players that generate sequences. The target sequence is displayed on the left most 7-segment display for each player. The player has to change this sequence by using the 7 toggle switches. Each toggle switch is mapped to each segment of the 7-segment display. The sequence jumps from one 7-segment to the next from right left at a specific interval of time. The player has to match with the target pattern before the sequence jumps from one 7-segment to other for seven times. If the player is successful replicating the target sequence before the sequence jumps from one 7-segment to other for seven time, his score increments by 1 and a new target sequence is displayed. If the player doesn’t match his sequence with the target sequence, he gets only 6 iterations of the sequence jumping from one 7-segment to the next. Now, a new target sequence displays. The player has to match the moving sequence with the target sequence before the 6th iteration. If the player fails to do so, HEX5 gets blocked. This way the 7-segments get blocked gradually if the player fails to match with target sequence. The game ends when all the 7-segments are blocked.

The scoring module is designed such that, every time the player replicates the target sequence he gets a point. Based on the score the level of the game changes. It is as below:

1. score < 5: Level 0, the sequence jumps from one 7-segment to the next with an interval of 6 seconds.
2. 5 ≤ score < 10: Level 1, the sequence jumps from one 7-segment to the next with an interval of 5 seconds.
3. 10 ≤ score < 15: Level 2, the sequence jumps from one 7-segment to the next with an interval of 4 seconds.
4. 15 ≤ score < 20: Level 3, the sequence jumps from one 7-segment to the next with an interval of 3 seconds.

The maximum points a player can score is 20. The game ends when the score is 20 or all the 7-segments get blocked. The player can request a new game once the game ends.

1. **Features:**



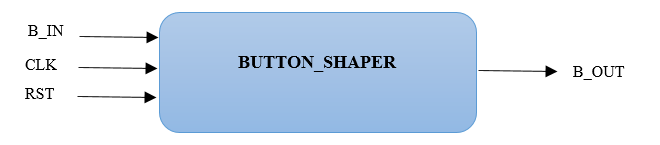
The system has the following modules.

1. Button shaper: button\_shaper.v
2. Mode select: mode\_select.v
3. Access control: ROM\_2AC.v
4. Single Access control: ROM\_AC.v
5. User ID ROM: ROM\_ID\_TOP.v
6. Password ROM: ROM2.v
7. Credential check: AC\_ROM\_USER.v
8. Output select: op\_select.v
9. Load register: LoadReg.v
10. Random number generator 1
11. LFSR module: LFSR1.v
12. Load register: LoadReg.v
13. Random number generator 2
14. LFSR module: LFSR2.v
15. Load register: LoadReg.v
16. Game timer
17. Milli-second pulse generator: LFSR\_1ms.v
18. 100 milli-second pulse generator: LFSR\_100ms.v
19. Variable timer: nSecPulseGenerator.v
20. Level selector: LevelSelector.v
21. One Player Game controller: game\_ctl.v
22. Two Player Game Controller: game\_ctl\_2pl.v
23. Score counter: SimpleCounter.v
24. Winner check: win\_checker.v
25. BCD score conversion: bin2bcd\_score.v
26. ASCII conversion: ASCII\_decoder.v
27. One player LCD Display: LCD\_display\_1pl.v
28. Two player LCD Display: LCD\_display.v
29. RAM\_TOP module: RAM\_TOP.v
30. RAM\_initalization module: RAM.v
31. RAM\_Controller module: RAM\_Controller.v
32. RAM for scores: RAM\_Score.v
33. Top module: Final\_project\_DigitalFortress.v
34. **Description of each module:**
35. **Button shaper: button\_shaper.v**

This system is used to shape the button pulse to give a *single cycle pulse*. The clock frequency is very high and when button is pressed, the signal is generated for several clock cycles. This is not desirable and hence to get a clean pulse for only one clock cycle we have designed the button shaper module.

It has the input signals – clk (clock signal), reset, b\_in (b\_in denotes the signal which is created when the button is pressed)- and output signal pulse. Since the push buttons are active-low, when the button is pressed b\_in is at logic low and it goes to logic high when the button is released. At the positive edge of the clock when b\_in is at logic low, pulse signal goes high for one clock cycle and then remains low until the button is pressed again. Thus, generating a single cycle pulse.

SYMBOL:



1. **Mode select: mode\_select.v**

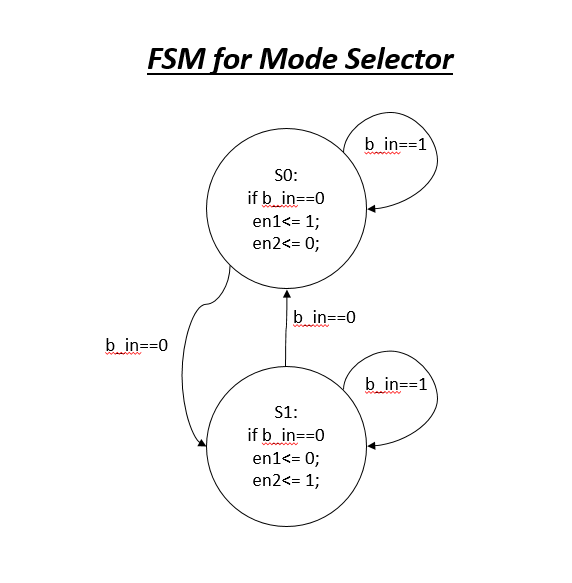
This module is designed to choose the mode between One player mode and Two players mode. It converts the mode select push button pulse into a continuous high enable signal. We have single cycle pulse as the input which is fanned out from a button shaper module as b\_in. It has two 1-bit output signals en1 and en2. The signal ‘en1’ when high the One player mode game is enabled and when ‘en2’ is at logic high the Two player mode is enabled.

Description of the FSM:

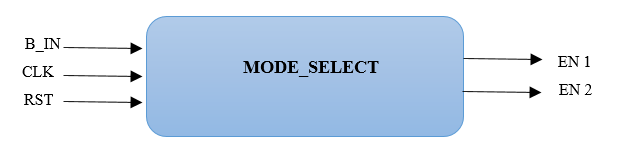
There are two states in the FSM. The process flow is as below:

1. Upon reset, both the enable signals – en1, en2- are set to logic low and the module is set to state 1.
2. In the first state the system waits for a button press. When a button press (single cycle pulse) is detected, ‘en1’ is set to high. Thereby, enabling One player mode of the game and the system goes to second state. If there is no button pressed then, the signal remains in first state.
3. Now when the system goes to second state, the system remains in this state until a button press is detected. Once the button is pressed, ‘en2’ is set to logic high and ‘en1’ is set to logic low and the system goes to first state.
4. This way the game toggles between the different modes.

FSM DIAGRAM:



SYMBOL:



1. **Access control: ROM\_2AC.v**

This module has ROM\_AC module instantiated thrice. First ROM\_AC module is used for one player mode, second and third are used for two player mode A and B respectively.

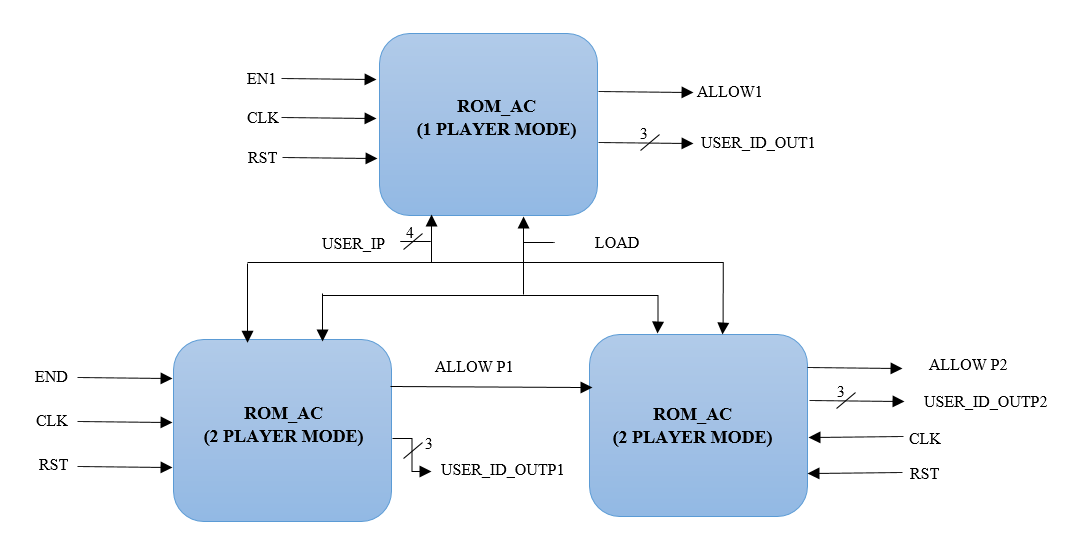
Inputs:

1. load- this is a single cycle output pulse given from the push button which is used to ress for every digit input.
2. En1- enable signal for one player mode
3. En2- enable signal for two player mode
4. User\_ip: 4 bit input for the user id / password

Output:

1. User\_id\_out1 – the 3 bit internal ID of the player in one player mode
2. User\_id\_outp1 – the 3 bit internal ID of the player A in two player mode
3. User\_id\_outp2 – the 3 bit internal ID of the player B in two player mode
4. Allow1 – authentication passed signal for one player mode
5. Allowp2 – authentication passed signal for two player mode

ARCHITECTURE:



1. **Single Access control: ROM\_AC.v**

This module consists of instantiation of ROM\_ID\_TOP, ROM2 and AC\_ROM\_USER modules. This is the access control module which sends out the allow signals for the game to start.

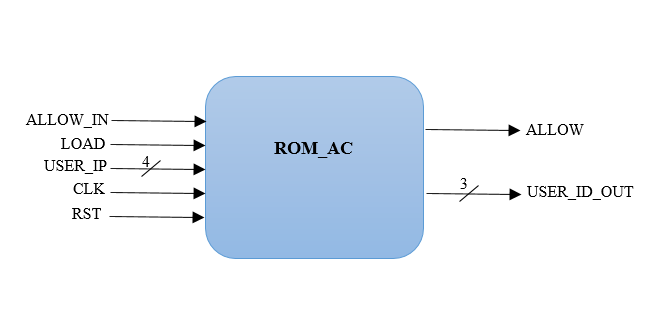
Input:

1. load – this is a push button which is used to enter the 4 bit user id / password of the user.
2. Allow\_in- is the enable signal for the AC\_ROM\_USER module

Output:

1. user\_id\_out is the 3 bit internal ID of the player
2. Allow : it is the signal which goes HIGH when the password matches or if the player enters the guest mode.

SYMBOL:



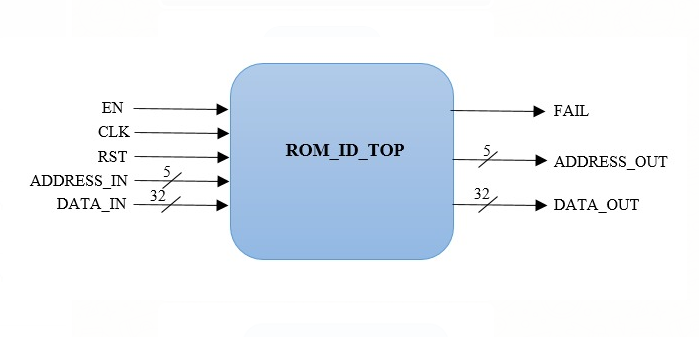
1. **User ID ROM: ROM\_ID\_TOP.v**

This module contains both Rom\_User\_initialization and Rom\_User\_Controller modules.

Inputs: The enable signal “en” which would be enabled after the user enters his/her four digits of the user id.

Outputs: The data\_out is the 32 bit output which contains the data at each address in RAM.

SYMBOL:



1. **Rom\_User\_initialization: ROM\_KP.v**

This module is generated from the *mif* file we create by entering the data values. Mif file is an ASCII text file which defines the initial values/data of the memory block or address. We specify the number of words and the word size(width) of the data in the mif file. Using the mif file, we can create the Rom\_user module which takes the “address” and clock as inputs and gives out the data present in that address as “q”.

*How is the user ID stored in ROM*:

The address size of Rom\_user is 5 bits whereas the user id stored in each address is of 32 bits. We use the 32 bits as following.

**User ID data in ROM**

|  |  |  |  |
| --- | --- | --- | --- |
| Valid bit | Internal ID | Redundant bits | User ID of players |

[One Bit 31] [three bits 30- 28] [27-16] [ 15-0]

SYMBOL:



1. **Rom\_User\_Controller: ROM\_Controller\_ID.v**

This module starts with address 0 whenever a user enters his/her user ID. Then the controller module keeps sending the address till its enable signal is high. This enable signal is usually given from the access control module which makes the enable high whenever an entered user id matches with any of the data value from the ROM.

The following are the input and the output signals to this module:

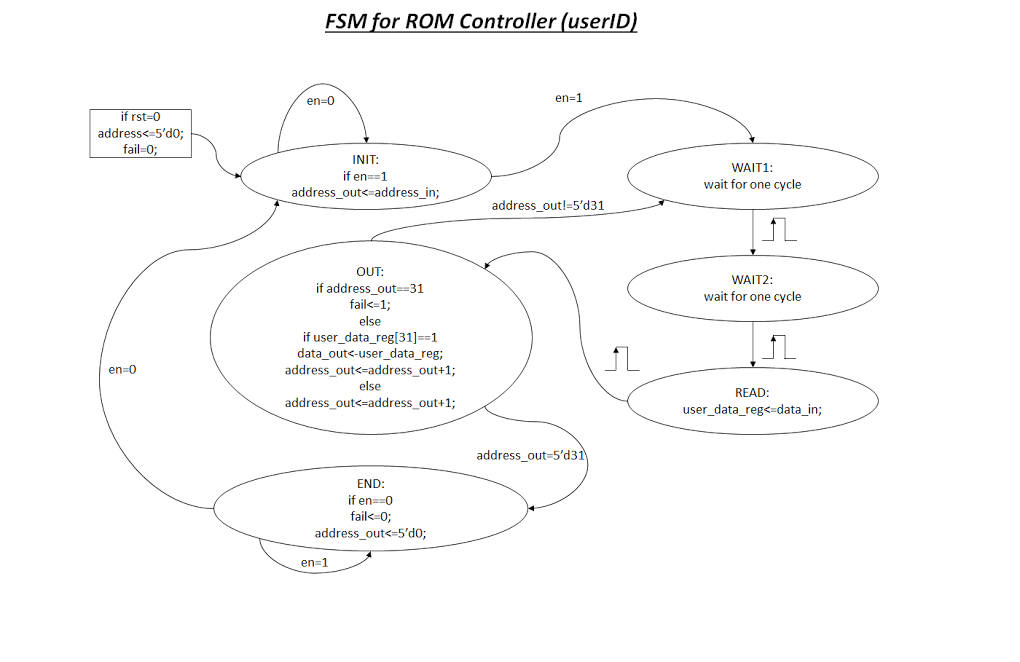
Inputs:

1. en: enable signal which is HIGH when the user has entered his 4 digit user ID
2. Address\_in: is the 5 bit address which is assigned to zero whenever a user ID is entered
3. Data\_in: is the 32 bit size data which comes from the rom\_user\_initilization module

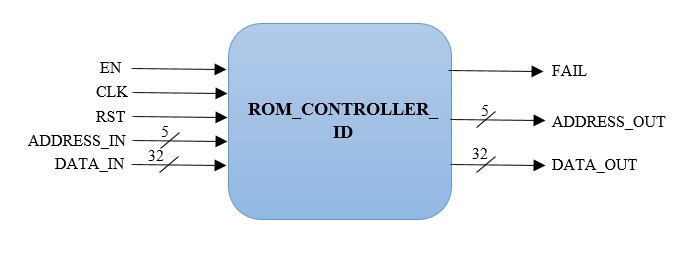
Outputs:

1. address\_out:5 bit address out gives out the address that the rom should currently point to
2. Data\_out: 32 bit data out has the internal id and the user id at the corresponding address sent to Rom\_user\_initilization
3. Fail: It is the flag sent to the top module telling that the user id entered is not available in the Rom module and thus shift to guest mode.

FSM DIAGRAM:



SYMBOL:



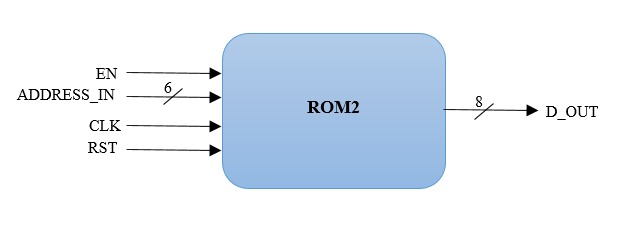
1. **Password ROM: ROM2.v**

This module consists of the instances of Rom\_password\_intialization and Rom\_password\_controller.

Input: address\_in of size 6 bits which takes the address associated with the internal ID of the user

Output: d\_out – this is each password digit that is extracted from ROM.

SYMBOL:



1. **Rom\_password\_initialization :Rom\_initialization\_file.v**

We specify the number of words and the word size(width) of the data in the mif file for the password ROM. Using the mif file, we can create the Rom\_password module which takes the “address” and clock as inputs and gives out the data present in that address as “q”.

*How is the password stored in ROM*: The Address in the ROM is organized as Address [2:0] representing the internal ID of the user. This internal ID is given to this module by the User\_ID ROM which does an exhaustive search to fetch the player’s internal ID. And then the Address [3:5] which has the count of the password digits. The first password digit is stored in the Address [3:5] of value 000, second password digit at 001 and so on till the last password bit which is at 111. After getting the internal ID, it goes to that particular address of the player. The password has a total of 7 digits which is stored in each address linked to the internal ID.

The address size is 6 bits and the password digit(data) size in each address is 8 bits. The Rom controller for the password block requests for the password bits continuously to get all the digits of the password of that user.

**Address**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0 | 0 | 1 | 0 | 1 | 1 |

Internal ID K th password digit

|  |  |
| --- | --- |
| Address | Password value |
| 000000 | 10001110 |
| 000001 | 10001100 |
| 000010 | 10001110 |
| 000011 | 10000111 |
| …. | …. |
| …. | …. |
| 000111 | 11110101 |
| 001000 | 10000111 |
| 001001 | 10001111 |
| 001010 | 10001011 |
| …. | …. |
| 001111 | 11110000 |

The Rom\_password module has the address size of 6 bits where address 0-2 bits denote the internal ID of the player that comes from the User Rom module. The bits address 3-5 denotes the kth digit of the password.

The inputs and outputs of this module are

Input: **address**- refers to the address in the memory file and clock

Output: **q**- data stored in the address

SYMBOL:



1. **Rom\_password\_controller**: **RomController.v**

Using this Rom controller module, we control the accessing of the data from the ROM. This module is enabled only when the user has finished entering his user ID. This module sends the address values and takes the data values to the *Rom\_password\_intialization* module. The address keeps incrementing till the user ID has matched or the end of the rom addresses are reached.

The inputs and outputs are as following:

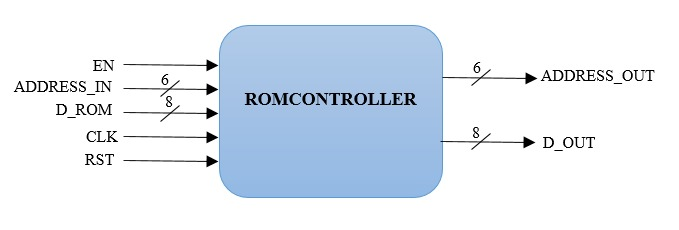
Inputs:

1. en- enables the controller module when the user has finished entering his/her user ID.
2. [5:0] address\_in – the module takes a 6-bit address input which is the start index of where the password should be fetched from.
3. [7:0] d\_rom – the module gets the data values from the *Rom\_password\_intialization* file when it sends address\_out to the initialization module.

Outputs:

* + - 1. address\_out [7:0] – the module sends address\_out to the *Rom\_password\_intialization* module to get the data from that address.
      2. d\_out [7:0] – the module sends the output to the ROM user top module with the password digits which are stored in ROM.

SYMBOL:



1. **Credential check: AC\_ROM\_USER.v**

The Access Control module performs Authentication. It has 6 inputs and 5 outputs. The input output descriptions are as shown below,

Inputs:

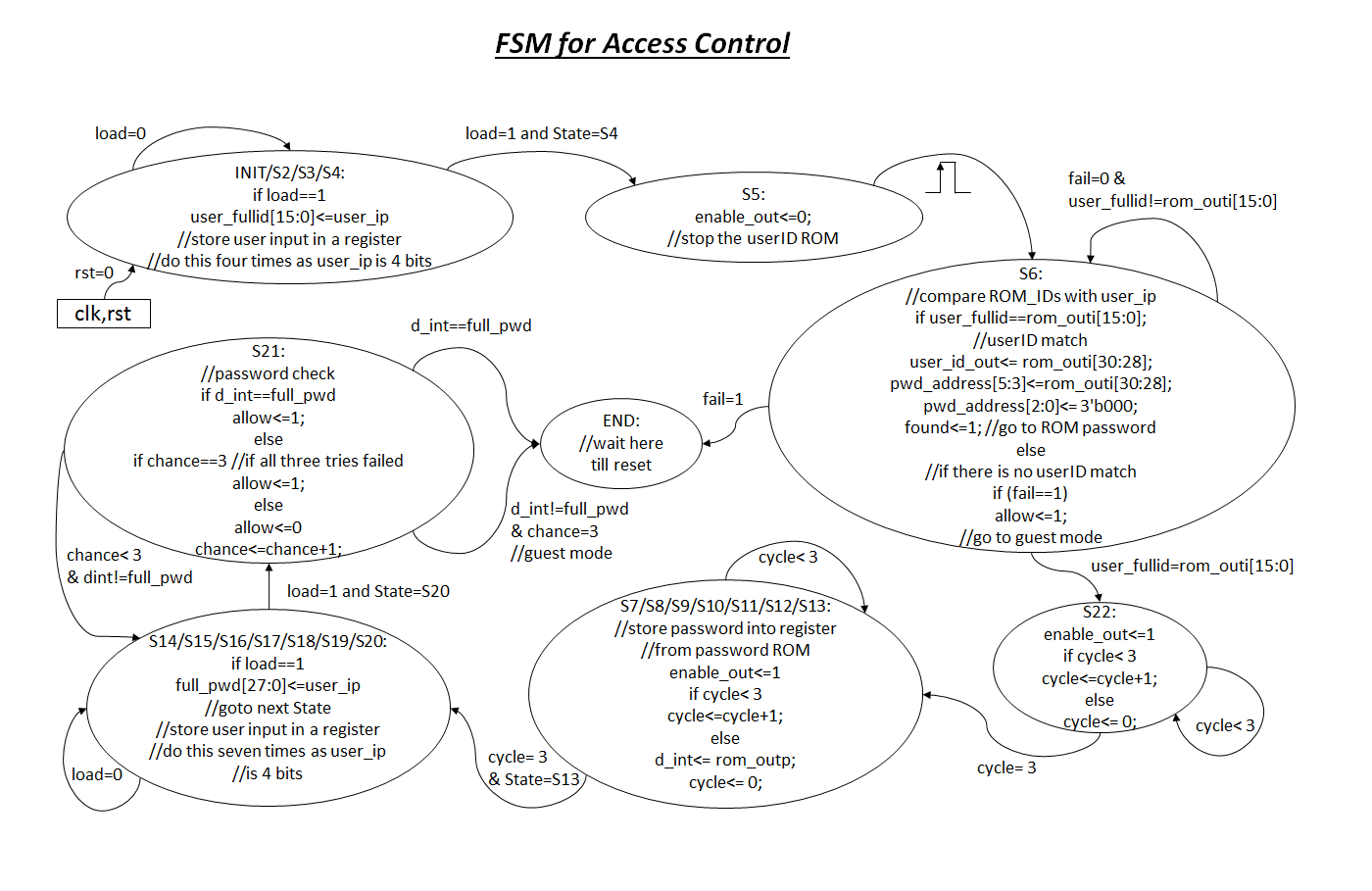
1. user\_ip: 4-toggle switches i.e., 4-bit input for entering user ID and Password.
2. load: To load the input to the Access Control.
3. allow\_in: Enable signal for the Access Control. For 1-player mode allow\_in is start game signal (from Mode select module). For 2-player mode, we have 2 access control modules. for the first access control the enable is start game from mode select module and for the second access control module, the enable signal is the output allow from the first access control.
4. fail: input signal to access control to indicating that the search in ROM failed.
5. rom\_out\_id: user\_id stored in ROM address address\_in.
6. rom\_out\_pwd: password stored in ROM address address\_in.

