Bangladesh University of Engineering & Technology

Project Report

Course No: EEE 304

Course Name: Digital Electronics Lab

Project Name:

Ladder-Snake Game

Group Detail (Names & IDs):

SM Haider Ali Shuvo	(1606134)
Istahaque Rahman Hasib	(1606139)
Shakil Ahmed	(1606140)

Section: C1

Department: EEE

Google Drive Link: https://bit.ly/3pawLKK[1]

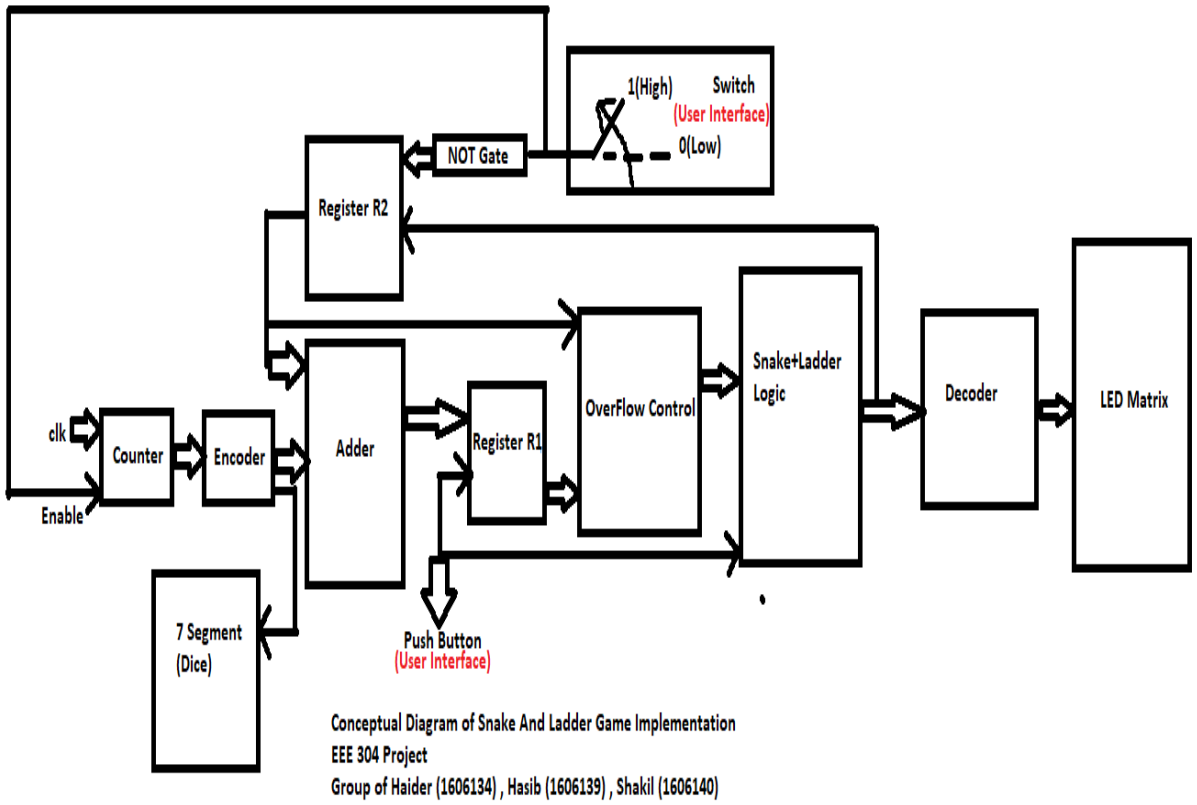
Or https://drive.google.com/drive/folders/1e7B9dOkzT3GR4d7mcECgwO6GkECRyU7i?usp=sharing

[1] Please copy & paste the drive link in your browser.

Table of Contents

Main Concept

In this section, the main concept's to achieve a Ladder-Snake game functionality is discussed. For simplicity, we will discuss here the building blocks to achieve Ladder-Snake functionality for only a single player. The necessary changes to add two players feature will be discussed in Verilog Implementation Section .*The conceptual diagram is shown below.*



There are two user interface Here- the Push Button and the Switch. The Switch is a SPDT switch connected to high and low logic in its double throw part. ***To play the game, User needs to operate in the following sequence –***

1. Take switch into Low state from high state to make the dice rolling.

2. Take switch from High to Low state to fix the dice to a value between (0-6).

3. Operate the push button once to make corresponding move in 4x4 LED Matrix.

Proteus Implementation