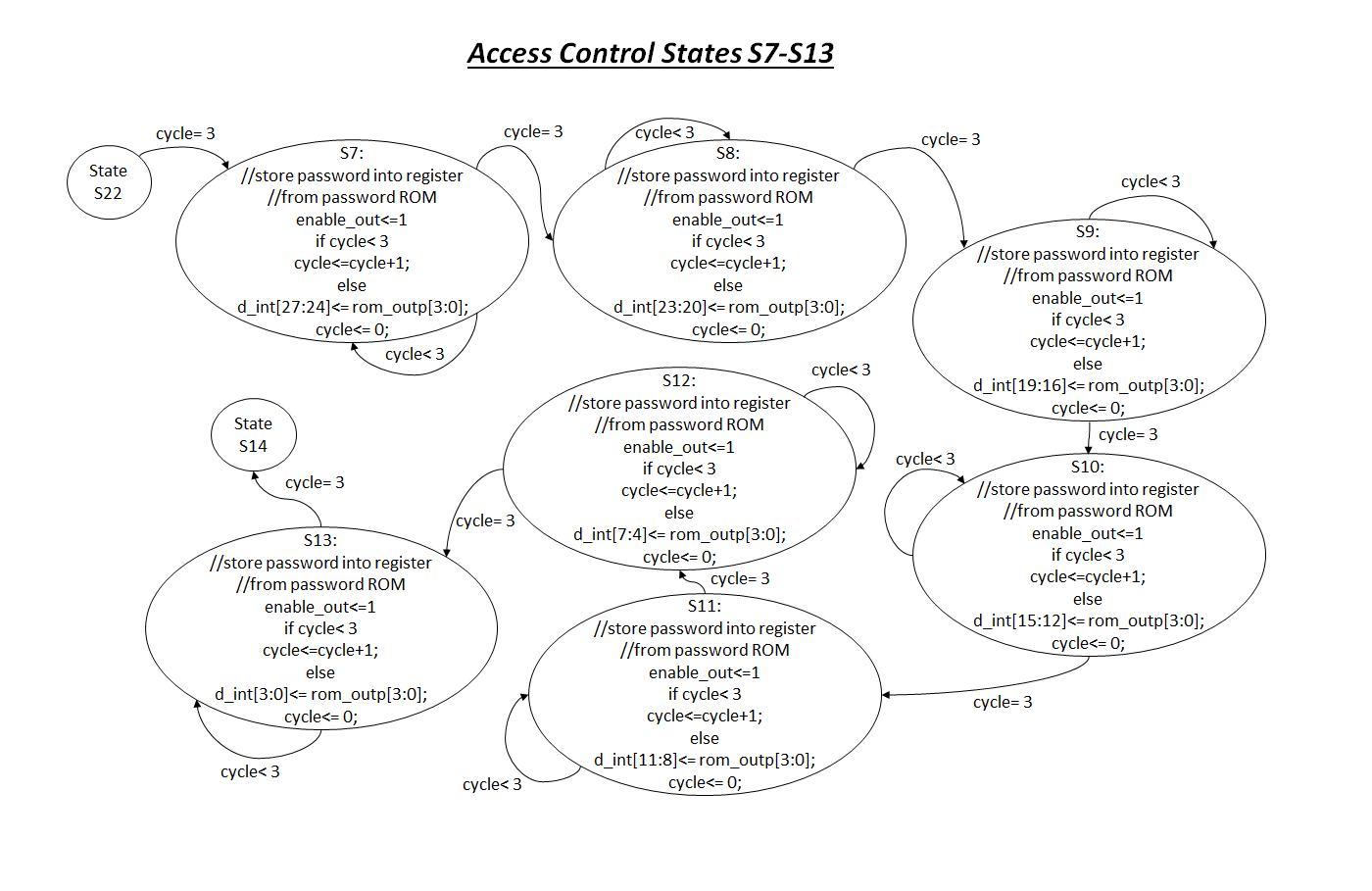
Outputs:

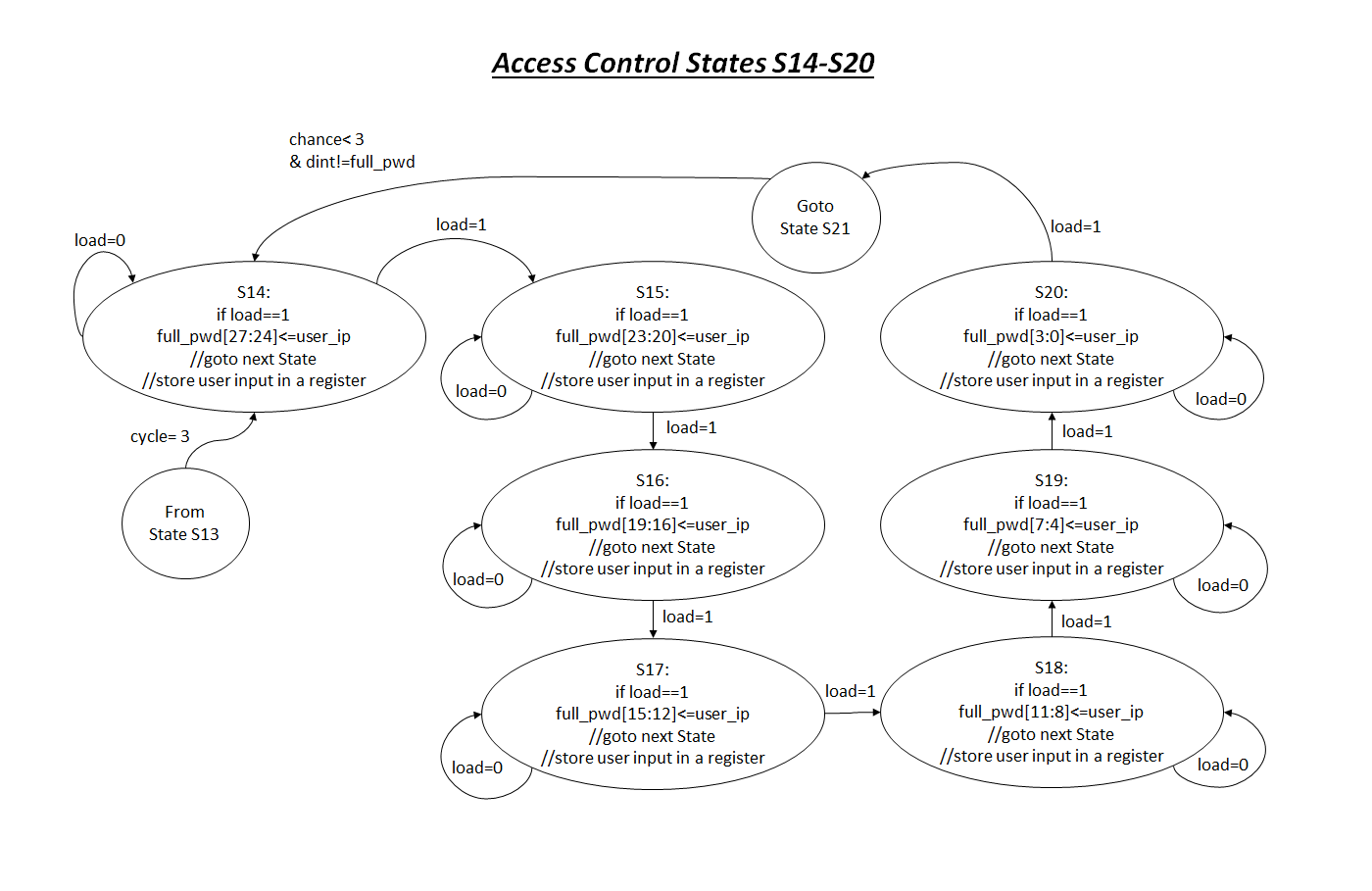
1. enable: enable signal to activate and access user ID ROM.
2. found: enable signal to activate and access Password ROM.
3. address\_in: if enable=1, to select a particular address in ROM from where the userID must be read.
   1. if found=1, to select a particular address in ROM from where the password must be read.
4. user\_id: The internal ID of the Player based on his actual User\_ID
5. allow: The output signal that allows the players to play the game.

The access control performs the following steps in sequence

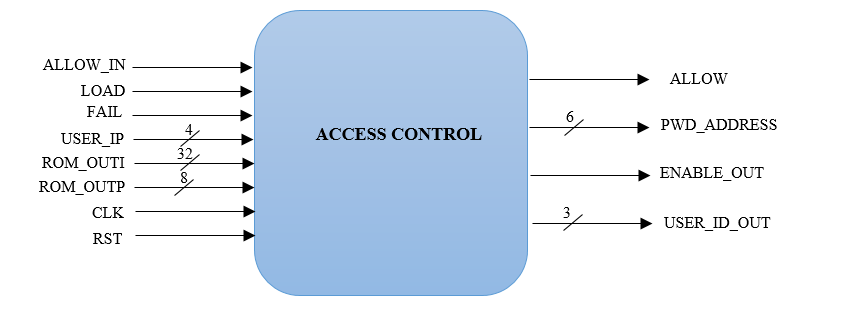
1. If allow\_in is high, the user enter their 4-digit user ID using 4 toggle switches and a load button.
2. The player must enter his 4-digit user-ID using 4-toggle switches and load button.
3. The access control activates user\_id ROM and searches matching user\_ID in Rom.
4. If there is a match, the internal ID (user\_id) of the player output to the password ROM. used\_id ROM is disabled and found signal is made high. found signal enables the password ROM.
5. The player now enters the 7-digit password using 4 toggle switches and a load button.
6. The access control reads the Password from password ROM using the internal ID (user\_id) of the player, and compares it with the password entered by the player.
7. If there is a match then the player is allowed to play the game. Otherwise, the player is given 2 more chances to enter the password. If the player fails three times, the game goes into guest mode, where the player is allowed to play the game but his scores will not be recorded.

FSM DIAGRAM:





SYMBOL:

****

1. **Output select: op\_select.v**

Since we have two modes of game, this module was designed. The module selects whether the one player mode game’s output has to be displayed or if the two players mode game’s output has to be selected based on which enable signal is at logic high. If ‘en1’ signal is at logic high, one player mode’s output is displayed on the corresponding 7-segments and LCD. If ‘en2’ signal is at logic high, two players mode’s output is displayed on the 7-segments and LCD. When none of the enable signals are high, the 7-segments and LCD display is designed to be blank.

The input and output signals are defined as below:

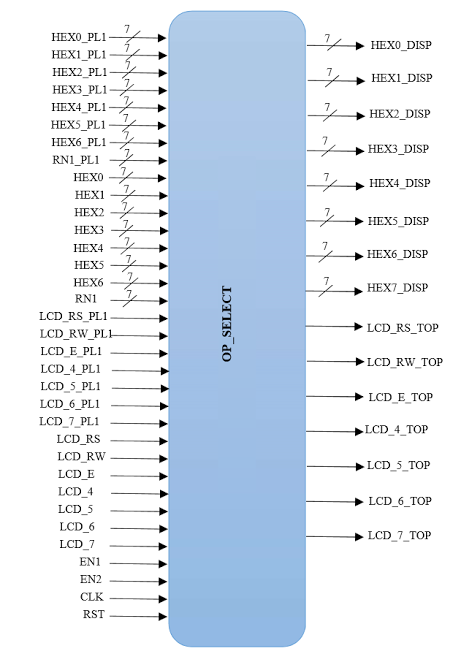
Input:

1. hex0\_pl1, hex1\_pl1, hex2\_pl1, hex3\_pl1, hex4\_pl1, hex5\_pl1, hex6\_pl1, RN1\_pl1: These are the 7-bit output signals from the one player mode game controller which are given here as input. hex0\_pl1 to hex6\_pl1 are to displayed on HEX0 to HEX6 7-segment display on the board respectively. They display the result of the 7-bit input sequence XORed with the player’s 7-bit input. RN1\_pl1 is the 7-bit target sequence which has to be displayed on 8th 7-segment i.e., HEX7.
2. hex0, hex1, hex2, RN2, hex4, hex5, hex6, RN1: These are the 7-bit output signals from the two player mode game controller which are given as input here. hex0 to hex2 are to displayed on HEX0 to HEX2 7-segment display on the board respectively. They display the player’s 7-bit input given by Player 2. RN2 is the 7-bit target sequence which the Player 2 has to replicate. hex4 to hex6 are to displayed on HEX4 to HEX6 7-segment display on the board respectively. They display the 7-bit input given by Player 1. RN1 is the 7-bit target sequence which the Player 1 has to replicate.
3. en1: It is 1-bit enable signal for one player mode. It is high when the One-player mode has been selected.
4. en2: It is 1-bit enable signal for two players mode. It is high when the Two-players mode has been selected.
5. lcd\_rs\_pl1, lcd\_rw\_pl1, lcd\_e\_pl1, lcd\_4\_pl1, lcd\_5\_pl1, lcd\_6\_pl1, lcd\_7\_pl1: These are the outputs from LCD display module for One player mode.
6. lcd\_rs, lcd\_rw, lcd\_e, lcd\_4, lcd\_5, lcd\_6, lcd\_7: These are the outputs from LCD display module for Two players mode.
7. rst: This is the active low reset signal
8. clk: This is clock input

Output:

1. hex0\_disp, hex1\_disp, hex2\_disp, hex3\_disp, hex4\_disp, hex5\_disp, hex6\_disp, hex7\_disp: These are 7-bit output signals which are given to 7-segment displays HEX0 to HEX7 respectively.
2. lcd\_rs\_top: This is one bit output signal given to LCD\_RS pin of the LCD display. It performs LCD Command/Data Select, 0 = Command, 1 = Data
3. lcd\_rw\_top: This is one bit output signal given to LCD\_RW pin of the LCD display. It performs LCD Read/Write Select, 0 = Write, 1 = Read.
4. lcd\_e\_top: This is one bit output signal given to LCD\_EN pin of the LCD display. It enables the LCD when set to high.
5. lcd\_4\_top, lcd\_5\_top, lcd\_6\_top, lcd\_7\_top: They are single bit data outputs given to LCD\_DATA[4], LCD\_DATA[5], LCD\_DATA[6], LCD\_DATA[7] of the LCD module.

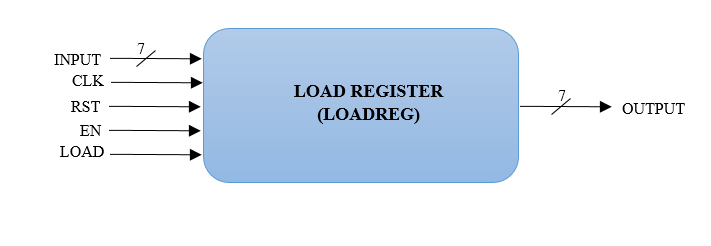
SYMBOL:



1. **Load register: LoadReg.v**

This module is deigned to load the 7-bit input onto the 7-bit output signal only when the enable signal (en) is at logic high and the request signal (Load) is at logic high. It has a 7-bit input signal ‘Input’ , clk (clock signal), reset, load and enable signal ‘en’. The output signal is Output which is of 7-bits. When the reset (active low) signal is at logic low the system sets the output signal to 1111111.

SYMBOL:



1. **Random number generator 1**

It is a combination of a Liner Feedback Shift Register and a Load Register.It is designed to generate a 7-bit sequence in a random fashion.

1. **LFSR module: LFSR1.v**

This module is a Linear Feedback Shift Register. LFSR is an *n*-bit shift register which pseudo-randomly scrolls between 2n-1 values, but does it *very quickly* because there is minimal combinational logic involved.  Once it reaches its final state, it will traverse the sequence exactly as before. Here we have used 16bit LFSR which has One-to-many feedback structure and the gates are XOR gates. We have used the maximum allowable bit size in order to increase the randomness.

Advantages of LFSR include high speeds and little logic to implement.

The logic is when the system is clocked, bits that are not taps are shifted one position to the right unchanged. The taps, on the other hand, are XOR'd with the output bit before they are stored in the next position. The new output bit is the next input bit. The effect of this is that when the output bit is zero all the bits in the register shift to the right unchanged, and the input bit becomes zero. When the output bit is one, the bits in the tap positions all flip (if they are 0, they become 1, and if they are 1, they become 0), and then the entire register is shifted to the right and the input bit becomes 1.

SYMBOL:



1. **Load register LoadReg.v**

The working of this module is same as the one describe earlier. The input to the system is 7-bits. 7-bits are selected in random from the 16- bit output of the LFSR and given as input to the load register.

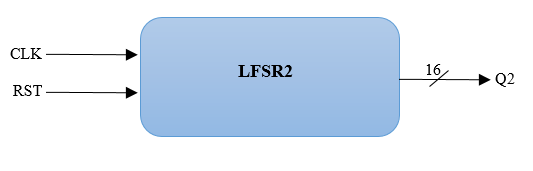
1. **Random number generator 2**

It is a combination of a Liner Feedback Shift Register and a Load Register.It is designed to generate a 7-bit sequence in a random fashion.

1. **LFSR module: LFSR2.v**

The working of this module is same as the one describe for LFSR1.v. The only difference is in the bits selected for feedback are different. This done to increase the randomness.

SYMBOL:

****

1. **Load register LoadReg.v**

The working of this module is same as the one describe earlier. The input to the system is 7-bits. 7-bits are selected in random from the 16- bit output of the LFSR and given as input to the load register.

1. **Game timer**
2. **Milli-second pulse generator: LFSR\_1ms.v**

This module is designed to generate a single cycle pulse for every 50,000 cycles of the 50 MHZ clock i.e., for every milli-second. A Linear Feedback Shift Register is used. This register is used as are aim is to count for 50,000 cycles of clock and we are not concerned with the value of the counter. The counter is runs only when the enable signal is high. After 50,000 clock cycles, a single cycle pulse is driven at the output Out1. If the enable is at logic low, the LFSR value remains unchanged. Upon reset, the LFSR value is initialized to 16'b1111111111111111 and the output is set to logic low.

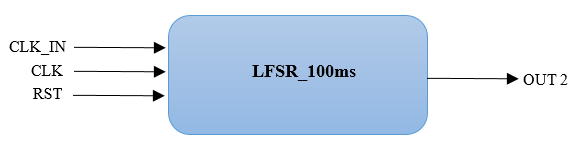
SYMBOL:

****

1. **100 milli-second pulse generator: LFSR\_100ms.v**

This module is used to generate a single cycle pulse every 100 milli-seconds. The output from LFSR\_1ms module above is given to this module as Clk\_in as an input. We use a 7-bit register – count1 - which is initialized to 0. When Clk\_in is at logic high, the count value is incremented by one. When the count reaches 100, a single cycle pulse is generated at the output, Out2 and the count value is reset to 0. This process keeps repeating.

SYMBOL:

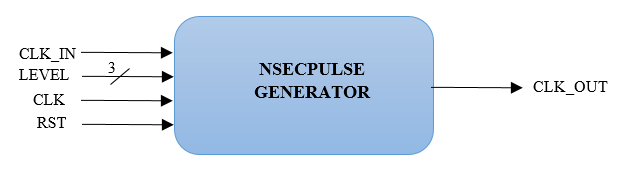
****

1. **Variable timer: nSecPulseGenerator.v**

This module is used to generate a single cycle pulse at fixed interval of time. Based on the 3-bit level as input, there are different count values. The output from LFSR\_100ms module above is given to this module as Clk\_in as an input. We use registers for different count values which is initialized to 0. When Clk\_in is at logic high, the count value is incremented by one.

1. Level 0: For this level count1=60. Hence, we get a single cycle pulse every 6 seconds at the output signal Clk\_out. So, the sequence jumps from one 7-segment to the next with an interval of 6 seconds.
2. Level 1: For this level count2=50. Hence, we get a single cycle pulse every 5 seconds at the output signal Clk\_out. So, the sequence jumps from one 7-segment to the next with an interval of 5 seconds.
3. Level 2: For this level count3=40. Hence, we get a single cycle pulse every 4 seconds at the output signal Clk\_out. So, the sequence jumps from one 7-segment to the next with an interval of 4 seconds.
4. Level 3: For this level count4=30. Hence, we get a single cycle pulse every 3 seconds at the output signal Clk\_out. So, the sequence jumps from one 7-segment to the next with an interval of 3 seconds.

SYMBOL:



1. **Level selector: LevelSelector.v**

This module sets up different levels for players according to their scores. As the scores increases, it increases the difficulty level of the game. Initially the game starts at level 0 and if score>5, the level is shifted to level1 where the rate of generating the random generator increases, thus increasing the difficulty of the player. The levels are designed as follows:

|  |  |
| --- | --- |
| Score | Level |
| score<5 | 0 |
| 5 ≤ score < 10 | 1 |
| 10 ≤ score < 15 | 2 |
| 15 ≤ score < 20 | 3 |

Inputs:

Score: this is a 6-bit score value of the player

Output:

Level: this is the 3 bit output level selected according to the score input.

1. **One Player Game controller: game\_ctl.v**

This is the most important module of the game. It checks if the player has generated the require pattern and shifts/jumps the sequence from one 7-segment to the next ( from Right to Left). Here we have count register which helps in determining the total number of 7-segments to shift in each round. It is enabled when ‘en1’ mode select module is set to logic high. The input and output signals are defined below:

Inputs:

1. a\_ip: This is the 7-bit sequence which has been loaded by player a
2. trgt\_num: This is the 7-bit random number generated which is loaded from the load register described above that the player needs to replicate. It is the target pattern that has to be replicated to win a point.
3. new\_num: This is the 7-bit random number generated which is loaded from the load register described above that the player needs to alter to create the target pattern.
4. timer\_ip: This is a single cycle pulse coming from the nSecPulseGenerator. This helps in jumping the output from one 7-segment to another at a specific time interval.
5. start\_gm: This is the enable signal which is generated in mode select.
6. new\_gm: This signal is used to restart the game after the game ends.
7. alw: This is the allow signal generated from the access control.
8. reset: This is the reset signal
9. clk: This is clock signal

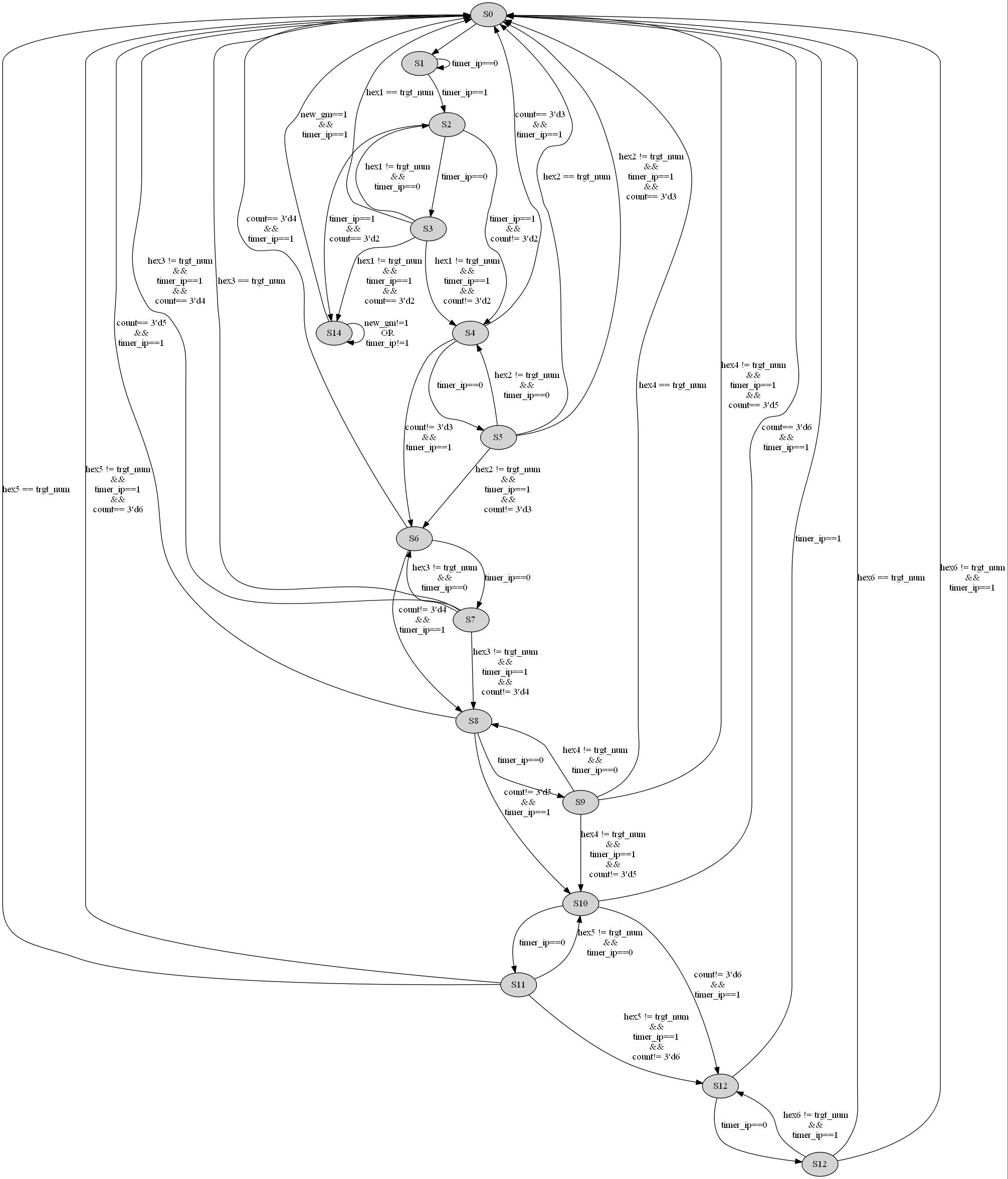
Output:

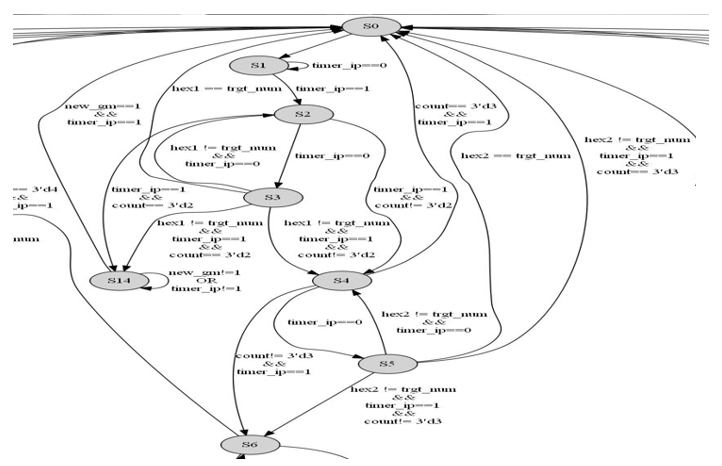
1. hex0, hex1, hex2, hex3, hex4, hex5, hex6: hex0 to hex6 are to be displayed on HEX0 to HEX6 7-segment display on the board respectively. They display the result of the 7-bit input sequence XORed with the player’s 7-bit input.
2. cout: This is a single cycle pulse which is generated when a\_ip ^ new\_num = trgt\_num. This is further given to score counter to increment the score.
3. req\_num: This signal is high when a\_ip ^ new\_num = trgt\_num or when the sequence pattern has jumped to the last un-blocked 7-segment display. It is sent to the load register of our random numbers and gets a new target sequence and input sequence.
4. end\_gm: This signal is high when all the 7-segments are blocked.

Description of FSM:

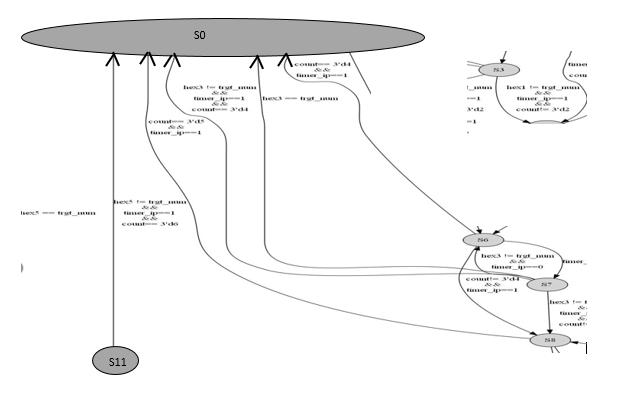
1. When the user is authenticated(alw=1) and the one player mode is elected (en=1) the system is in state-0. In state-0, a new target sequence and input sequence is requested to be loaded from the random number generator and the system goes to the next state.
2. In state-1, the req\_num signal is set to logic low and the player’s input and the input sequence are XORed and displayed on HEX0. The module waits in this state until a timer\_ip pulse is detected. When the timer\_ip is high, the system goes to the next state.
3. In state-2, the previous value of hex0 is set to 7’d127. This basically makes the previous 7-segment output to blank and the player’s input and the input sequence are XORed and displayed on HEX1 7-segment display. If timer\_ip is high then the system checks if the succeeding(third) 7-segment display has been blocked or not. If blocked, then it sets end\_gm signal to high denoting the end of the game and the system goes to state-14. If the third 7-segment display is not blocked, then the system goes to state-4. If the timer\_ip pulse is not set to high, the system goes to the next state.
4. In state-3, the system checks if the player has created the required target sequence. If so, then an increment pulse is generated i.e., cout is set to high and the system is set to state-0 where a new sequence is requested. If the player fails to match then the system checks if timer\_ip is high. If so, then the system checks if the succeeding(third) 7-segment display has been blocked or not. If blocked, then it sets end\_gm signal to high denoting the end of the game and the system goes to state-14. If the third 7-segment display is not blocked, then the system goes to next state. If the timer\_ip pulse is not set to high, the system goes to the previous state and calculates a\_ip ^ new\_num and the process continues.
5. In state-4, the previous value of hex1 is set to 7’d127. This basically makes the previous 7-segment output to blank and the player’s input and the input sequence are XORed and displayed on HEX2 7-segment display. If timer\_ip is high then the system checks if the succeeding(fourth) 7-segment display has been blocked or not. If blocked, then it goes to state 0. If the fourth 7-segment display is not blocked, then the system goes to state-6. If the timer\_ip pulse is not set to high, the system goes to the next state i.e., the sequence jumps to fourth 7-segment.
6. In state-5, the system checks if the player has created the required target sequence. If so, then an increment pulse is generated i.e., cout is set to high and the system is set to state-0 where a new sequence is requested. If the player fails to match then the system checks if timer\_ip is high. If so, then the system checks if the succeeding(fourth) 7-segment display has been blocked or not. If blocked, then it goes to state 0. If the fourth 7-segment display is not blocked, then the system goes to next state. If the timer\_ip pulse is not set to high, the system goes to the previous state and calculates a\_ip ^ new\_num and the process continues.
7. state-6 to state-13, the process repeats similar to that of state-4 and state-5 for the fifth, sixth and seventh 7-segment displays.
8. Once the sequence reaches the 7th 7-segment, the system goes to state-0 and the process repeats. If the player fails to match the pattern even after the sequence jumps through all the seven 7-segments, the 7th 7-segment is blocked.
9. In state-14, the system checks if a new game is requested. If so, then it unblocks all the 7-segment, sets end\_gm signal to logic lows and goes to state-0. If not, it waits in this state continuously.

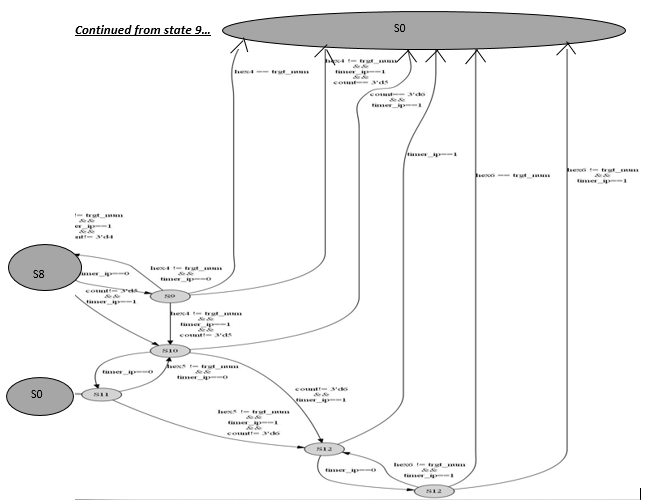
FSM DIAGRAM:



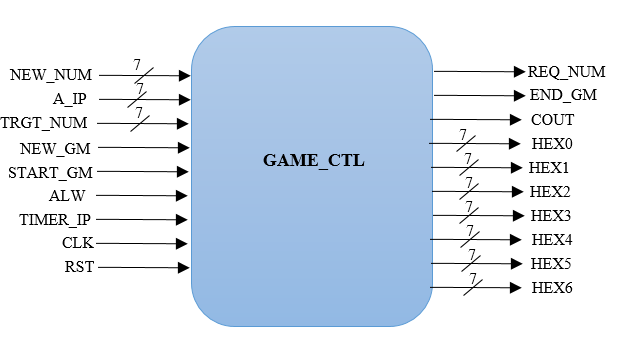
***FSM DIAGRAM ZOOMED IN: State 1-6***







SYMBOL:



1. **Two Player Game Controller: game\_ctl\_2pl.v**

This is the most important module of the game. It checks if the player has generated the require pattern and shifts/jumps the sequence from one 7-segment to the next ( from Right to Left). Here we have count register which helps in determining the total number of 7-segments to shift in each round. It starts working when ‘en2’ of mode select module is set to high. The input and output signals are defined below:

Inputs:

1. a\_ip: This is the 7-bit sequence which has been loaded by player.
2. trgt\_num: This is the 7-bit random number generated which is loaded from the load register described above that the player needs to replicate. It is the target pattern that has to be replicated to win a point.
3. timer\_ip: This is a single cycle pulse coming from the nSecPulseGenerator module. This helps in jumping the output from one 7-segment to another at a specific time interval.
4. start\_gm: This is the enable signal which is generated in mode select.
5. new\_gm: This signal is used to restart the game after the game ends.
6. alw: This is the allow signal generated from the access control.
7. reset: This is the reset signal
8. clk: This is clock signal

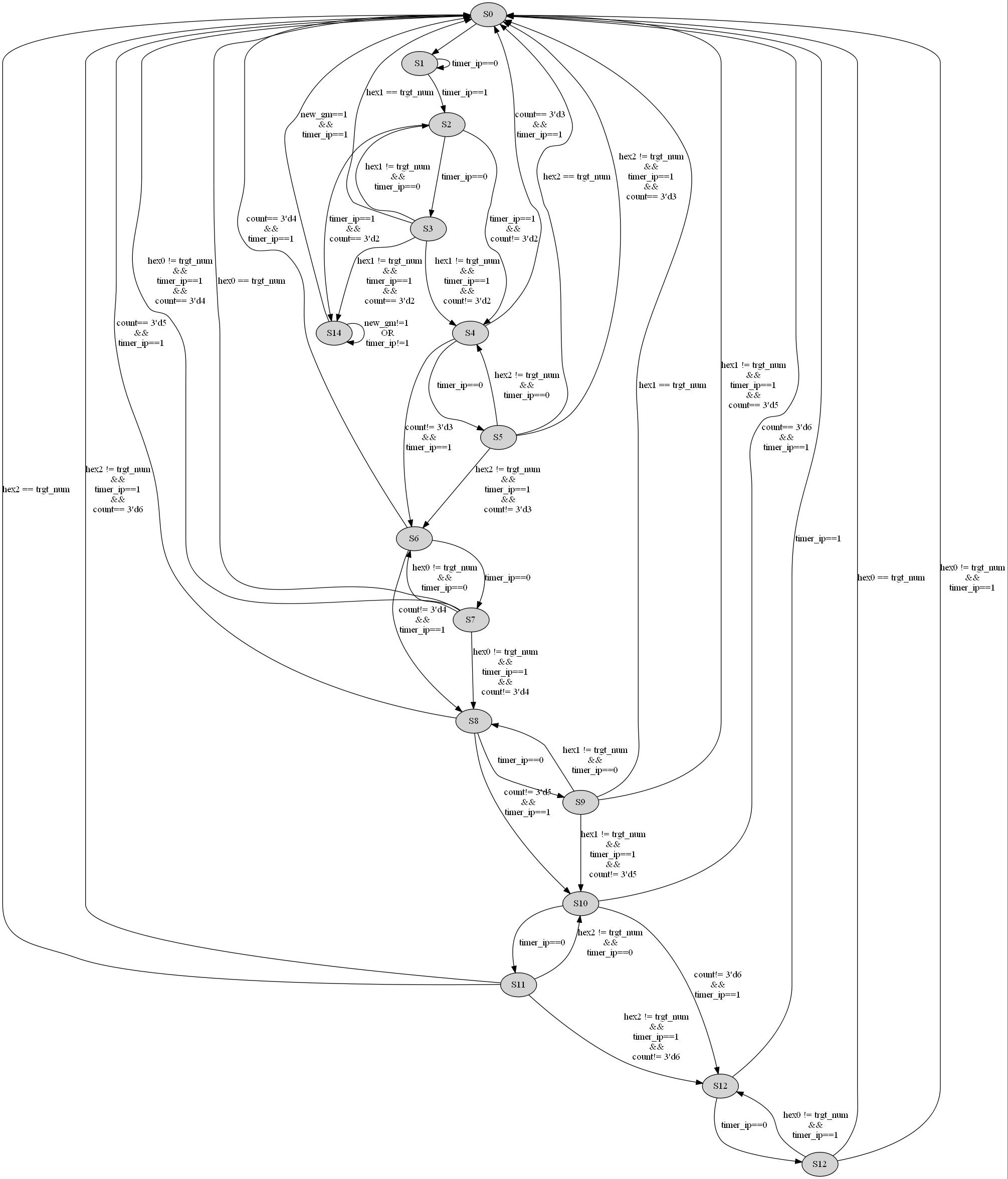
Output:

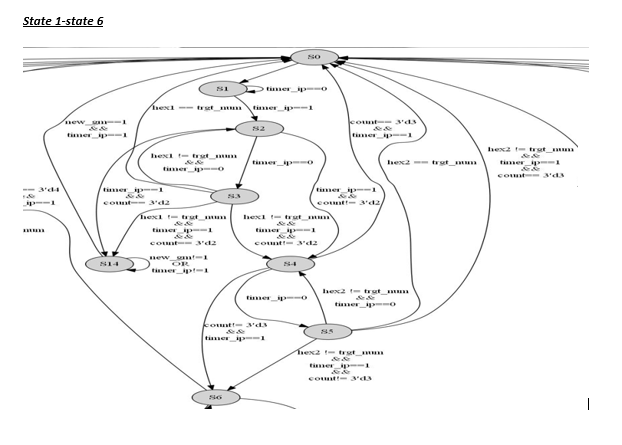
1. hex0, hex1, hex2: hex0 to hex2 are to be displayed on the first three 7-segment display of each player. They display the result of the 7-bit sequence of the player’s 7-bit input.
2. cout: This is a single cycle pulse which is generated when a\_ip ^ new\_num = trgt\_num. This is further given to score counter to increment the score.
3. req\_num: This signal is high when a\_ip ^ new\_num = trgt\_num or when the sequence pattern has jumped to the last un-blocked 7-segment display. It is sent to the load register of our random numbers and gets a new target sequence and input sequence.
4. end\_gm: This signal is high when all the 7-segments are blocked.

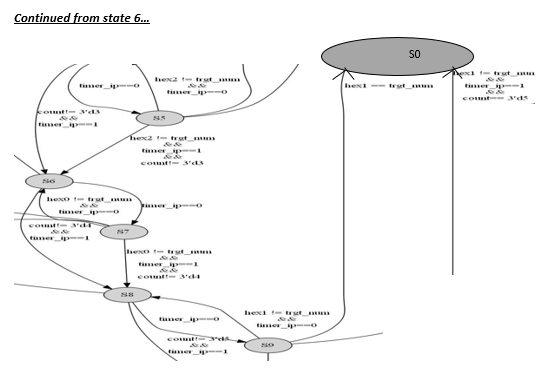
Description of FSM:

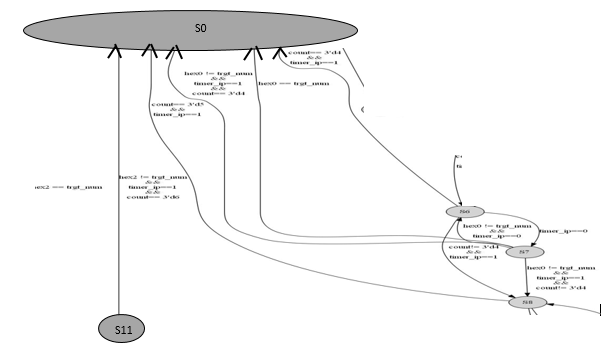
1. When the user is authenticated(alw=1) and the one player mode is elected (en=1) the system is in state-0. In state-0, a new target sequence and input sequence is requested to be loaded from the random number generator and the system goes to the next state.
2. In state-1, the req\_num signal is set to logic low and the player’s input sequence is displayed on the first 7-segment(from right) of each player. The module waits in this state until a timer\_ip pulse is detected. When the timer\_ip is high, the system goes to the next state.
3. In state-2, the previous value of hex0 is set to 7’d127. This basically makes the previous 7-segment output to blank and the player’s input sequence is displayed on second 7-segment display. If timer\_ip is high then the system checks if the succeeding(third) 7-segment display has been blocked or not. If blocked, then it sets end\_gm signal to high denoting the end of the game and the system goes to state-14. If the third 7-segment display is not blocked, then the system goes to state-4. If the timer\_ip pulse is not set to high, the system goes to the next state.
4. In state-3, the system checks if the player has created the required target sequence. If so, then an increment pulse is generated i.e., cout is set to high and the system is set to state-0 where a new sequence is requested. If the player fails to match then the system checks if timer\_ip is high. If so, then the system checks if the succeeding(third) 7-segment display has been blocked or not. If blocked, then it sets end\_gm signal to high denoting the end of the game and the system goes to state-14. If the third 7-segment display is not blocked, then the system goes to next state. If the timer\_ip pulse is not set to high, the system goes to the previous state and displays the player’s input and the process continues.
5. In state-4, the previous value of hex1 is set to 7’d127. This basically makes the previous 7-segment output to blank and the player’s input sequence is displayed on third 7-segment display. If timer\_ip is high then the system checks if the succeeding(first) 7-segment display has been blocked or not. If blocked, then it goes to state 0. If the first 7-segment display is not blocked, then the system goes to state-6. If the timer\_ip pulse is not set to high, the system goes to the next state i.e., the sequence jumps to first 7-segment.
6. In state-5, the system checks if the player has created the required target sequence. If so, then an increment pulse is generated i.e., cout is set to high and the system is set to state-0 where a new sequence is requested. If the player fails to match then the system checks if timer\_ip is high. If so, then the system checks if the succeeding(first) 7-segment display has been blocked or not. If blocked, then it goes to state 0. If the third 7-segment display is not blocked, then the system goes to next state. If the timer\_ip pulse is not set to high, the system goes to the previous state and displays the player’s input and the process continues.
7. state-6 to state-13, the process repeats similar to that of state-4 and state-5 for three more cycles..
8. Once the sequence reaches the 7th 7-segment, the system goes to state-0 and the process repeats.
9. In state-14, the system checks if a new game is requested. If so, then it unblocks all the 7-segment, sets end\_gm signal to logic lows and goes to state-0. If not, it waits in this state continuously.

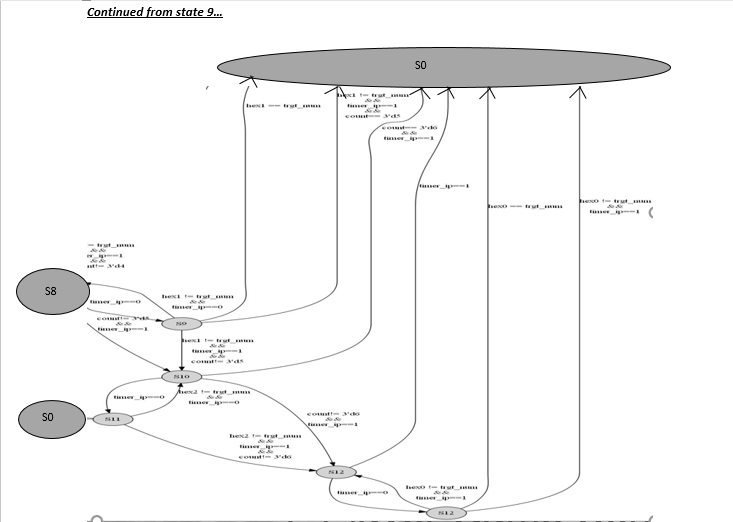
FSM DIAGRAM:



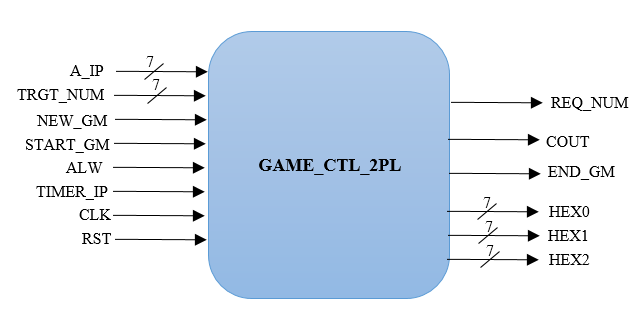








SYMBOL:



1. **Score counter: SimpleCounter.v**

This module is designed to calculate the score for the players. When input score is high then the count value increases by one as long as the count value is less than 20. When the count value is 20, then the signal win and end\_gm is set to logic high. The input and output signals are defined below:

Input:

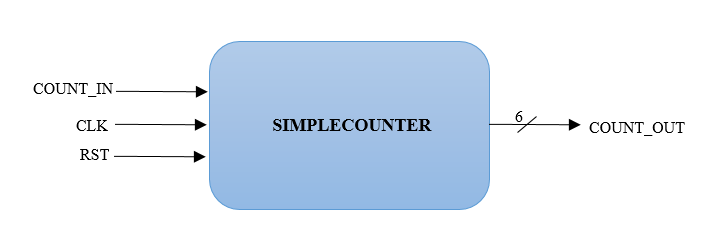
1. Count\_in: This is the input fanned out from the game controller. It is the pulse which tells the counter to increment the score.
2. rst: This resets all the outputs to 0.
3. clk: This is clock signal

Outputs:

1. Count\_out: This is the 6-bit output which is the score for the player.
2. Tout1: This signal is high when the score is 20. Signifying that the player has reached the maximum score and the game has ended.

This module is instantiated twice. One for player 1 and one for player 2.

SYMBOL:



1. **Winner checker: win\_checker.v**

This module is designed to check the final winner. At timeout, the scores for a and b are compared. The player with the highest score is the winner. If both the players have equal scores then both of them are winners. If the timeout has not occurred then, the player who has attained the highest score is declared the winner. The input and output signals are defined below:

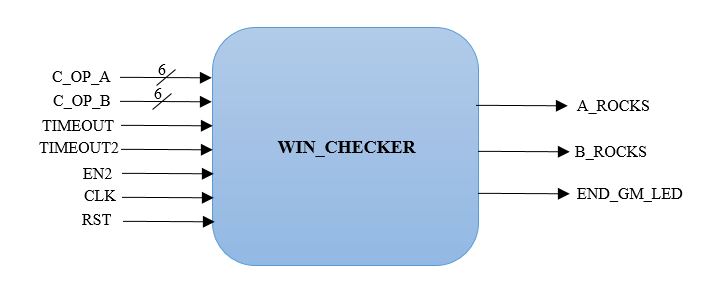
Input:

1. en2: It enables the module only when two-player mode game is selected.
2. c\_op\_a: This is score for player a. It is 6-bit input signal.
3. c\_op\_b: This is score for player b. It is 6-bit input signal.
4. timeout: This signal is high when all the 7-segments are blocked and the game has ended. It is the end\_gm signal from the gm\_ctl\_2pl module of player1.
5. timeout2: This signal is high when all the 7-segments are blocked and the game has ended. It is the end\_gm signal from the gm\_ctl\_2pl module of player2.
6. reset: This resets the output to 0 when enabled.
7. clk: It’s the clock signal

Outputs:

1. a\_rocks: This signal is high when player a is the winner.
2. b\_rocks: This signal is high when player b is the winner.

SYMBOL:



1. **BCD score display: bin2bcd\_score.v**

This module is designed to convert the hexadecimal score into BCD value. We use a 12-bit register shift for the conversion.The binary to BCD conversion is done using the “shift and add 3” algorithm. The algorithm steps are as below:

1. If any column (10's, 1's) is 5 or greater, add 3 to that column.
2. Shift all bits to the left by 1 position.
3. After 4 shifts have been performed, evaluate each column for the BCD values.

The input and output signals are defined below:

Input:

1. bin:This is the binary input which is of 6-bits

Output:

1. tens: This is 4-bit BCD output for tens digit.
2. ones: This is 4-bit BCD output for ones digit.

This module is instantiated twice. One for player 1 and one for player 2.

SYMBOL:



1. **ASCII conversion: ASCII\_decoder.v**

This module converts the input 4 bit “count” to the ASCII values and the resulting ASCII values are divided into two halves which are the 6 bits upper\_nibble and lower\_nibble.

Inputs:

1. Count: this is the 4-bit input of each digit of the score (in BCD)

Outputs:

1. upper\_nibble, lower\_nibble: These are the upper and lower nibbles of the 6 bit ASCII converted values of the count.

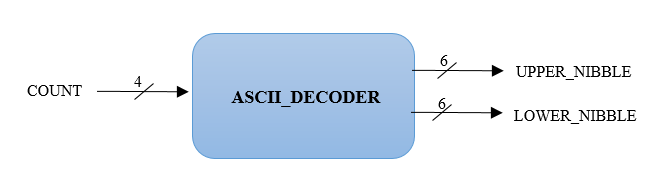
For example, if the score is ‘F’. It is converted to 15 by the bin2bcd.v. Now, digit one is converted to as below:

uppernible= 6’h23 and lowernibble= 6’h21 by the ASCII\_decoder module and digit five is converted to as below:

uppernible= 6’h23 and lowernibble= 6’h25 by the ASCII\_decoder module.

Note: ASCII value for 1 is 31 and 5 is 35.

SYMBOL:



1. **One player LCD Display: LCD\_display\_1pl.v**

This module is used for the one player mode and is thus enabled when the one player mode enable output from the mode\_select module is HIGH.

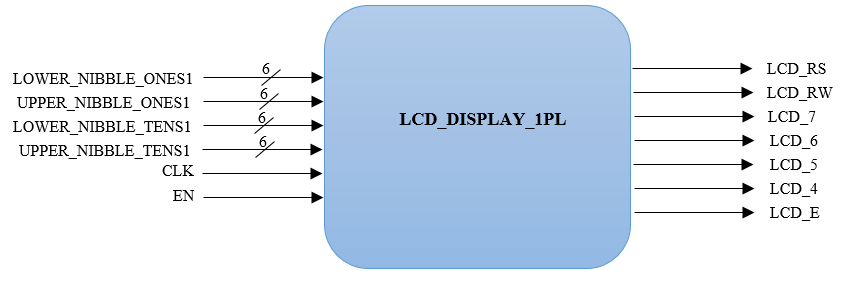
Inputs:

1. upper\_nibble\_ones1, lower\_nibble\_ones1, upper\_nibble\_tens1, lower\_nibble\_tens1- Hexadecimal Each digit of the Score is converted into BCD in bin2bcd\_score module and for each digit the ASCII value is calculated in the ASCII\_decoder module. The output nibbles from the ASCII decoder is given as input.

Outputs:

1. lcd\_rs - This is one bit output signal given to LCD\_RS pin of the LCD display. It performs LCD Command/Data Select, 0 = Command, 1 = Data.
2. lcd\_rw\_top: This is one bit output signal given to LCD\_RW pin of the LCD display. It performs LCD Read/Write Select, 0 = Write, 1 = Read.
3. lcd\_e\_top: This is one bit output signal given to LCD\_EN pin of the LCD display. It enables the LCD when set to high.
4. lcd\_4\_top, lcd\_5\_top, lcd\_6\_top, lcd\_7\_top: They are single bit data outputs given to LCD\_DATA[4], LCD\_DATA[5], LCD\_DATA[6], LCD\_DATA[7] of the LCD module.

SYMBOL:



1. **Two player LCD Display: LCD\_display.v**

This module takes the enable “en” signal from the mode\_select module and thus comes into picture only when two player mode is selected.

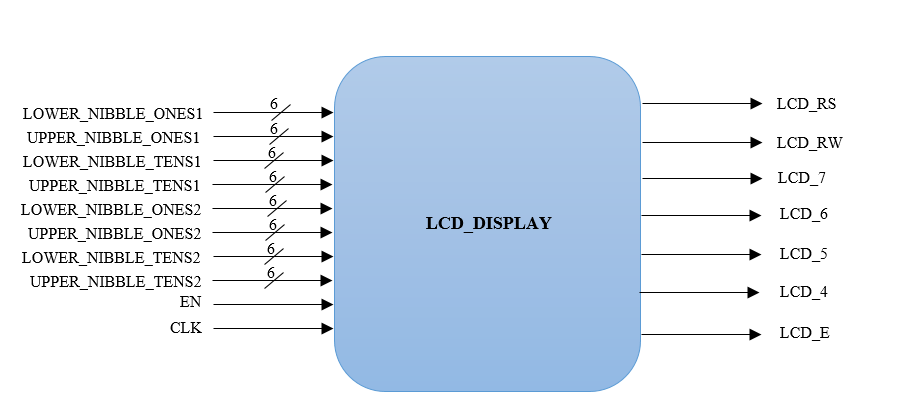
Inputs:

1. lower\_nibble\_ones1, upper\_nibble\_ones1, upper\_nibble\_tens1, lower\_nibble\_tens1, upper\_nibble\_ones2, lower\_nibble\_ones2, upper\_nibble\_tens2, lower\_nibble\_tens2 – Hexadecimal Each digit of the Score is converted into BCD in bin2bcd\_score module and for each digit the ASCII value is calculated in the ASCII\_decoder module. The output nibbles from the ASCII decoder is given as input. The variables ending in 1 are for the player 1 and that ending in 2 are for the player 2.

Outputs:

1. lcd\_rs - This is one bit output signal given to LCD\_RS pin of the LCD display. It performs LCD Command/Data Select, 0 = Command, 1 = Data.
2. lcd\_rw\_top: This is one bit output signal given to LCD\_RW pin of the LCD display. It performs LCD Read/Write Select, 0 = Write, 1 = Read.
3. lcd\_e\_top: This is one bit output signal given to LCD\_EN pin of the LCD display. It enables the LCD when set to high.
4. lcd\_4\_top, lcd\_5\_top, lcd\_6\_top, lcd\_7\_top: They are single bit data outputs given to LCD\_DATA[4], LCD\_DATA[5], LCD\_DATA[6], LCD\_DATA[7] of the LCD module.

SYMBOL:



1. **RAM\_TOP module: RAM\_TOP.v**

This module reads/ write into the RAM\_initialization module using RAM\_Controller module. The inputs and output signals used are the following.

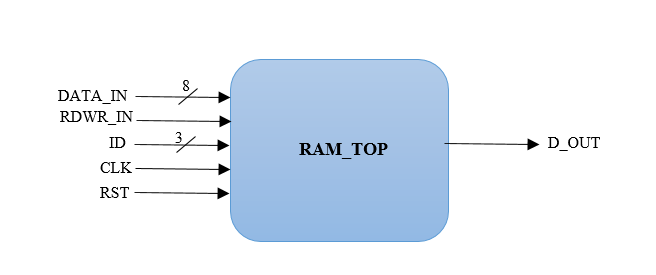
Inputs:

1. id- 3 bit internal address sent from the ROM user module when the input user ID matches into the id present in the ROM.
2. Rdwr\_in – tells the ram to read /write.
3. Data\_in – 8 bit data to be written into the RAM.

Output:

1. d\_out – 8 bit data that is read from the RAM module when wren is LOW

SYMBOL:



* 1. **RAM\_initalization module: RAM.v**

This module stores the maximum scores of the players. When the player’s score is greater than the current score earned by the player, his/her new score is written at the corresponding internal address. This module has “wren” signal as input which tells the module whether to read or write from the memory address. The input address is a 3 bit value address which is the internal ID of the player sent from the ROM after the user ID match is found. The 8 bit data from the address is read and sent out through the “q”.

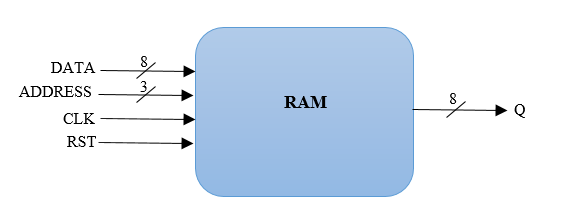
Inputs:

* + 1. address – 3 bit internal ID corresponding to player
    2. Data- 8 bit data to put in input of the player’s scores
    3. Wren- read /write enable signal

Output:

1. q – 7 bit data output when wren=0 i.e. when RAM reads the data from the address

SYMBOL:



* 1. **RAM\_Controller module: RAM\_Controller.v**

This module controls the RAM\_initalization module when to read/ write and at what address the read/write should happen. This module has the following input and output variables.

Inputs: rdwr\_in : takes the command from the score module whether the data has be read or written into the address.

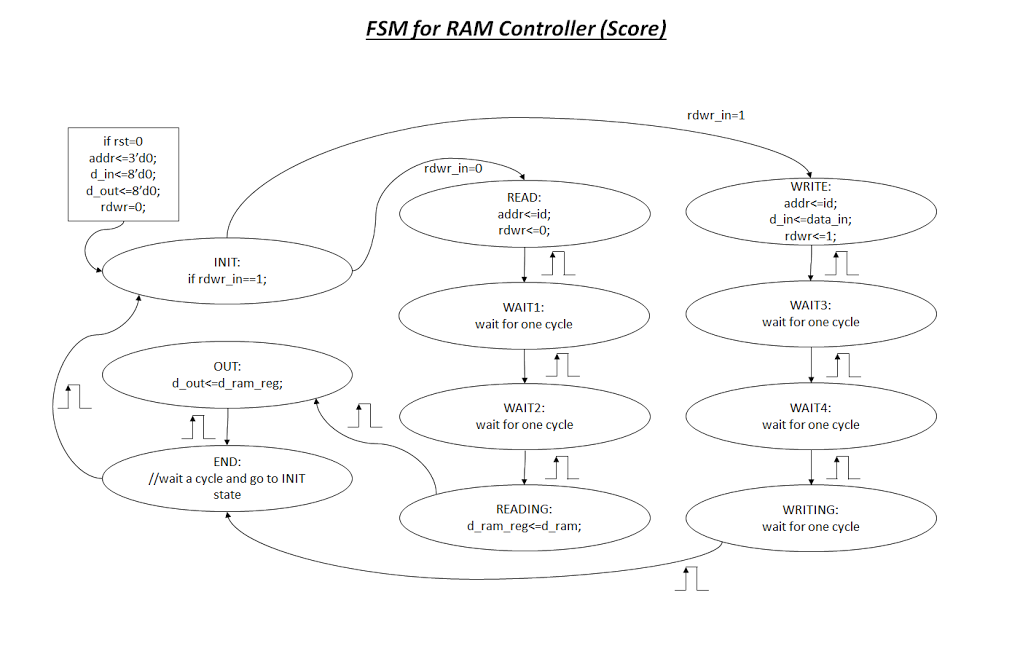
Inputs:

1. Data\_in- 8 bit data value of the score which should be written into the RAM when wren signal is HIGH
2. D\_ram- This is the 8 bit data taken from the RAM module when the controller sends a write signal to it.
3. Id – This is 3 bit internal ID sent from the RAM when it finds the user ID match.

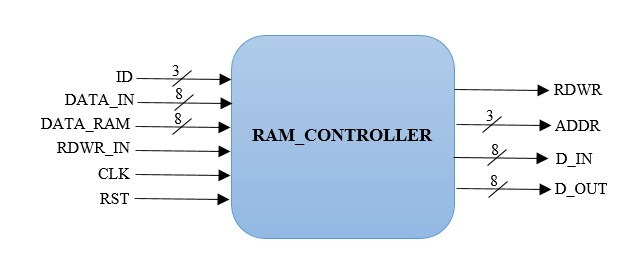
Ouputs:

1. rdwr- command signal to the RAM\_initialization file whether or read/write in RAM.
2. Addr- 3 bit internal ID sent to RAM\_initialization file to read/ write data in RAM.
3. D\_in- 7 bit Data to be written into the RAM module
4. D\_out- 7 bit data that is read from the RAM module

FSM DIAGRAM:



SYMBOL:



1. **RAM for scores: RAM\_Score.v**

This module is used to update the highest scores of the player whenever the game ends. This module takes the user id inputs of the player whose game has ended and sends a write command to the RAM\_TOP to write into the corresponding user’s internal ID the score of the player. So when end game has happened, it first reads the score of the player using the internal address and then compares if the new score is greater than that stored in RAM and updates it if the new score is greater than the old score.

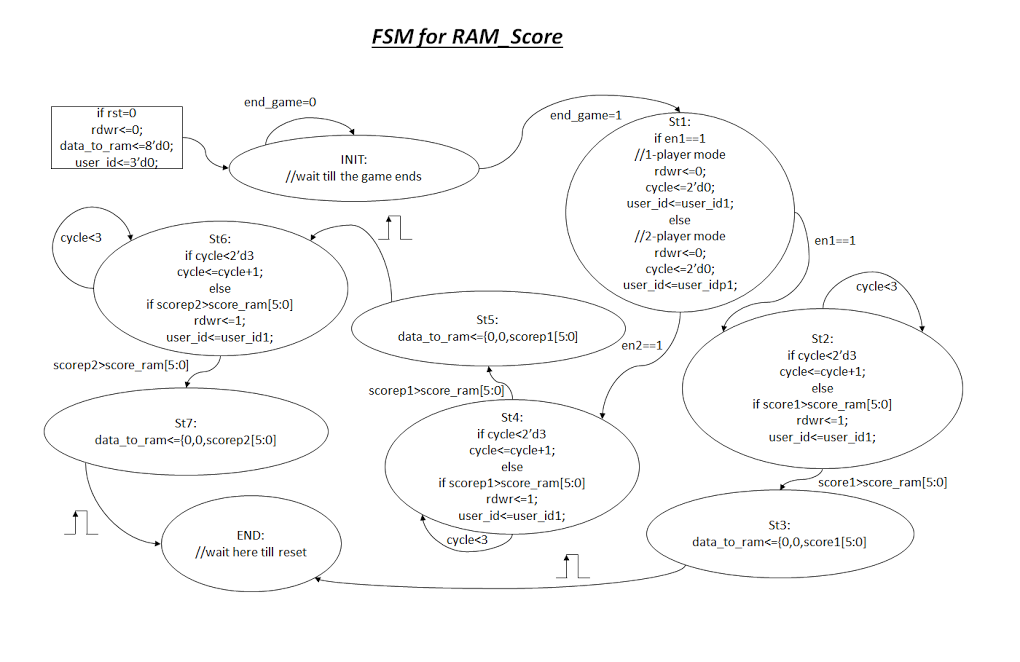
Inputs:

1. end\_game -signal to let the module know that the game has ended.
2. Score\_ram – the score read from the ram module
3. Score1- player’s score in one player mode
4. Scorep1- player A’s score in two player mode
5. Scorep2- player B’s score in two player mode
6. User\_id1 - player’s user id in one player mode
7. User\_idp1 – player A’s user id in two player mode
8. User\_idp2 – player B’s user id in two player mode

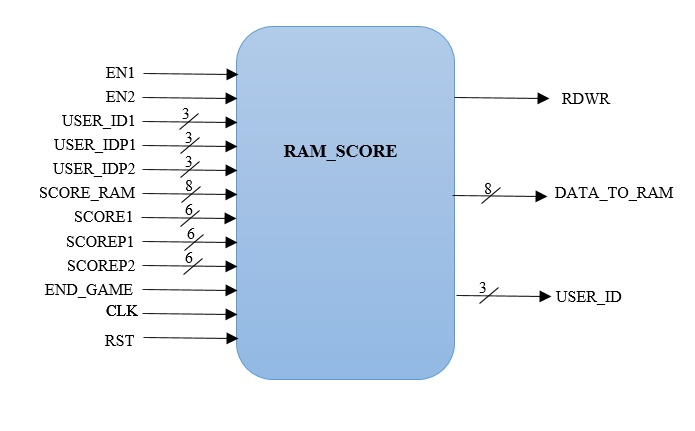
Outputs:

1. rdwr- read/write signal for the RAM\_TOP module
2. Data\_to\_ram – data sent to write into RAM
3. User\_id – address sent to the RAM to read/ write

FSM DIAGRAM:



SYMBOL:



1. **Top module: Final\_Project\_DigitalFortress.v**

This is the main module which defines the game.

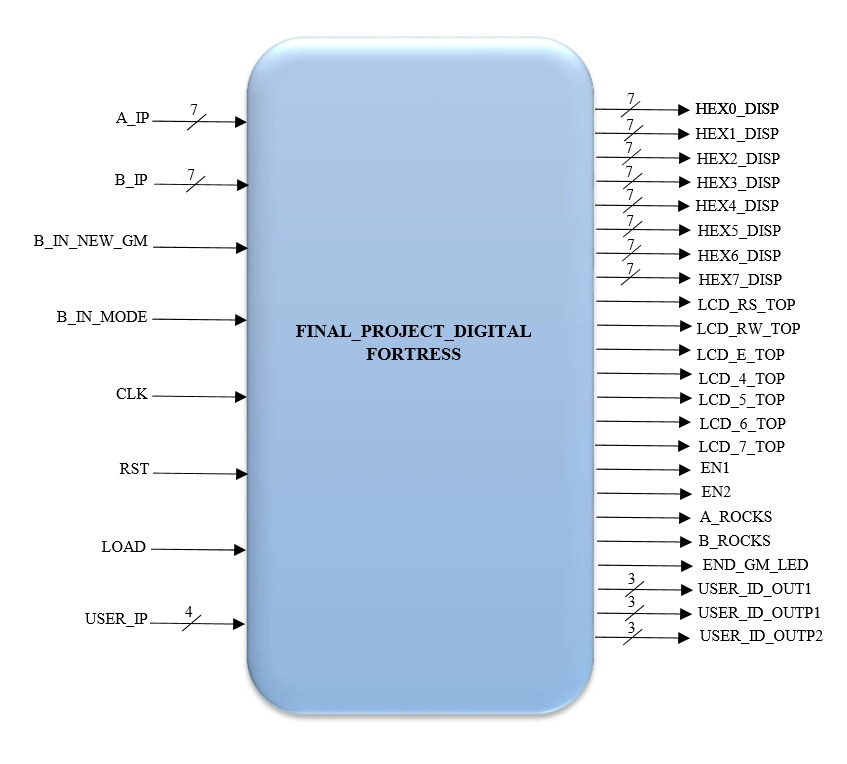
Inputs:

1. a\_ip,b\_ip: 7-bit input for player A and Player B respectively.
2. b\_in\_new\_gm: push button input to restart the game
3. b\_in\_mode: push button input to select game mode (1-player mode/2-player mode)
4. clk,rst: clock and reset signals. reset signal to reset the device
5. load: push button input to load user input.
6. user\_ip: 4-bit user input to enter User ID and Password (for both player A and player B)

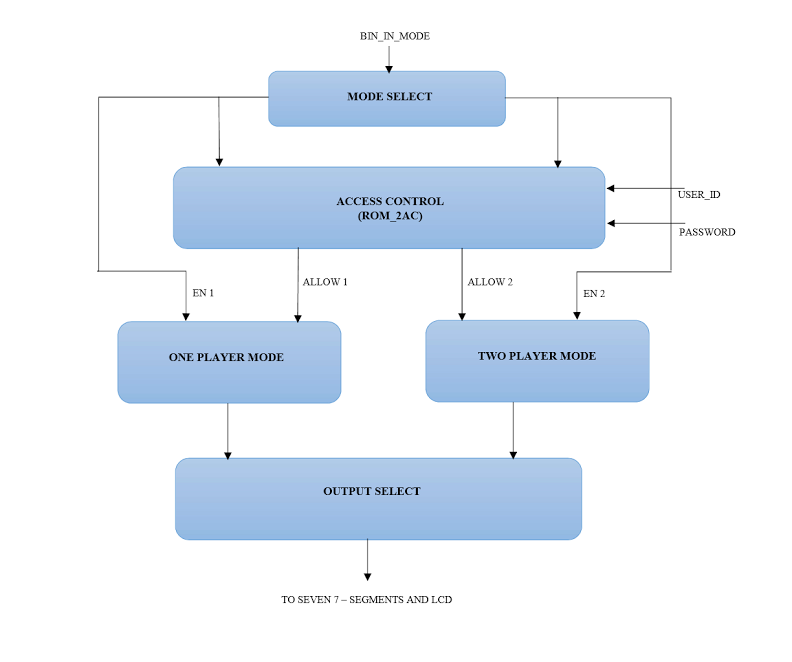
Outputs:

1. hex0\_disp, hex1\_disp, hex2\_disp, hex3\_disp, hex4\_disp, hex5\_disp, hex6\_disp, hex7\_disp: These are 7-bit output signals which are given to 7-segment displays HEX0 to HEX7 respectively.
2. lcd\_rs\_top: This is one bit output signal given to LCD\_RS pin of the LCD display. It performs LCD Command/Data Select, 0 = Command, 1 = Data
3. lcd\_rw\_top: This is one bit output signal given to LCD\_RW pin of the LCD display. It performs LCD Read/Write Select, 0 = Write, 1 = Read.
4. lcd\_e\_top: This is one bit output signal given to LCD\_EN pin of the LCD display. It enables the LCD when set to high.
5. lcd\_4\_top, lcd\_5\_top, lcd\_6\_top, lcd\_7\_top: They are single bit data outputs given to LCD\_DATA[4], LCD\_DATA[5], LCD\_DATA[6], LCD\_DATA[7] of the LCD module.
6. en1: This is one bit output signal that drives a green LED. It will glow if the game user has selected 1-Player mode.
7. en2: This is one bit output signal that drives a green LED. It will glow if the game user has selected 2-Player mode.
8. a\_rocks,b\_rocks: These are two one bit output signals that are connected to a green LED. They will glow when Player A or Player B has won. a\_rocks for Player A and b\_rocks for Player B.
9. end\_gm\_led: This is a one bit output signal that is connected to a Red LED. It will glow when the game ends.

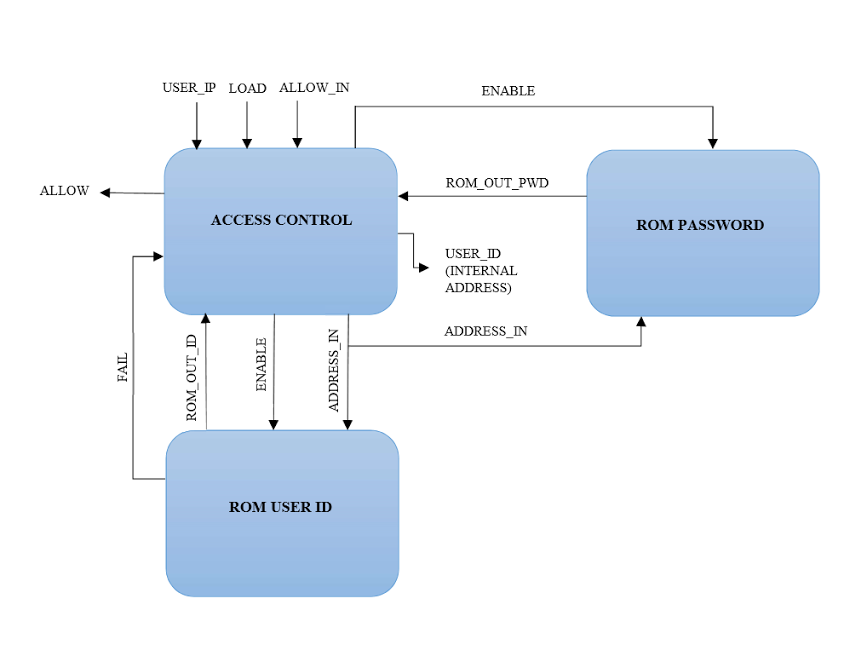
SYMBOL:



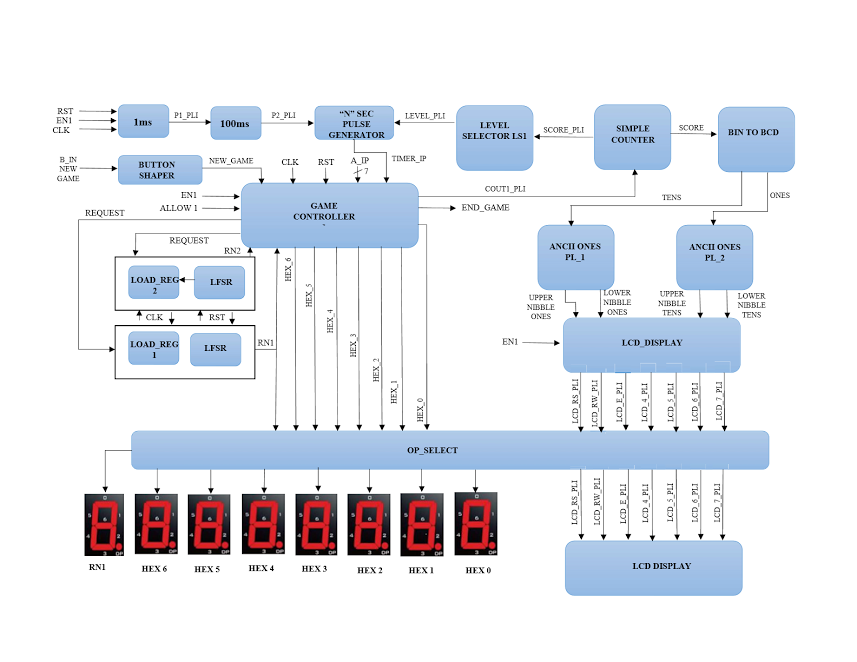
1. **Final Project Architecture:**
2. **Top level architecture:**



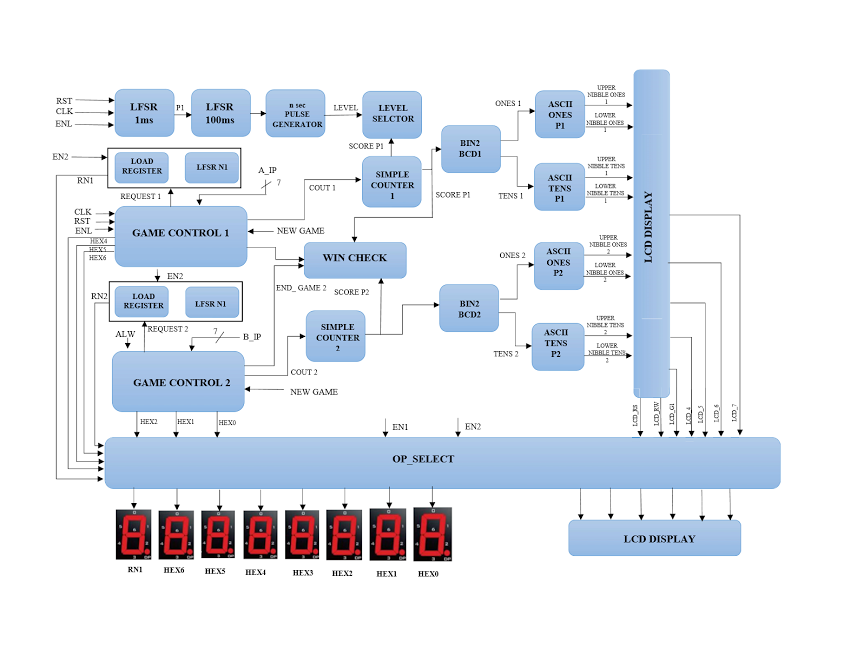
1. **Access control architecture:**



1. **One player mode architecture:**

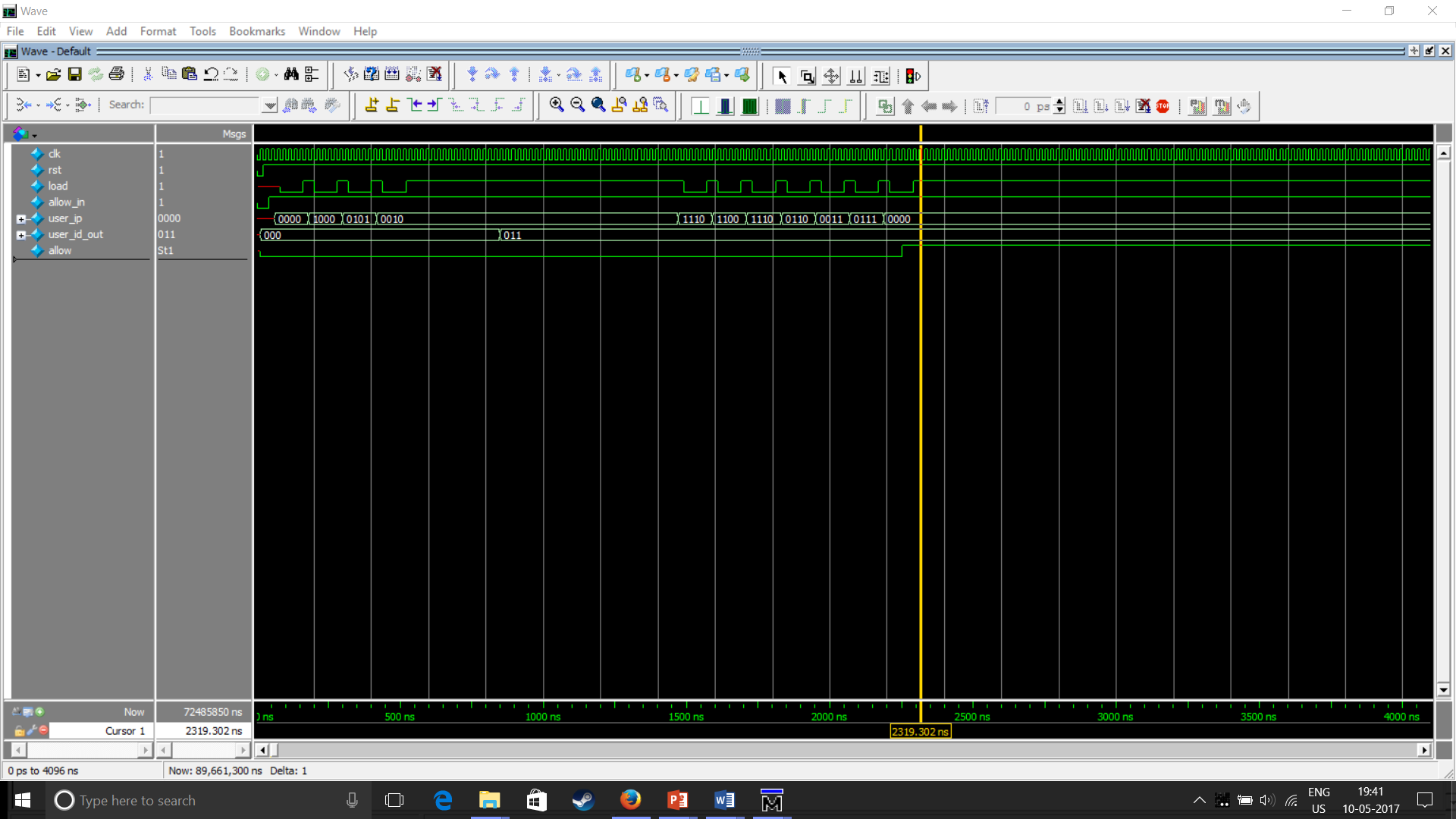


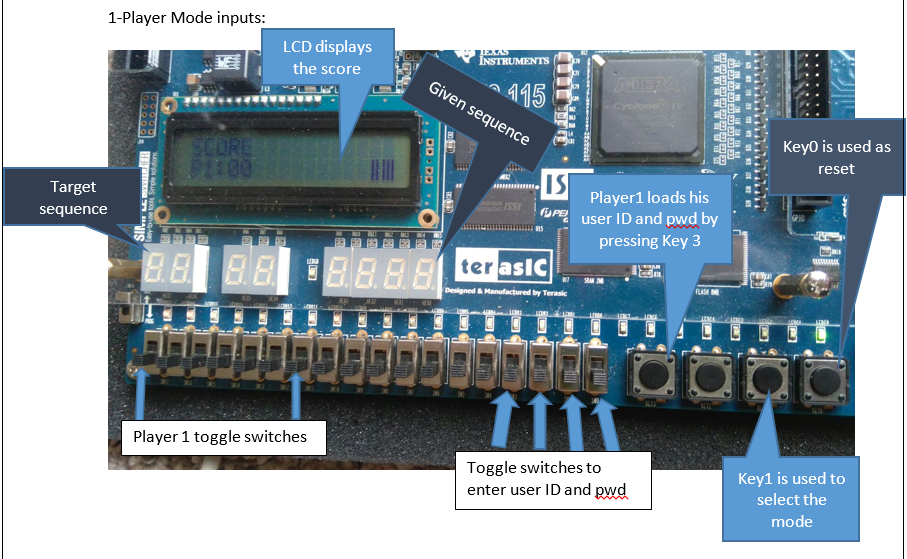
1. **Two player mode architecture:**

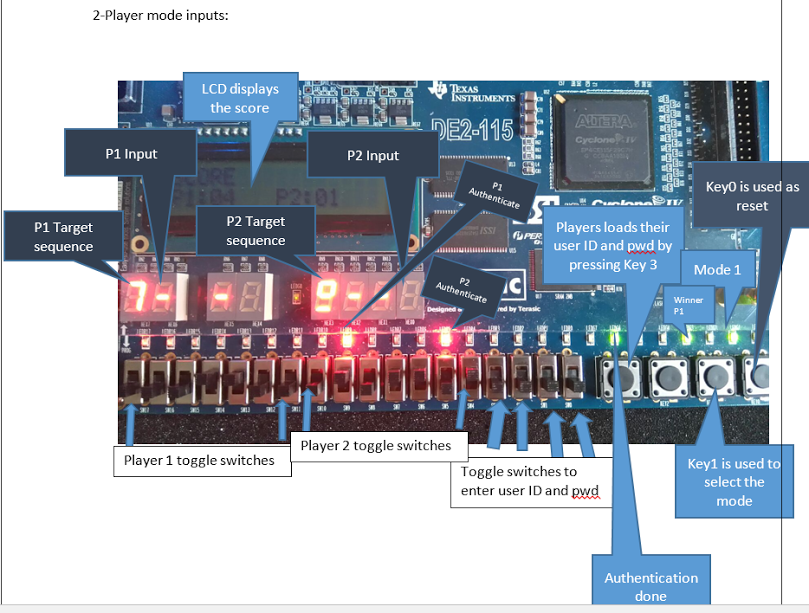


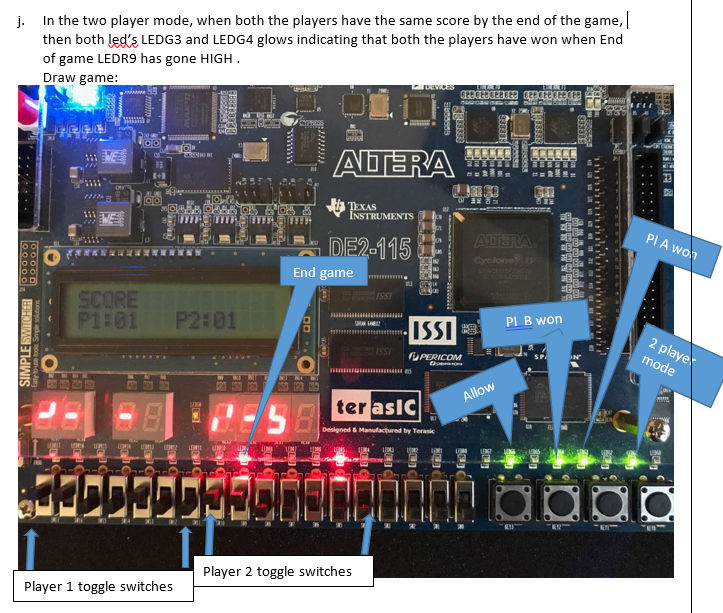
1. **Screen captures of simulation:**
2. Access control simulation output:

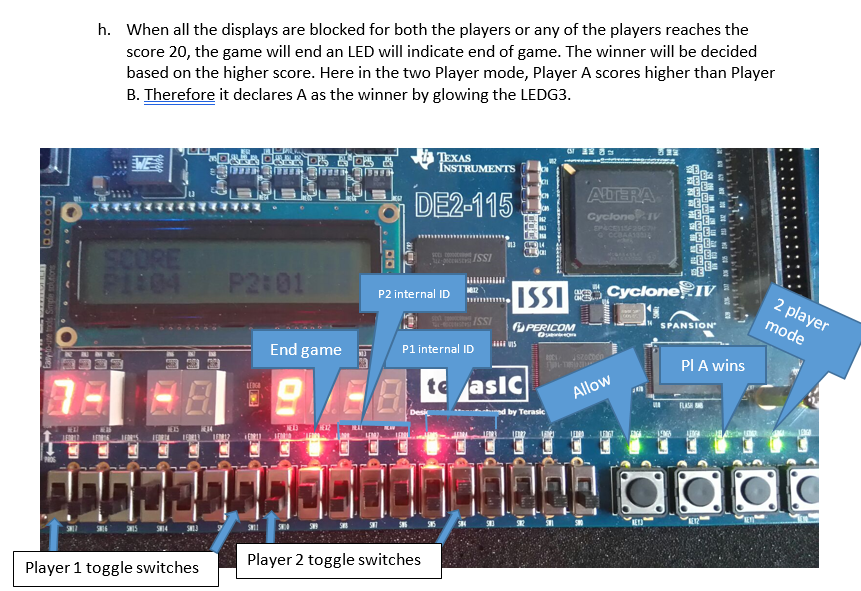
We can see that allow signal is generated after the credentials are verified.

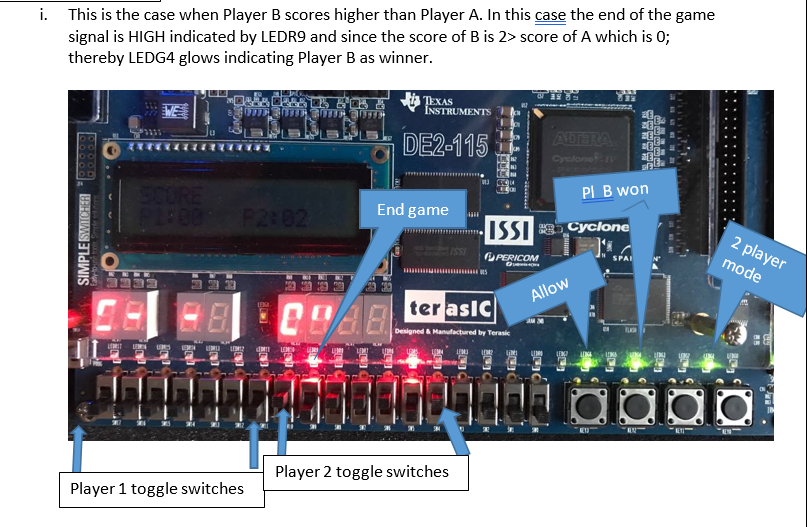


1. **Board Images:**
2. Below image explains all the outputs assigned on the board 









1. **Inference**

In our final project we have successfully designed a pattern matching game with the features including the applications of RAM, ROM and LCD to mention a few.

As a future advancement we would like to improve the user interface by providing instructions over the LCD display.

1. **Code Listing**
2. **Button shaper:**

// ECE6370

// Team: Digital fortress

//verilog module

//b\_in is the input from the button pressed, pulse is the single cycle pulse created

module button\_shaper (b\_in, pulse, clk, reset);

//Declare inputs,outputs and internal variables

input b\_in, clk, reset;

output pulse;

reg pulse;

reg [1:0] state;

//Define state parameters

parameter S0= 2'd0, S1=2'd1;

//Procedure description to generate a single cycle pulse

always @ (posedge clk) begin

if (reset==0) begin

state=3'd0;

end

case(state)

S0:begin

pulse<=0;

if (b\_in==0) begin pulse<=1;

state<=2'd1;

end

else begin

state<= 2'd0;

end

end

S1:begin

pulse<=0;

if (b\_in==0) begin

state<=2'd1;

end

else begin

state<= 2'd0;

end

end

default: state<=2'd0;

endcase

end

endmodule

1. **Mode select: mode\_select.v**

//ECE 6370 Advanced Digital Design

//Team: Digital Fortress

// en1 and en2 are the output enable signals for one player and two player modes respectively.

// b\_in is the button shaper input to be pressed once for one player and twice for two player.

module mode\_select(b\_in,clk,rst,en1, en2);

input b\_in,clk,rst;

output en1, en2;

reg en1, en2;

reg [1:0] state;

always@(posedge clk)

begin

if(rst==1'b0)

begin

en1<=1'b0;

en2<=1'b0;

state<=2'd0;

end

else

begin

case(state)

2'd0:begin

if(b\_in==0) begin

en1<=1'b1;

en2<=1'b0;

state<=2'd1;

end

else begin

state<=2'd0;

end

end

2'd0:begin

if(b\_in==0) begin

en1<=1'b0;

en2<=1'b1;

state<=2'd0;

end

else begin

state<=2'd1;

end

end

default: begin

en1<=1'b0;

en2<=1'b0;

state<=2'd0;

end

endcase

end

end

endmodule

1. Access control: ROM\_2AC.v
2. Single Access control: ROM\_AC.v

//ECE 6370 Advanced Digital Design

//Team: Digital Fortress

//This module uses user\_ip for user id and password input and gives an allow signal when the input password matches the corresponding user id’s passoword.

module ROM\_AC(clk,rst,user\_ip,allow\_in,load,user\_id\_out,allow);

input clk,rst,load, allow\_in;

input [3:0]user\_ip;

wire found,fail,enable\_out;

wire [5:0]pwd\_address;

wire [31:0]rom\_outi;

wire [7:0]rom\_outp;

wire load\_out;

output [2:0]user\_id\_out;

output allow;

ButtonShaper bs1(load, rst, load\_out, clk);

ROM\_ID\_TOP id\_top(enable\_out, clk, rst, rom\_outi,pwd\_address[4:0],fail);

ROM2 pwd\_top(found,clk,rst,pwd\_address,rom\_outp);

AC\_ROM\_USER ac2(fail,user\_ip,allow\_in,rom\_outp,rom\_outi,load\_out,enable\_out,

rst,clk,allow,user\_id\_out,pwd\_address,found);

endmodule

1. User ID ROM: ROM\_ID\_TOP.v

//ECE 6370 Advanced Digital Design

//Team: Digital Fortress

//This module takes in the address\_in as 0 and starts doing exhaustive search and //whenever it matches with the ID sends the data\_out which contains the userID //and the internal ID of the player.

module ROM\_ID\_TOP(en, clk, rst, data\_out,address\_in,fail);

input en,clk,rst;

input [4:0]address\_in;

wire [31:0]data\_in;

wire [4:0]address\_out;

output [31:0]data\_out;

output fail;

ROM\_Controller\_ID rom\_control(en,clk,rst,address\_in,data\_in,data\_out,address\_out,fail);

ROM\_KP rom\_data(address\_out,clk,data\_in);

//AC\_ROM\_USER ac\_top(fail,user\_ip,allow\_in,rom\_outp,rom\_outi,load,enable\_out,rst,clk,allow,

//user\_id\_out,pwd\_address,found);

endmodule

1. Password ROM: ROM2.v

//ECE 6370 Advanced Digital Design

//Team: Digital Fortress

//This module is the top module for the ROM password

module ROM2(en,clk,rst,address\_in,d\_out);

input en,clk,rst;

input [5:0]address\_in;

output [7:0]d\_out;

wire [7:0]d\_rom;

wire [5:0]address\_out;

RomController R1(en,clk,rst,d\_rom,address\_in,address\_out,d\_out);

Rom\_initialization\_file R2(address\_out,clk,d\_rom);

endmodule

1. Credential check: AC\_ROM\_USER.v

//ECE 6370 Advanced Digital Design

//Team: Digital Fortress

//This module checks the user id and the password of the player entered with that of the values stored in ROM.

module AC\_ROM\_USER(fail,user\_ip,allow\_in,rom\_outp,rom\_outi,load,enable\_out,rst,clk,allow,

user\_id\_out,pwd\_address,found);

input [3:0]user\_ip;

input [7:0]rom\_outp;

input [31:0]rom\_outi;

input load,rst,clk,allow\_in,fail;

output reg allow, enable\_out;

output reg [2:0]user\_id\_out;

output reg [5:0]pwd\_address;

output reg found;

reg [15:0]user\_fullid;

reg [4:0]State;

reg [2:0] chance,cycle ;

reg [27:0] dint,full\_pwd;

parameter INIT=5'd0,ST1=5'd1,ST2=5'd2,ST3=5'd3,ST4=5'd4,ST5=5'd5,ST6=5'd6,ST7=5'd7,ST8=5'd8,ST9=5'd9,ST10=5'd10,ST11=5'd11,ST12=5'd12,ST13=5'd13,ST14=5'd14,ST15=5'd15,ST16=5'd16,ST17=5'd17,ST18=5'd18,ST19=5'd19,ST20=5'd20,ST21=5'd21,END=5'd22;

always@(posedge clk)

begin

if(rst==0) begin

allow=0;

user\_id\_out=3'd0;

enable\_out<=0;

State= INIT;

pwd\_address<=6'd0;

cycle<=3'd0;

chance<=2'd0;

end

else begin

if(allow\_in== 1) begin

case(State)

INIT: begin

if(load==1) begin

user\_fullid[15:12]<= user\_ip;

State= ST2;

enable\_out<=0;

end

else begin

State= INIT;

end

end

ST2: begin

if(load==1) begin

user\_fullid[11:8]<= user\_ip;

State= ST3;

enable\_out<=0;

end

else begin

State= ST2;

end

end

ST3: begin

if(load==1) begin

user\_fullid[7:4]<= user\_ip;

State= ST4;

enable\_out<=0;

end

else begin

State= ST3;

end

end

ST4: begin

if(load==1) begin

user\_fullid[3:0]<= user\_ip;

State= ST5;

enable\_out<=0;

end

else begin

State= ST4;

end

end

ST5: begin

enable\_out<=1;

State<= ST6;

end

ST6: begin

if(user\_fullid == rom\_outi[15:0])

begin

user\_id\_out<= rom\_outi[30:28];

pwd\_address[5:3]<= rom\_outi[30:28];

pwd\_address[2:0]<= 3'b000;

found<=1;

enable\_out<=0;

State<= ST7;

end

else begin

if(fail==1) begin

enable\_out<=0;

allow<=1;

State<=END;

end

else begin

found<=0;

enable\_out<=1;

State<= ST6;

end

end

end

ST14: begin //password 0

if(load==1 ) begin

full\_pwd[27:24]<= user\_ip;

State= ST15;

//enable\_pwd<=0;

end

else begin

State= ST14;

end

end

ST15: begin //password 1

if(load==1 ) begin

full\_pwd[23:20]<= user\_ip;

State= ST16;

//enable\_pwd<=0;

end

else begin

State= ST15;

end

end

ST16: begin //password 2

if(load==1 ) begin

full\_pwd[19:16]<= user\_ip;

State= ST17;

//enable\_pwd<=0;

end

else begin

State= ST16;

end

end

ST17: begin //password 3

if(load==1 ) begin

full\_pwd[15:12]<= user\_ip;

State= ST18;

//enable\_pwd<=0;

end

else begin

State= ST17;

end

end

ST18: begin //password 4

if(load==1 ) begin

full\_pwd[11:8]<= user\_ip;

State= ST19;

//enable\_pwd<=0;

end

else begin

State= ST18;

end

end

ST19: begin //password 5

if(load==1 ) begin

full\_pwd[7:4]<= user\_ip;

State= ST20;

//enable\_pwd<=0;

end

else begin

State= ST19;

end

end

ST20: begin //password 6

if(load==1 ) begin

full\_pwd[3:0]<= user\_ip;

State= ST21;

//enable\_pwd<=0;

end

else begin

State= ST20;

end

end

ST7: begin //password bit 0

enable\_out<=1;

if(cycle< 3'd5 ) begin

cycle<=cycle+1;

State= ST7;

end

else begin

dint[27:24]=rom\_outp[3:0];

cycle<=3'd0;

State<=ST8;

end

end

ST8: begin //password bit 1

enable\_out<=1;

if(cycle< 3'd5 ) begin

cycle<=cycle+1;

State= ST8;

end

else begin

dint[23:20]=rom\_outp[3:0];

cycle<=3'd0;

State<=ST9;

end

end

ST9: begin //password bit 2

enable\_out<=1;

if(cycle< 3'd5 ) begin

cycle<=cycle+1;

State= ST9;

end

else begin

dint[19:16]=rom\_outp[3:0];

cycle<=3'd0;

State<=ST10;

end

end

ST10: begin //password bit 3

enable\_out<=1;

if(cycle< 3'd5 ) begin

cycle<=cycle+1;

State= ST10;

end

else begin

dint[15:12]=rom\_outp[3:0];

cycle<=3'd0;

State<=ST11;

end

end

ST11: begin //password bit 4

enable\_out<=1;

if(cycle< 3'd5 ) begin

cycle<=cycle+1;

State= ST11;

end

else begin

dint[11:8]=rom\_outp[3:0];

cycle<=3'd0;

State<=ST12;

end

end

ST12: begin //password bit 5

enable\_out<=1;

if(cycle< 3'd5 ) begin

cycle<=cycle+1;

State= ST12;

end

else begin

dint[7:4]=rom\_outp[3:0];

cycle<=3'd0;

State<=ST13;

end

end

ST13: begin //password bit 5

enable\_out<=1;

if(cycle< 3'd5 ) begin

cycle<=cycle+1;

State= ST13;

end

else begin

dint[3:0]=rom\_outp[3:0];

cycle<=3'd0;

State<=ST14;

end

end

ST21: begin //password check begin1

if(dint==full\_pwd)

begin //begin2

allow<=1;

State<=END;

end //end2

else

begin //begin3

if(chance==3)

begin //begin 4

allow<=1;

State<=END;

end //end4

else

begin //b5

allow<=0;

chance<=chance+1;

State<=ST14;

end //e5

end //e3

end //e1

END: begin

State<=END;

end

default: begin

State<=INIT;

allow<=0;

user\_id\_out=3'd0;

enable\_out<=0;

pwd\_address<=6'd0;

cycle<=3'd0;

chance<=2'd0;

end

endcase

end

end

end

endmodule

1. Output select: op\_select.v

//ECE 6370 Advanced Digital Design

// Team: Digital Fortress

//verilog module

//disp is the output which is selected to display

module op\_select (hex0\_disp, hex1\_disp, hex2\_disp, hex3\_disp, hex4\_disp, hex5\_disp, hex6\_disp, hex7\_disp, lcd\_rs\_top, lcd\_rw\_top, lcd\_e\_top, lcd\_4\_top, lcd\_5\_top, lcd\_6\_top, lcd\_7\_top, en1, en2, hex0\_pl1, hex1\_pl1, hex2\_pl1, hex3\_pl1, hex4\_pl1, hex5\_pl1, hex6\_pl1, RN1\_pl1, lcd\_rs\_pl1, lcd\_rw\_pl1, lcd\_e\_pl1, lcd\_4\_pl1, lcd\_5\_pl1, lcd\_6\_pl1, lcd\_7\_pl1, lcd\_rs, lcd\_rw, lcd\_e, lcd\_4, lcd\_5, lcd\_6, lcd\_7, hex0, hex1, hex2, RN2, hex4, hex5, hex6, RN1, clk, reset);

//Declare inputs,outputs and internal variables

input [6:0] hex0\_pl1, hex1\_pl1, hex2\_pl1, hex3\_pl1, hex4\_pl1, hex5\_pl1, hex6\_pl1, RN1\_pl1;

input [6:0] hex0, hex1, hex2, RN2, hex4, hex5, hex6, RN1;

input lcd\_rs\_pl1, lcd\_rw\_pl1, lcd\_e\_pl1, lcd\_4\_pl1, lcd\_5\_pl1, lcd\_6\_pl1, lcd\_7\_pl1;

input lcd\_rs, lcd\_rw, lcd\_e, lcd\_4, lcd\_5, lcd\_6, lcd\_7;

input clk, reset, en1, en2;

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

output lcd\_rs\_top, lcd\_rw\_top, lcd\_e\_top, lcd\_4\_top, lcd\_5\_top, lcd\_6\_top, lcd\_7\_top;

reg [6:0] hex0\_disp, hex1\_disp, hex2\_disp, hex3\_disp, hex4\_disp, hex5\_disp, hex6\_disp, hex7\_disp;

reg lcd\_rs\_top, lcd\_rw\_top, lcd\_e\_top, lcd\_4\_top, lcd\_5\_top, lcd\_6\_top, lcd\_7\_top;

//Procedure description to display score in BCD at the end of game

always @ (posedge clk) begin

if (reset==0) begin

hex0\_disp<= 7'd127;

hex1\_disp<= 7'd127;

hex2\_disp<= 7'd127;

hex3\_disp<= 7'd127;

hex4\_disp<= 7'd127;

hex5\_disp<= 7'd127;

hex6\_disp<= 7'd127;

hex7\_disp<= 7'd127;

lcd\_rs\_top<= lcd\_rs\_pl1;

lcd\_rw\_top<= lcd\_rw\_pl1;

lcd\_e\_top<= lcd\_e\_pl1;

lcd\_4\_top<= lcd\_4\_pl1;

lcd\_5\_top<= lcd\_5\_pl1;

lcd\_6\_top<= lcd\_6\_pl1;

lcd\_7\_top<= lcd\_7\_pl1;

end

else begin

if (en1==1'b1 && en2== 1'b0) begin

hex0\_disp<= hex0\_pl1;

hex1\_disp<= hex1\_pl1;

hex2\_disp<= hex2\_pl1;

hex3\_disp<= hex3\_pl1;

hex4\_disp<= hex4\_pl1;

hex5\_disp<= hex5\_pl1;

hex6\_disp<= hex6\_pl1;

hex7\_disp<= RN1\_pl1;

lcd\_rs\_top<= lcd\_rs\_pl1;

lcd\_rw\_top<= lcd\_rw\_pl1;

lcd\_e\_top<= lcd\_e\_pl1;

lcd\_4\_top<= lcd\_4\_pl1;

lcd\_5\_top<= lcd\_5\_pl1;

lcd\_6\_top<= lcd\_6\_pl1;

lcd\_7\_top<= lcd\_7\_pl1;

end

else if (en2==1'b1 && en1== 1'b0) begin

hex0\_disp<= hex0;

hex1\_disp<= hex1;

hex2\_disp<= hex2;

hex3\_disp<= RN2;

hex4\_disp<= hex4;

hex5\_disp<= hex5;

hex6\_disp<= hex6;

hex7\_disp<= RN1;

lcd\_rs\_top<= lcd\_rs;

lcd\_rw\_top<= lcd\_rw;

lcd\_e\_top<= lcd\_e;

lcd\_4\_top<= lcd\_4;

lcd\_5\_top<= lcd\_5;

lcd\_6\_top<= lcd\_6;

lcd\_7\_top<= lcd\_7;

end

end

end

endmodule

1. Load register: LoadReg.v

//ECE 6370 Advanced Digital Design

//Team: Digital Fortress

// This module is used as register to hold the input

module LoadReg(Input,en,Clk,Rst,Load,Output);

input [6:0] Input;

input Clk,Rst, en;

input Load;

output [6:0] Output;

reg [6:0] Output;

always@(posedge Clk)

begin

if(Rst==1'b0)

begin

Output=7'b1111111;

end

else if(Load==1'b1 && en==1'b1)

begin

Output<=Input;

end

end

endmodule

1. Random number generator 1
2. LFSR module: LFSR1.v

//ECE 6370 Advanced Digital Design

//Team: Digital Fortress

module LFSR1(clk,rst, q);

input clk,rst;

output [15:0] q;

reg [15:0] LFSR;

wire feedback = LFSR[15];

always @(posedge clk)

begin

if(rst==1'b0)

begin

LFSR=16'd65535;

end

else

begin

LFSR[0] <= feedback;

LFSR[1] <= LFSR[0];

LFSR[2] <= LFSR[1];

LFSR[3] <= LFSR[2];

LFSR[4] <= LFSR[3] ^ feedback;

LFSR[5] <= LFSR[4];

LFSR[6] <= LFSR[5];

LFSR[7] <= LFSR[6];

LFSR[8] <= LFSR[7];

LFSR[9] <= LFSR[8];

LFSR[10] <= LFSR[9];

LFSR[11] <= LFSR[10];

LFSR[12] <= LFSR[11];

LFSR[13] <= LFSR[12] ^ feedback;

LFSR[14] <= LFSR[13];

LFSR[15] <= LFSR[14] ^ feedback;

end

end

assign q = LFSR;

endmodule

1. Load register: LoadReg.v

//ECE 6370 Advanced Digital Design

//Team: Digital Fortress

// This module is used as register to hold the input

module LoadReg(Input,en,Clk,Rst,Load,Output);

input [6:0] Input;

input Clk,Rst, en;

input Load;

output [6:0] Output;

reg [6:0] Output;

always@(posedge Clk)

begin

if(Rst==1'b0)

begin

Output=7'b1111111;

end

else if(Load==1'b1 && en==1'b1)

begin

Output<=Input;

end

end

endmodule

1. Random number generator 2
2. LFSR module: LFSR2.v

//ECE 6370 Advanced Digital Design

//Team: Digital Fortress

module LFSR1(clk,rst, q);

input clk,rst;

output [15:0] q;

reg [15:0] LFSR;

wire feedback = LFSR[15];

always @(posedge clk)

begin

if(rst==1'b0)

begin

LFSR=16'd65535;

end

else

begin

LFSR[0] <= feedback;

LFSR[1] <= LFSR[0];

LFSR[2] <= LFSR[1];

LFSR[3] <= LFSR[2];

LFSR[4] <= LFSR[3] ^ feedback;

LFSR[5] <= LFSR[4];

LFSR[6] <= LFSR[5];

LFSR[7] <= LFSR[6];

LFSR[8] <= LFSR[7];

LFSR[9] <= LFSR[8];

LFSR[10] <= LFSR[9];

LFSR[11] <= LFSR[10];

LFSR[12] <= LFSR[11];

LFSR[13] <= LFSR[12] ^ feedback;

LFSR[14] <= LFSR[13];

LFSR[15] <= LFSR[14] ^ feedback;

end

end

assign q = LFSR;

endmodule

1. Load register: LoadReg.v

//ECE 6370 Advanced Digital Design

//Team: Digital Fortress

// This module is used as register to hold the input

module LoadReg(Input,en,Clk,Rst,Load,Output);

input [6:0] Input;

input Clk,Rst, en;

input Load;

output [6:0] Output;

reg [6:0] Output;

always@(posedge Clk)

begin

if(Rst==1'b0)

begin

Output=7'b1111111;

end

else if(Load==1'b1 && en==1'b1)

begin

Output<=Input;

end

end

endmodule

1. Game timer
2. Milli-second pulse generator: LFSR\_1ms.v

//ECE 6370 Advanced Digital Design

//Team: Digital Fortress

//This module generates a single cycle output pulse for every millisecond.

module LFSR\_1ms(clk,rst,en, Out1);

input clk,rst,en;

output Out1;

reg Out1;

reg [15:0] LFSR;

wire feedback = LFSR[15];

always @(posedge clk)

begin

if(rst==1'b0)

begin

LFSR<=16'd65535;

Out1<=1'b0;

end

else if(en==1'b0)

begin

Out1<=1'b0;

end

else if(LFSR==16'b1011011011110101)

begin

LFSR=16'd65535;

Out1<=1'b1;

end

else

begin

Out1<=1'b0;

LFSR[0] <= feedback;

LFSR[1] <= LFSR[0];

LFSR[2] <= LFSR[1] ^ feedback;

LFSR[3] <= LFSR[2] ^ feedback;

LFSR[4] <= LFSR[3];

LFSR[5] <= LFSR[4] ^ feedback;

LFSR[6] <= LFSR[5];

LFSR[7] <= LFSR[6];

LFSR[8] <= LFSR[7];

LFSR[9] <= LFSR[8];

LFSR[10] <= LFSR[9];

LFSR[11] <= LFSR[10];

LFSR[12] <= LFSR[11];

LFSR[13] <= LFSR[12];

LFSR[14] <= LFSR[13];

LFSR[15] <= LFSR[14];

end

end

endmodule

1. 100 milli-second pulse generator: LFSR\_100ms.v

//ECE 6370 Advanced Digital Design

//Team: Digital Fortress

//This module generates a single cycle output high pulse for every 100ms

module LFSR\_100ms(clk,rst,Clk\_in, Out2);

input clk,rst,Clk\_in;

output Out2;

reg Out2;

reg [6:0]count1;

always @(posedge clk)

begin

if(rst==1'b0)

begin

Out2<=1'b0;

count1<=7'd0;

end

else if(Clk\_in==1'b1)

begin

if(count1==7'd100)

begin

Out2<=1'b1;

count1<=7'd0;

end

else

begin

Out2<=1'b0;

count1<=count1+1;

end

end

else

begin

Out2<=1'b0;

end

end

endmodule

1. Variable timer: nSecPulseGenerator.v

//ECE 6370 Advanced Digital Design

//Team: Digital Fortress

//This module generates a single cycle output pulse based on the level of the player.

module nSecPulseGenerator(Clk\_in,Level,Clk,Rst,Clk\_out);

input Clk\_in,Clk,Rst;

input [2:0] Level;

output Clk\_out;

reg Clk\_out;

reg [5:0] count1;

reg [4:0] count2;

reg [3:0] count3;

reg [3:0] count4;

reg [3:0] count5;

reg [2:0] count6;

reg [2:0] count7;

reg [2:0] count8;

always @(posedge Clk)

begin

if(Rst==0)

begin

Clk\_out=0;count1<=0;count2<=0;count3<=0;count4<=0;count5<=0;count6<=0;count7<=0;count8<=0;

end

else

begin

case(Level)

3'd0: begin

if(Clk\_in==1)

begin

if(count1==6'd60)

begin

count1<=6'd0;

Clk\_out<=1;

end

else

begin

count1<=count1+1;

Clk\_out<=0;

end

end

else

begin

Clk\_out<=0;

end

end

3'd1: begin

if(Clk\_in==1)

begin

if(count2==5'd50)

begin

count2<=5'd0;

Clk\_out<=1;

end

else

begin

count2<=count2+1;

Clk\_out<=0;

end

end

else

begin

Clk\_out<=0;

end

end

3'd2: begin

if(Clk\_in==1)

begin

if(count3==4'd40)

begin

count3<=4'd0;

Clk\_out<=1;

end

else

begin

count3<=count3+1;

Clk\_out<=0;

end

end

else

begin

Clk\_out<=0;

end

end

3'd3: begin

if(Clk\_in==1)

begin

if(count4==4'd30)

begin

count4<=4'd0;

Clk\_out<=1;

end

else

begin

count4<=count4+1;

Clk\_out<=0;

end

end

else

begin

Clk\_out<=0;

end

end

3'd4: begin

if(Clk\_in==1)

begin

if(count5==4'd8)

begin

count5<=4'd0;

Clk\_out<=1;

end

else

begin

count5<=count5+1;

Clk\_out<=0;

end

end

else

begin

Clk\_out<=0;

end

end

3'd5: begin

if(Clk\_in==1)

begin

if(count6==3'd7)

begin

count6<=3'd0;

Clk\_out<=1;

end

else

begin

count6<=count6+1;

Clk\_out<=0;

end

end

else

begin

Clk\_out<=0;

end

end

3'd6: begin

if(Clk\_in==1)

begin

if(count7==3'd6)

begin

count7<=3'd0;

Clk\_out<=1;

end

else

begin

count7<=count7+1;

Clk\_out<=0;

end

end

else

begin

Clk\_out<=0;

end

end

3'd7: begin

if(Clk\_in==1)

begin

if(count8==3'd5)

begin

count8<=3'd0;

Clk\_out<=1;

end

else

begin

count8<=count8+1;

Clk\_out<=0;

end

end

else

begin

Clk\_out<=0;

end

end

default: begin

Clk\_out<=0;

end

endcase

end

end

endmodule

1. Level selector: LevelSelector.v

//ECE-6370 Advanced Digital Design

//Team: Digital Fortress

// This module takes the score as input and sets the difficulty level as output based on the scores

module LevelSelector(Score,Clk,Rst,Level);

input Clk,Rst;

input [5:0] Score;

output [2:0] Level;

reg [2:0] Level;

always@(posedge Clk)

begin

if(Rst==0)

begin

Level<=3'd0;

end

else

begin

if(Score<6'd5)

begin

Level<=3'd0;

end

if(Score>=6'd5 & Score<6'd10)

begin

Level<=3'd1;

end

if(Score>=6'd10 & Score<6'd15)

begin

Level<=3'd2;

end

if(Score>=6'd15 & Score<6'd20)

begin

Level<=3'd3;

end

end

end

endmodule

1. One Player Game controller: game\_ctl.v

//ECE 6370 Advanced Digital Design

//Team: Digital Fortress

// this module compares if the player has entered his inputs that matches the one generated by //the random number generator

module game\_ctl(cout, hex0, hex1, hex2, hex3, hex4, hex5, hex6, req\_num, end\_gm, new\_num, a\_ip, trgt\_num, new\_gm, start\_gm, alw, timer\_ip, clk, reset);

input [6:0] new\_num, a\_ip, trgt\_num;

input new\_gm, start\_gm, alw, timer\_ip, clk, reset;

output reg req\_num, end\_gm, cout;

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

reg [4:0] state;

reg [2:0] count;

always@(posedge clk)

begin

if(reset==0) begin

count<= 3'd7;

end\_gm<= 0; //test<=1'b0;

hex0<= 7'd127; hex1<= 7'd127; hex2<= 7'd127; hex3<= 7'd127; hex4<= 7'd127; hex5<= 7'd127; hex6<= 7'd127;

end

else begin

if (alw==1 && start\_gm==1) begin

//test<=1'b1;

state<= 5'd0;

case(state)

5'd0:begin

req\_num<= 1;

state<= 5'd1; cout<=1'b0;

end

5'd1:begin // first 7-seg

req\_num<= 0;

hex0<= new\_num^a\_ip;

if (timer\_ip==1) begin

state<= 5'd2;

end

else begin

state<= 5'd1;

end

end

5'd2:begin // second 7-seg

req\_num<= 0;

hex0<= 7'd127;

hex1<= new\_num ^ a\_ip;

if (timer\_ip==1) begin

if ( count== 3'd2) begin

state<= 5'd14;

end\_gm<= 1;

hex1<=7'd126;hex0<= 7'd127;

end

else begin

state<= 5'd4;

end

end

else begin

state<=5'd3;

end

end

5'd3:begin

if (hex1 == trgt\_num) begin

state<= 5'd0; cout<=1'b1;

hex0<= 7'd127; hex1<= 7'd127; hex2<= 7'd127; hex3<= 7'd127; hex4<= 7'd127; hex5<= 7'd127; hex6<= 7'd127;

end

else begin

if (timer\_ip==1) begin

if ( count== 3'd2) begin

state<= 5'd14;

end\_gm<= 1;

hex1<=7'd126;hex0<= 7'd127;

end

else begin

state<= 5'd4;

end

end

else begin

state<= 5'd2;

end

end

end

5'd4:begin // third 7-seg

req\_num<= 0;

hex1<= 7'd127;

hex2<= new\_num ^ a\_ip;

if (timer\_ip==1) begin

if ( count== 3'd3) begin

state<= 5'd0;

count<= count - 3'd1;

hex2<=7'd126;hex0<= 7'd127; hex1<= 7'd127;

end

else begin

state<= 5'd6;

end

end

else begin

state<=5'd5;

end

end

5'd5:begin

if (hex2 == trgt\_num) begin

state<= 5'd0; cout<=1'b1;

hex0<= 7'd127; hex1<= 7'd127; hex2<= 7'd127; hex3<= 7'd127; hex4<= 7'd127; hex5<= 7'd127; hex6<= 7'd127;

end

else begin

if (timer\_ip==1) begin

if ( count== 3'd3) begin

state<= 5'd0;

count<= count - 3'd1;

hex2<=7'd126;hex0<= 7'd127; hex1<= 7'd127;

end

else begin

state<= 5'd6;

end

end

else begin

state<= 5'd4;

end

end

end

5'd6:begin // fourth 7-seg

req\_num<= 0;

hex2<= 7'd127;

hex3<= new\_num ^ a\_ip;

if (timer\_ip==1) begin

if ( count== 3'd4) begin

state<= 5'd0;

count<= count - 3'd1;

hex3<=7'd126;hex0<= 7'd127; hex1<= 7'd127; hex2<= 7'd127;

end

else begin

state<= 5'd8;

end

end

else begin

state<=5'd7;

end

end

5'd7:begin

if (hex3 == trgt\_num) begin

state<= 5'd0; cout<=1'b1;

hex0<= 7'd127; hex1<= 7'd127; hex2<= 7'd127; hex3<= 7'd127; hex4<= 7'd127; hex5<= 7'd127; hex6<= 7'd127;

end

else begin

if (timer\_ip==1) begin

if ( count== 3'd4) begin

state<= 5'd0;

count<= count - 3'd1;

hex3<=7'd126;hex0<= 7'd127; hex1<= 7'd127; hex2<= 7'd127;

end

else begin

state<= 5'd8;

end

end

else begin

state<= 5'd6;

end

end

end

5'd8:begin // fifth 7-seg

req\_num<= 0;

hex3<= 7'd127;

hex4<= new\_num ^ a\_ip;

if (timer\_ip==1) begin

if ( count== 3'd5) begin

state<= 5'd0;

count<= count - 3'd1;

hex4<=7'd126;hex0<= 7'd127; hex1<= 7'd127; hex2<= 7'd127; hex3<= 7'd127;

end

else begin

state<= 5'd10;

end

end

else begin

state<=5'd9;

end

end

5'd9:begin

if (hex4 == trgt\_num) begin

state<= 5'd0; cout<=1'b1;

hex0<= 7'd127; hex1<= 7'd127; hex2<= 7'd127; hex3<= 7'd127; hex4<= 7'd127; hex5<= 7'd127; hex6<= 7'd127;

end

else begin

if (timer\_ip==1) begin

if ( count== 3'd5) begin

state<= 5'd0;

count<= count - 3'd1;

hex4<=7'd126;hex0<= 7'd127; hex1<= 7'd127; hex2<= 7'd127; hex3<= 7'd127;

end

else begin

state<= 5'd10;

end

end

else begin

state<= 5'd8;

end

end

end

5'd10:begin // sixth 7-seg

req\_num<= 0;

hex4<= 7'd127;

hex5<= new\_num ^ a\_ip;

if (timer\_ip==1) begin

if ( count== 3'd6) begin

state<= 5'd0;

count<= count - 3'd1;

hex5<=7'd126;hex0<= 7'd127; hex1<= 7'd127; hex2<= 7'd127; hex3<= 7'd127; hex4<= 7'd127;

end

else begin

state<= 5'd12;

end

end

else begin

state<=5'd11;

end

end

5'd11:begin

if (hex5 == trgt\_num) begin

state<= 5'd0; cout<=1'b1;

hex0<= 7'd127; hex1<= 7'd127; hex2<= 7'd127; hex3<= 7'd127; hex4<= 7'd127; hex5<= 7'd127; hex6<= 7'd127;

end

else begin

if (timer\_ip==1) begin

if ( count== 3'd6) begin

state<= 5'd0;

count<= count - 3'd1;

hex5<=7'd126;hex0<= 7'd127; hex1<= 7'd127; hex2<= 7'd127; hex3<= 7'd127; hex4<= 7'd127;

end

else begin

state<= 5'd12;

end

end

else begin

state<= 5'd10;

end

end

end

5'd12:begin // seventh 7-seg

req\_num<= 0;

hex5<= 7'd127;

hex6<= new\_num ^ a\_ip;

if (timer\_ip==1) begin

count<= count - 1'd1;

state<= 5'd0;

hex6<=7'd126;hex0<= 7'd127; hex1<= 7'd127; hex2<= 7'd127; hex3<= 7'd127; hex4<= 7'd127; hex5<= 7'd127;

end

else begin

state<= 5'd13;

end

end

5'd13:begin

if (hex6 == trgt\_num) begin

state<= 5'd0; cout<=1'b1;

hex0<= 7'd127; hex1<= 7'd127; hex2<= 7'd127; hex3<= 7'd127; hex4<= 7'd127; hex5<= 7'd127; hex6<= 7'd127;

end

else begin

if (timer\_ip==1) begin

state<= 5'd0;

count<= count - 1'd1;

hex6<=7'd126;hex0<= 7'd127; hex1<= 7'd127; hex2<= 7'd127; hex3<= 7'd127; hex4<= 7'd127; hex5<= 7'd127;

end

else begin

state<= 5'd12;

end

end

end

5'd14:begin

if(new\_gm==1 && timer\_ip==1) begin

state<= 5'd0;

count<= 3'd7;

end\_gm<= 0;

hex0<= 7'd127; hex1<= 7'd127; hex2<= 7'd127; hex3<= 7'd127; hex4<= 7'd127; hex5<= 7'd127; hex6<= 7'd127;

end

else begin

state<= 5'd14;

hex0<=7'd127;

end

end

default: begin

end\_gm<= 0;

hex0<= 7'd127; hex1<= 7'd127; hex2<= 7'd127; hex3<= 7'd127; hex4<= 7'd127; hex5<= 7'd127; hex6<= 7'd127;

state<=5'd0; cout<=1'b0;

end

endcase

end

end

end

endmodule

1. Two Player Game Controller: game\_ctl\_2pl.v

//ECE 6370 Advanced Digital Design

//Team: Digital Fortress

//This is the game controller module for the two player mode. Gives a single cycle output pulse //to the score module whenever the pattern matches

module game\_ctl\_2pl(cout, hex0, hex1, hex2, req\_num, end\_gm, a\_ip, trgt\_num, new\_gm, start\_gm, alw, timer\_ip, clk, reset);

input [6:0] a\_ip, trgt\_num;

input new\_gm, start\_gm, alw, timer\_ip, clk, reset;

output reg cout, req\_num, end\_gm;

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

reg [4:0] state;

reg [2:0] count;

reg [6:0] new\_num;

always@(posedge clk)

begin

if(reset==0) begin

count<= 3'd7;

end\_gm<= 0;

new\_num<=7'd0;

hex0<= 7'd127; hex1<= 7'd127; hex2<= 7'd127; cout<=1'b0;

end

else begin

if (alw==1 && start\_gm==1) begin

state<= 5'd0;

case(state)

5'd0:begin

req\_num<= 1;

state<= 5'd1;

cout<=1'b0;

hex0<= 7'd127;// hex1<= 7'd127; hex2<= 7'd127;

end

5'd1:begin // first 7-seg

req\_num<= 0;

hex0<= new\_num^a\_ip;

if (timer\_ip==1) begin

state<= 5'd2;

end

else begin

state<= 5'd1;

end

end

5'd2:begin // second 7-seg

req\_num<= 0;

hex0<= 7'd127;

hex1<= new\_num ^ a\_ip;

if (timer\_ip==1) begin

if ( count== 3'd2) begin

state<= 5'd14;

end\_gm<= 1;

hex1<=7'd126;hex0<= 7'd127;

end

else begin

state<= 5'd4;

end

end

else begin

state<=5'd3;

end

end

5'd3:begin

if (hex1 == trgt\_num) begin

state<= 5'd0; cout<=1'b1;

hex0<= 7'd127; hex1<= 7'd127; hex2<= 7'd127;

end

else begin

if (timer\_ip==1) begin

if ( count== 3'd2) begin

state<= 5'd14;

end\_gm<= 1;

hex1<=7'd126; hex0<= 7'd127;

end

else begin

state<= 5'd4;

end

end

else begin

state<= 5'd2;

end

end

end

5'd4:begin // third 7-seg

req\_num<= 0;

hex1<= 7'd127;

hex2<= new\_num ^ a\_ip;

if (timer\_ip==1) begin

if ( count== 3'd3) begin

state<= 5'd0;

count<= count - 3'd1;

hex2<=7'd126;hex0<= 7'd127; hex1<= 7'd127;

end

else begin

state<= 5'd6;

end

end

else begin

state<=5'd5;

end

end

5'd5:begin

if (hex2 == trgt\_num) begin

state<= 5'd0; cout<=1'b1;

hex0<= 7'd127; hex1<= 7'd127; hex2<= 7'd127;

end

else begin

if (timer\_ip==1) begin

if ( count== 3'd3) begin

state<= 5'd0;

count<= count - 3'd1;

hex2<=7'd126;hex0<= 7'd127; hex1<= 7'd127;

end

else begin

state<= 5'd6;

end

end

else begin

state<= 5'd4;

end

end

end

5'd6:begin // fourth 7-seg

req\_num<= 0;

hex2<= 7'd127;

hex0<= new\_num ^ a\_ip;

if (timer\_ip==1) begin

if ( count== 3'd4) begin

state<= 5'd0;

count<= count - 3'd1;

hex0<=7'd126; hex1<= 7'd127; hex2<= 7'd127;

end

else begin

state<= 5'd8;

end

end

else begin

state<=5'd7;

end

end

5'd7:begin

if (hex0 == trgt\_num) begin

state<= 5'd0; cout<=1'b1;

hex0<= 7'd127; hex1<= 7'd127; hex2<= 7'd127;

end

else begin

if (timer\_ip==1) begin

if ( count== 3'd4) begin

state<= 5'd0;

count<= count - 3'd1;

hex0<=7'd126; hex1<= 7'd127; hex2<= 7'd127;

end

else begin

state<= 5'd8;

end

end

else begin

state<= 5'd6;

end

end

end

5'd8:begin // fifth 7-seg

req\_num<= 0;

hex0<= 7'd127;

hex1<= new\_num ^ a\_ip;

if (timer\_ip==1) begin

if ( count== 3'd5) begin

state<= 5'd0;

count<= count - 3'd1;

hex1<=7'd126; hex0<= 7'd127; hex2<= 7'd127;

end

else begin

state<= 5'd10;

end

end

else begin

state<=5'd9;

end

end

5'd9:begin

if (hex1 == trgt\_num) begin

state<= 5'd0; cout<=1'b1;

hex1<= 7'd127; hex0<= 7'd127; hex2<= 7'd127;

end

else begin

if (timer\_ip==1) begin

if ( count== 3'd5) begin

state<= 5'd0;

count<= count - 3'd1;

hex1<=7'd126; hex2<= 7'd127; hex0<= 7'd127;

end

else begin

state<= 5'd10;

end

end

else begin

state<= 5'd8;

end

end

end

5'd10:begin // sixth 7-seg

req\_num<= 0;

hex1<= 7'd127;

hex2<= new\_num ^ a\_ip;

if (timer\_ip==1) begin

if ( count== 3'd6) begin

state<= 5'd0;

count<= count - 3'd1;

hex2<=7'd126;hex0<= 7'd127; hex1<= 7'd127;

end

else begin

state<= 5'd12;

end

end

else begin

state<=5'd11;

end

end

5'd11:begin

if (hex2 == trgt\_num) begin

state<= 5'd0; cout<=1'b1;

hex0<= 7'd127; hex1<= 7'd127; hex2<= 7'd127;

end

else begin

if (timer\_ip==1) begin

if ( count== 3'd6) begin

state<= 5'd0;

count<= count - 3'd1;

hex2<=7'd126;hex0<= 7'd127; hex1<= 7'd127;

end

else begin

state<= 5'd12;

end

end

else begin

state<= 5'd10;

end

end

end

5'd12:begin // seventh 7-seg

req\_num<= 0;

hex2<= 7'd127;

hex0<= new\_num ^ a\_ip;

if (timer\_ip==1) begin

count<= count - 1'd1;

state<= 5'd0;

hex0<=7'd126;hex2<= 7'd127; hex1<= 7'd127;

end

else begin

state<= 5'd13;

end

end

5'd13:begin

if (hex0 == trgt\_num) begin

state<= 5'd0; cout<=1'b1;

hex0<= 7'd127; hex1<= 7'd127; hex2<= 7'd127;

end

else begin

if (timer\_ip==1) begin

state<= 5'd0;

count<= count - 1'd1;

hex0<=7'd126;hex2<= 7'd127; hex1<= 7'd127;

end

else begin

state<= 5'd12;

end

end

end

5'd14:begin

if(new\_gm==1 && timer\_ip==1) begin

state<= 5'd0;

count<= 3'd7;

end\_gm<= 0;

hex0<= 7'd127; hex1<= 7'd127; hex2<= 7'd127;

end

else begin

state<= 5'd14;

hex0<=7'd127;

end

end

default: begin

end\_gm<= 0;

hex0<= 7'd127; hex1<= 7'd127; hex2<= 7'd127;

state<=5'd0; cout<=1'b0;

end

endcase

end

end

end

endmodule

1. Score counter: SimpleCounter.v

//ECE 6370 Advanced Digital Design

//Team: Digital Fortress

//This module increments the count\_out by one whenever it gets a single cycle count\_in pulse

module SimpleCounter(Count\_in,Clk,Rst,Tout1,Count\_out);

input Count\_in;

input Clk,Rst;

output [5:0] Count\_out;

reg [5:0] Count\_out;

output reg Tout1;

always@(posedge Clk)

begin

if(Rst==1'b0)

begin

Count\_out<=6'd0;

end

else

begin

if(Count\_in==1'b1)

begin

if(Count\_out==6'd20)

begin

Count\_out<=6'd0;

Tout1<=1'b1;

end

else

begin

Tout1<=1'b0;

Count\_out<=Count\_out+6'd1;

end

end

else

begin

Tout1<=1'b0;

end

end

end

endmodule

1. Winner check: win\_checker.v

// ECE6370

// Team: Digital Fortress

//verilog module

//Input:

//1. c\_op\_a: This is score for player a. It is 6-bit input signal.

//2. c\_op\_b: This is score for player b. It is 6-bit input signal.

//3. timeout: This signal is high when all the 7-segments are blocked and the game has ended. It is the //end\_gm signal from the gm\_ctl\_2pl module of player1.

//4. reset: This resets the output to 0 when enabled.

//5. clk: It’s the clock signal

//6. timeout2: This signal is high when all the 7-segments are blocked and the game has ended. It is the //end\_gm signal from the gm\_ctl\_2pl module of player2.

//7. en2: It enables the module only when two-player mode game is selected. //

//Outputs:

//1. a\_rocks: This signal is high when player a is the winner.

//2. b\_rocks: This signal is high when player b is the winner.

//

//

//

//

module win\_checker (en2,a\_rocks, b\_rocks, end\_gm\_led, c\_op\_a, c\_op\_b, timeout,timeout2,clk, rst);

//Declare inputs,outputs and internal variables

input [5:0] c\_op\_a, c\_op\_b;

input clk, rst, timeout,timeout2,en2;

output a\_rocks, b\_rocks,end\_gm\_led;

reg a\_rocks, b\_rocks,end\_gm\_led;

//Procedure description to decide the winner at the end of the game

always @ (posedge clk) begin

if (rst==0) begin

a\_rocks<=1'b0;

b\_rocks<=1'b0;

end\_gm\_led<=1'b0;

end

else begin

if ((timeout==1'b1 || timeout2==1'b1 ||c\_op\_a==6'd20 || c\_op\_b==6'd20) && en2==1'b1) begin

if(c\_op\_a>c\_op\_b) begin

a\_rocks<=1'b1;

end\_gm\_led<=1'b1;

b\_rocks<=1'b0;

end

else if(c\_op\_b>c\_op\_a) begin

b\_rocks<=1'b1;

end\_gm\_led<=1'b1;

a\_rocks<=1'b0;

end

else begin

a\_rocks<=1'b1;

b\_rocks<=1'b1;

end\_gm\_led<=1'b1;

end

end

else

begin

a\_rocks<=1'b0;

b\_rocks<=1'b0;

end\_gm\_led<=1'b0;

end

end

end

endmodule

1. BCD score conversion: bin2bcd\_score.v

// ECE6370

// Team : Digital Fortress

//verilog module

//bin is the 4-bit binary input,

//tens is the 4-bit BCD value for tens place value, ones is the 4-bit BCD value for ones place value

module bin2bcd\_score (tens, ones, bin);

//Declare inputs,outputs and internal variables

input [5:0] bin;

output [3:0] tens;

output [3:0] ones;

reg [3:0] tens;

reg [3:0] ones;

reg [13:0] shift;

integer i;

//Procedure description to binary to BCD to display score

always @ (bin)

begin

// Clear previous number and store new number in shift register

shift[13:4] = 12'd0;

shift[5:0] = bin;

// Loop ten times

for (i=0; i<6; i=i+1) begin

if (shift[9:6] >= 5)

shift[9:6] = shift[9:6] + 3;

if (shift[13:10] >= 5)

shift[13:10] = shift[13:10] + 3;

// Shift entire register left once

shift = shift << 1;

end

// Push decimal numbers to output

tens = shift[13:10];

ones = shift[9:6];

end

endmodule

1. ASCII conversion: ASCII\_decoder.v

// ECE6370

// Team : Digital Fortress

//This module converts the input ASCII digit into the upper\_nibble and lower\_nibble

module ASCII\_decoder (count, upper\_nibble, lower\_nibble);

input [3:0] count;

output [5:0] upper\_nibble, lower\_nibble;

reg [5:0] upper\_nibble, lower\_nibble;

always @ (count) begin

case(count)

4'b0000:begin

upper\_nibble= 6'h23;

lower\_nibble= 6'h20;

end

4'b0001:begin

upper\_nibble= 6'h23;

lower\_nibble= 6'h21;

end

4'b0010:begin

upper\_nibble= 6'h23;

lower\_nibble= 6'h22;

end

4'b0011 : begin

upper\_nibble= 6'h23;

lower\_nibble= 6'h23;

end

4'b0100:begin

upper\_nibble= 6'h23;

lower\_nibble= 6'h24;

end

4'b0101: begin

upper\_nibble= 6'h23;

lower\_nibble= 6'h25;

end

4'b0110: begin

upper\_nibble= 6'h23;

lower\_nibble= 6'h26;

end

4'b0111: begin

upper\_nibble= 6'h23;

lower\_nibble= 6'h27;

end

4'b1000: begin

upper\_nibble= 6'h23;

lower\_nibble= 6'h28;

end

4'b1001: begin

upper\_nibble= 6'h23;

lower\_nibble= 6'h29;

end

default:begin

upper\_nibble= 6'h23;

lower\_nibble= 6'h2F;

end

endcase

end

endmodule

1. One player LCD Display: LCD\_display\_1pl.v

//ECE 6370 Advanced Digital Design

//Team: Digital Fortress

//This module is the LCD display module for one player mode

// synthesis attribute STEPPING top "ES"

module LCD\_display\_1pl (clk, en, upper\_nibble\_ones1, lower\_nibble\_ones1, upper\_nibble\_tens1, lower\_nibble\_tens1, lcd\_rs, lcd\_rw, lcd\_e, lcd\_4, lcd\_5, lcd\_6, lcd\_7);

parameter n = 27;

parameter k = 17;

input clk, en;

input [5:0] lower\_nibble\_ones1, upper\_nibble\_ones1, upper\_nibble\_tens1, lower\_nibble\_tens1;

reg [n-1:0] count=0;

reg lcd\_busy=1;

reg lcd\_stb;

reg [5:0] lcd\_code,abc;

reg [6:0] lcd\_stuff;

output reg lcd\_rs;

output reg lcd\_rw;

output reg lcd\_7;

output reg lcd\_6;

output reg lcd\_5;

output reg lcd\_4;

output reg lcd\_e;

always @ (posedge clk) begin

if (en==1'b1) begin

count <= count + 1;

case (count[k+8:k+2])

0: lcd\_code <= 6'b000010; // function set

1: lcd\_code <= 6'b000010;

2: lcd\_code <= 6'b001100;

3: lcd\_code <= 6'b000000; // display on/off control

4: lcd\_code <= 6'b001100;

5: lcd\_code <= 6'b000000; // display clear

6: lcd\_code <= 6'b000001;

7: lcd\_code <= 6'b000000; // entry mode set

8: lcd\_code <= 6'b000110;

9: lcd\_code <= 6'h25; // S

10: lcd\_code <= 6'h23;

11: lcd\_code <= 6'h24; // C

12: lcd\_code <= 6'h23;

13: lcd\_code <= 6'h24; // O

14: lcd\_code <= 6'h2f;

15: lcd\_code <= 6'h25; // R

16: lcd\_code <= 6'h22;

17: lcd\_code <= 6'h24; // E

18: lcd\_code <= 6'h25;

19: lcd\_code <= 6'h22; //

20: lcd\_code <= 6'h20;

21: lcd\_code <= 6'b001100; // first position of next line

22: lcd\_code <= 6'b000000;

23: lcd\_code <= 6'h25; // P

24: lcd\_code <= 6'h20;

25: lcd\_code <= 6'h23; // 1

26: lcd\_code <= 6'h21;

27: lcd\_code <= 6'h23; // :

28: lcd\_code <= 6'h2A;

29: lcd\_code <= upper\_nibble\_tens1; // 0

30: lcd\_code <= lower\_nibble\_tens1;

31: lcd\_code <= upper\_nibble\_ones1; // 0

32: lcd\_code <= lower\_nibble\_ones1;

33: lcd\_code <= 6'h22; //

34: lcd\_code <= 6'h20;

35: lcd\_code <= 6'h22; //

36: lcd\_code <= 6'h20;

37: lcd\_code <= 6'h22; //

38: lcd\_code <= 6'h20;

39: lcd\_code <= 6'h22; //

40: lcd\_code <= 6'h20;

41: lcd\_code <= 6'h22; //

42: lcd\_code <= 6'h20;

43: lcd\_code <= 6'h22; //

44: lcd\_code <= 6'h20;

45: lcd\_code <= 6'h22; //

46: lcd\_code <= 6'h20;

47: lcd\_code <= 6'h22; //

48: lcd\_code <= 6'h20;

49: lcd\_code <= 6'h22; //

50: lcd\_code <= 6'h20;

51: lcd\_code <= 6'h22; //

52: begin lcd\_code <= 6'h20; count[k+8:k+2]<= 6'd21; end

default: lcd\_code <= 6'b010000;

endcase

if (lcd\_rw)

lcd\_busy <= 0;

lcd\_stb <= ^count[k+1:k+0] & ~lcd\_rw & lcd\_busy; // clkrate / 2^(k+2)

lcd\_stuff <= {lcd\_stb,lcd\_code};

{lcd\_e,lcd\_rs,lcd\_rw,lcd\_7,lcd\_6,lcd\_5,lcd\_4} <= lcd\_stuff;

end

else

begin

count[k+8:k+2]<= 6'd0;

end

end

endmodule

1. Two player LCD Display: LCD\_display.v

//ECE 6370 Advanced Digital Design

//Team: Digital Fortress

//This module is the LCD display module for two player mode

// synthesis attribute STEPPING top "ES"

module LCD\_display (clk, en, upper\_nibble\_ones1, lower\_nibble\_ones1, upper\_nibble\_tens1, lower\_nibble\_tens1, upper\_nibble\_ones2, lower\_nibble\_ones2, upper\_nibble\_tens2, lower\_nibble\_tens2, lcd\_rs, lcd\_rw, lcd\_e, lcd\_4, lcd\_5, lcd\_6, lcd\_7);

parameter n = 27;

parameter k = 17;

input clk, en;

input [5:0] lower\_nibble\_ones1, upper\_nibble\_ones1, upper\_nibble\_tens1, lower\_nibble\_tens1, upper\_nibble\_ones2, lower\_nibble\_ones2, upper\_nibble\_tens2, lower\_nibble\_tens2;

reg [n-1:0] count=0;

reg lcd\_busy=1;

reg lcd\_stb;

reg [5:0] lcd\_code,abc;

reg [6:0] lcd\_stuff;

output reg lcd\_rs;

output reg lcd\_rw;

output reg lcd\_7;

output reg lcd\_6;

output reg lcd\_5;

output reg lcd\_4;

output reg lcd\_e;

always @ (posedge clk) begin

if(en==1'b1) begin

count <= count + 1;

case (count[k+8:k+2])

0: lcd\_code <= 6'b000010; // function set

1: lcd\_code <= 6'b000010;

2: lcd\_code <= 6'b001100;

3: lcd\_code <= 6'b000000; // display on/off control

4: lcd\_code <= 6'b001100;

5: lcd\_code <= 6'b000000; // display clear

6: lcd\_code <= 6'b000001;

7: lcd\_code <= 6'b000000; // entry mode set

8: lcd\_code <= 6'b000110;

9: lcd\_code <= 6'h25; // S

10: lcd\_code <= 6'h23;

11: lcd\_code <= 6'h24; // C

12: lcd\_code <= 6'h23;

13: lcd\_code <= 6'h24; // O

14: lcd\_code <= 6'h2f;

15: lcd\_code <= 6'h25; // R

16: lcd\_code <= 6'h22;

17: lcd\_code <= 6'h24; // E

18: lcd\_code <= 6'h25;

19: lcd\_code <= 6'h22; //

20: lcd\_code <= 6'h20;

21: lcd\_code <= 6'b001100; // first position of next line

22: lcd\_code <= 6'b000000;

23: lcd\_code <= 6'h25; // P

24: lcd\_code <= 6'h20;

25: lcd\_code <= 6'h23; // 1

26: lcd\_code <= 6'h21;

27: lcd\_code <= 6'h23; // :

28: lcd\_code <= 6'h2A;

29: lcd\_code <= upper\_nibble\_tens1; // 0

30: lcd\_code <= lower\_nibble\_tens1;

31: lcd\_code <= upper\_nibble\_ones1; // 0

32: lcd\_code <= lower\_nibble\_ones1;

33: lcd\_code <= 6'h22; //

34: lcd\_code <= 6'h20;

35: lcd\_code <= 6'h22; //

36: lcd\_code <= 6'h20;

37: lcd\_code <= 6'h22; //

38: lcd\_code <= 6'h20;

39: lcd\_code <= 6'h25; // P

40: lcd\_code <= 6'h20;

41: lcd\_code <= 6'h23; // 2

42: lcd\_code <= 6'h22;

43: lcd\_code <= 6'h23; // :

44: lcd\_code <= 6'h2A;

45: lcd\_code <= upper\_nibble\_tens2; // 0

46: lcd\_code <= lower\_nibble\_tens2;

47: lcd\_code <= upper\_nibble\_ones2; // 0

48: lcd\_code <= lower\_nibble\_ones2;

49: lcd\_code <= 6'h22; //

50: lcd\_code <= 6'h20;

51: lcd\_code <= 6'h22; //

52: lcd\_code <= 6'h20;

53: lcd\_code <= 6'h22; //

54: lcd\_code <= 6'h20;

55: lcd\_code <= 6'h22; //

56: begin lcd\_code <= 6'h20; count[k+8:k+2]<= 6'd21; end

default: lcd\_code <= 6'b010000;

endcase

if (lcd\_rw)

lcd\_busy <= 0;

lcd\_stb <= ^count[k+1:k+0] & ~lcd\_rw & lcd\_busy; // clkrate / 2^(k+2)

lcd\_stuff <= {lcd\_stb,lcd\_code};

{lcd\_e,lcd\_rs,lcd\_rw,lcd\_7,lcd\_6,lcd\_5,lcd\_4} <= lcd\_stuff;

end

else

begin

count[k+8:k+2]<= 6'd0;

end

end

endmodule

1. RAM.v

//ECE 6370 Advanced Digital Design

//Team: Digital Fortress

//This is the RAM initialization module

// 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!

//

// 16.1.0 Build 196 10/24/2016 SJ Lite Edition

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

//Copyright (C) 2016  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 MegaCore Function 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     [2:0]  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 = "RAM\_Score\_init.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 = 8,

                             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 = 3,

                             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 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 "RAM\_Score\_init.mif"

// Retrieval info: PRIVATE: NUMWORDS\_A NUMERIC "8"

// 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 "3"

// 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 "RAM\_Score\_init.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 "8"

// 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 "3"

// Retrieval info: CONSTANT: WIDTH\_A NUMERIC "8"

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

// Retrieval info: USED\_PORT: address 0 0 3 0 INPUT NODEFVAL "address[2..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 3 0 address 0 0 3 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

1. **RAM\_Controller module:**

//ECE 6370 Advanced Digital Design

//Team: Digital Fortress

//This is the ram controller module with rdwr\_in, id, d\_ram, data\_in as input and rdwr, addr, d\_in, d\_out as ouputs

module RAM\_Controller(clk,rst,rdwr\_in, id,data\_in,d\_ram,rdwr,addr,d\_in,d\_out);

input clk,rst;

input rdwr\_in;

input [2:0]id;

input [7:0]d\_ram, data\_in;

output reg rdwr;

output reg [2:0]addr;

output reg [7:0]d\_in;

output reg [7:0]d\_out;

reg ram\_init\_flag;

reg [3:0]State;

reg [7:0]d\_ram\_reg;

parameter INIT=4'd0, READ= 4'd1, WAIT1 = 4'd2, WAIT2= 4'd3,READING=4'd4, OUT=4'd5, WRITE=4'd6, WAIT3=4'd7, WAIT4=4'd8, WRITING=4'd9, END=4'd10;

always@(posedge clk)

begin

 if(rst==0)

 begin

  addr<=3'd0;

  d\_in<=8'd0;

  d\_out<=8'd0;

  ram\_init\_flag<=1'b0;

  State<=INIT;

  rdwr<=0;

 end

 else

 begin

  case(State)

  INIT:begin

                             if(rdwr\_in==1'b0)

                                           begin

                                                          State<=READ;

                                           end

                             else

                                           begin

                                                          State<=WRITE;

                                                          ram\_init\_flag<=1'b1;

                                           end

                             end

  READ:begin

                                           addr<= id;

                                           rdwr<=0;

                                           State<= WAIT1;

                              end

  WAIT1: begin

                                          State<=WAIT2;

                                end

  WAIT2: begin

                                                          State<=READING;

                                end

READING: begin

                                    d\_ram\_reg<= d\_ram;

                                                           State<= OUT;

                                end

              OUT: begin

                                    d\_out<= d\_ram\_reg;

             State<=END;

                                  end

  WRITE: begin

                                                          addr<=id;

                                                          d\_in<=data\_in;

                                                          rdwr<=1;

                                                          State<= WAIT3;

                                            end

   WAIT3: begin

                                          State<=WAIT4;

                                end

  WAIT4: begin

                                                          State<=WRITING;

                                end

              WRITING: begin

                                                            State<= END;

                                                          end

              END:  begin

                       State<= INIT;

                                           end

              default: begin

                         State<= INIT;

                                                            addr<=3'd0;

                                                            d\_in<=8'd0;

                                                            d\_out<=8'd0;

                                                          end

                             endcase

                             end

              end

endmodule

**RAM\_TOP :**

//ECE 6370 Advanced Digital Design

//Team: Digital Fortress

//This is the top module for RAM consisting of instances of RAM\_Controller and RAM

module RAM\_TOP(clk,rst,rdwr\_in,id,data\_in,d\_out);

              input     [2:0]id;

              input     clk, rst, rdwr\_in;

   input [7:0]data\_in;

              output [7:0]d\_out;

              wire rdwr;

              wire [7:0]d\_ram,d\_in;

              wire [2:0]addr;

RAM\_Controller RC1(clk,rst,rdwr\_in, id,d\_ram,data\_in,rdwr,addr,d\_in,d\_out);

RAM RAM1(addr,clk,d\_in,rdwr,d\_ram);

Endmodule

1. RAM\_Score.v

//ECE 6370 Advanced Digital Design

//Team: Digital Fortress

// This module updates the highest score of each player

module RAM\_Score( clk,rst, end\_game,score\_ram, user\_id1, user\_idp1, user\_idp2, score1,scorep1,scorep2, en1,en2, rdwr\_in, data\_to\_ram, user\_id );

input clk,rst, en1,en2,end\_game;

input [7:0]score\_ram;

input [5:0]score1,scorep1,scorep2;

input [2:0]user\_id1, user\_idp1, user\_idp2;

output reg rdwr;

output reg [7:0] data\_to\_ram;

output reg [2:0]user\_id;

reg [2:0] State;

parameter INIT=3'd0;St1=3'd1,St2=3'd2,St3=3'd3,St4=3'd4,St5=3'd5,St6=3'd6,St7=3'd7;

always@(posedge clk) begin

if (rst==0) begin

rdwr<=0;

data\_to\_ram<=8'd0;

user\_id<= 3'd0;

end

else begin

case(State)

INIT: begin

if(end\_game==1) begin

State<= St1;

end

else begin

State<= INIT;

end

end

St1: begin

if(en1==1) begin

rdwr<=0;

cycle<=2'd0;

user\_id<= user\_id1;

State<= St2;

end

else begin

rdwr<=0;

cycle<=2'd0;

user\_id<= user\_idp1;

State<= St4;

end

St2: begin

if(cycle<2'd3) begin

cycle<=cycle+1'd1;

State<=St2;

end

else begin

if(score1> score\_ram[5:0]) begin

rdwr<=1;

user\_id<= user\_id1;

State<= St3;

end

St3: begin

data\_to\_ram<= {0,0,score1[5:0]};

State<=END;

end

St4: begin

if(cycle<2'd3) begin

cycle<=cycle+1'd1;

State<=St4;

end

else begin

if(scorep1> score\_ram[5:0]) begin

rdwr<=1;

user\_id<= user\_id1;

State<= St5;

cycle<= 2'd0;

end

St5: begin

data\_to\_ram<= {0,0,scorep1[5:0]};

State<=St6;

end

St6: begin

if(cycle<2'd3) begin

cycle<=cycle+1'd1;

State<=St6;

end

else begin

if(scorep1> score\_ram[5:0]) begin

rdwr<=1;

user\_id<= user\_id1;

State<= St7;

cycle<= 2'd0;

end

St7: begin

data\_to\_ram<= {0,0,scorep2[5:0]};

State<=END;

end

END: begin

State<=END;

end

endcase

end

end

endmodule

1. **Top module: Final\_projetc\_DigitalFortress.v**

//ECE 6370 Advanced Digital Design

//Team: Digital Fortress

// This is the top module of our final project

module top\_test( allow1,allowp2,user\_ip,load,user\_id\_out1,user\_id\_outp1,user\_id\_outp2,en1, en2, a\_rocks, b\_rocks,end\_gm\_led, hex0\_disp, hex1\_disp, hex2\_disp, hex3\_disp, hex4\_disp, hex5\_disp, hex6\_disp, hex7\_disp, lcd\_rs\_top, lcd\_rw\_top, lcd\_e\_top, lcd\_4\_top, lcd\_5\_top, lcd\_6\_top, lcd\_7\_top, a\_ip, b\_ip, b\_in\_new\_gm, b\_in\_mode, clk, rst);

input [6:0] a\_ip, b\_ip;

input b\_in\_new\_gm, b\_in\_mode, clk, rst;

//input [1:0] level;

input load;

input [3:0]user\_ip;

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

output lcd\_rs\_top, lcd\_rw\_top, lcd\_e\_top, lcd\_4\_top, lcd\_5\_top, lcd\_6\_top, lcd\_7\_top;

output en1, en2,a\_rocks, b\_rocks,end\_gm\_led;

output [2:0]user\_id\_out1,user\_id\_outp1,user\_id\_outp2;

output allow1,allowp2;

wire pulse, en1, en2, new\_gm;

wire allow1,allowp2;

//input [2:0] level;

wire [6:0] hex0\_pl1, hex1\_pl1, hex2\_pl1, hex3\_pl1, hex4\_pl1, hex5\_pl1, hex6\_pl1;

wire [6:0] RN1\_pl1;

wire lcd\_rs\_pl1, lcd\_rw\_pl1, lcd\_e\_pl1, lcd\_4\_pl1, lcd\_5\_pl1, lcd\_6\_pl1, lcd\_7\_pl1, end\_gm\_pl1;

wire [15:0] q1\_pl1, q2\_pl1;

wire P1\_pl1, P2\_pl1;

wire [6:0] RN2\_pl1;

wire [5:0] Score1\_pl1;

wire timer\_ip\_pl1, Request\_pl1, cout1\_pl1, cout2\_pl1, Tout1\_p1\_pl1;

wire [3:0] tens1\_pl1, ones1\_pl1;

wire [5:0] upper\_nibble\_ones1\_pl1, lower\_nibble\_ones1\_pl1;

wire [5:0] upper\_nibble\_tens1\_pl1, lower\_nibble\_tens1\_pl1;

//input [6:0] a\_ip, b\_ip;

//input new\_gm, clk, rst;

wire [6:0] hex0, hex1, hex2, hex4, hex5, hex6;

wire [6:0] RN1,RN2;

wire lcd\_rs, lcd\_rw,lcd\_e, lcd\_4, lcd\_5, lcd\_6, lcd\_7;

wire [15:0] q1, q2;

//wire [6:0] RN2;

wire [5:0] Score1, Score2;

//input [1:0] level;

wire P1, P2, timer\_ip, Request1, Request2, cout1, cout2, end\_gm1, end\_gm2, Tout1\_p1, Tout1\_p2;

wire [3:0] tens1, ones1, tens2, ones2;

wire [5:0] upper\_nibble\_ones1, lower\_nibble\_ones1, upper\_nibble\_ones2, lower\_nibble\_ones2;

wire [5:0] upper\_nibble\_tens1, lower\_nibble\_tens1, upper\_nibble\_tens2, lower\_nibble\_tens2;

//Player mode select

mode\_select mode\_sel(pulse, clk, rst, en1, en2);

button\_shaper bs\_mode(b\_in\_mode, pulse, clk, rst);

//new game input

button\_shaper bs\_new\_gm(b\_in\_new\_gm, new\_gm, clk, rst);

//display select

op\_select op\_sel(hex0\_disp, hex1\_disp, hex2\_disp, hex3\_disp, hex4\_disp, hex5\_disp, hex6\_disp, hex7\_disp, lcd\_rs\_top, lcd\_rw\_top, lcd\_e\_top, lcd\_4\_top, lcd\_5\_top, lcd\_6\_top, lcd\_7\_top, en1, en2, hex0\_pl1, hex1\_pl1, hex2\_pl1, hex3\_pl1, hex4\_pl1, hex5\_pl1, hex6\_pl1, RN1\_pl1, lcd\_rs\_pl1, lcd\_rw\_pl1, lcd\_e\_pl1, lcd\_4\_pl1, lcd\_5\_pl1, lcd\_6\_pl1, lcd\_7\_pl1, lcd\_rs, lcd\_rw, lcd\_e, lcd\_4, lcd\_5, lcd\_6, lcd\_7, hex0, hex1, hex2, RN2, hex4, hex5, hex6, RN1, clk, rst);

//////////////////////////////////////////////////////////////////////

//

//ONE PLAYER MODE

//////////////////////////////////////////////////////////////////////

//target\_num generator

LFSR1 N1\_pl1(clk, rst, q1\_pl1);

LoadReg LoadRN1\_pl1({q1\_pl1[3], q1\_pl1[10], q1\_pl1[9], q1\_pl1[2], q2\_pl1[1], q2\_pl1[6], q2\_pl1[5]}, en1, clk, rst, Request\_pl1, RN1\_pl1); //target\_sequence

//new\_num generator

LFSR2 N2\_pl1(clk, rst, q2\_pl1);

LoadReg LoadRN2\_pl1({q2\_pl1[2], q2\_pl1[4], q2\_pl1[1], q2\_pl1[9], q2\_pl1[8], q2\_pl1[15], q2\_pl1[12]}, en1, clk, rst, Request\_pl1, RN2\_pl1); //hex0\_sequence

//timer

LFSR\_1ms MS1\_pl1(clk, rst, en1, P1\_pl1);

LFSR\_100ms MS2\_pl1(clk, rst, P1\_pl1, P2\_pl1);

//nSecPulseGenerator NS\_pl1(P2\_pl1, level\_pl1, clk, rst, timer\_ip\_pl1);

nSecPulseGenerator NS\_pl1(P2\_pl1, {1'b0, level1}, clk, rst, timer\_ip\_pl1);

//LevelSelector LS1\_pl1(Score\_pl1, clk, rst, level\_pl1);

//one player game

game\_ctl game\_ctl1\_pl1(cout1\_pl1, hex0\_pl1, hex1\_pl1, hex2\_pl1, hex3\_pl1, hex4\_pl1, hex5\_pl1, hex6\_pl1, Request\_pl1, end\_gm\_pl1, RN2\_pl1, a\_ip

, RN1\_pl1, new\_gm, en1,allow1, timer\_ip\_pl1, clk, rst);

//one player scoring

SimpleCounter SC1\_p1\_pl1(cout1\_pl1, clk, rst, Tout1\_p1\_pl1, Score1\_pl1);

bin2bcd\_score bcd\_score1\_p1\_pl1(tens1\_pl1, ones1\_pl1, Score1\_pl1);

ASCII\_decoder ASCII\_ones\_p1\_pl1(ones1\_pl1, upper\_nibble\_ones1\_pl1, lower\_nibble\_ones1\_pl1);

ASCII\_decoder ASCII\_tens\_p1\_pl1(tens1\_pl1, upper\_nibble\_tens1\_pl1, lower\_nibble\_tens1\_pl1);

LCD\_display\_1pl LCD\_disp\_1\_pl1(clk, en1, upper\_nibble\_ones1\_pl1, lower\_nibble\_ones1\_pl1, upper\_nibble\_tens1\_pl1, lower\_nibble\_tens1\_pl1, lcd\_rs\_pl1, lcd\_rw\_pl1, lcd\_e\_pl1, lcd\_4\_pl1, lcd\_5\_pl1, lcd\_6\_pl1, lcd\_7\_pl1);

//////////////////////////////////////////////////////////////////////

//

//TWO PLAYER MODE

//////////////////////////////////////////////////////////////////////

//ROM MODULE

ROM\_2AC AC2(clk,rst,en1,en2,load,user\_ip, allow1,allowp2, user\_id\_out1,user\_id\_outp1,user\_id\_outp2);

//target\_num generator

LFSR1 N1(clk, rst, q1);

LoadReg LoadRN1({q1[3], q1[10], q1[9], q1[2], q2[1], q2[6], q2[5]}, en2, clk,rst,Request1,RN1); //target\_sequence

LFSR1 N2(clk, rst, q2);

LoadReg LoadRN2({q2[3], q2[10], q2[9], q2[2], q1[1], q1[6], q1[5]}, en2, clk,rst,Request2,RN2); //target\_sequence

//timer

LFSR\_1ms MS1(clk, rst, en2, P1);

LFSR\_100ms MS2(clk, rst, P1, P2);

nSecPulseGenerator NS(P2, {1'b0, level2}, clk, rst, timer\_ip);

LevelSelector LS1(Score1\_pl1, clk, rst, level1);

LevelSelector LS2(Score1, clk, rst, level2);

//two player game

game\_ctl\_2pl game\_ctl\_2pl\_1(cout1, hex4, hex5, hex6, Request1, end\_gm1, a\_ip, RN1, new\_gm, en2, allowp2, timer\_ip, clk, rst);

game\_ctl\_2pl game\_ctl\_2pl\_2(cout2, hex0, hex1, hex2, Request2, end\_gm2, b\_ip, RN2, new\_gm, en2, allowp2, timer\_ip, clk, rst);

//player\_1 scoring

SimpleCounter SC1\_p1(cout1, clk, rst, Tout1\_p1, Score1);

bin2bcd\_score bcd\_score1\_p1(tens1, ones1, Score1);

ASCII\_decoder ASCII\_ones\_p1(ones1, upper\_nibble\_ones1, lower\_nibble\_ones1);

ASCII\_decoder ASCII\_tens\_p1(tens1, upper\_nibble\_tens1, lower\_nibble\_tens1);

//player\_2 scoring

SimpleCounter SC1\_p2(cout2, clk, rst, Tout1\_p2, Score2);

bin2bcd\_score bcd\_score1\_p2(tens2, ones2, Score2);

ASCII\_decoder ASCII\_ones\_p2(ones2, upper\_nibble\_ones2, lower\_nibble\_ones2);

ASCII\_decoder ASCII\_tens\_p2(tens2, upper\_nibble\_tens2, lower\_nibble\_tens2);

LCD\_display LCD\_disp1(clk, en2, upper\_nibble\_ones1, lower\_nibble\_ones1, upper\_nibble\_tens1, lower\_nibble\_tens1, upper\_nibble\_ones2, lower\_nibble\_ones2, upper\_nibble\_tens2, lower\_nibble\_tens2, lcd\_rs, lcd\_rw, lcd\_e, lcd\_4, lcd\_5, lcd\_6, lcd\_7);

win\_checker win (en2, a\_rocks, b\_rocks,end\_gm\_led, Score1, Score2, end\_gm1,end\_gm2,clk, rst);

endmodule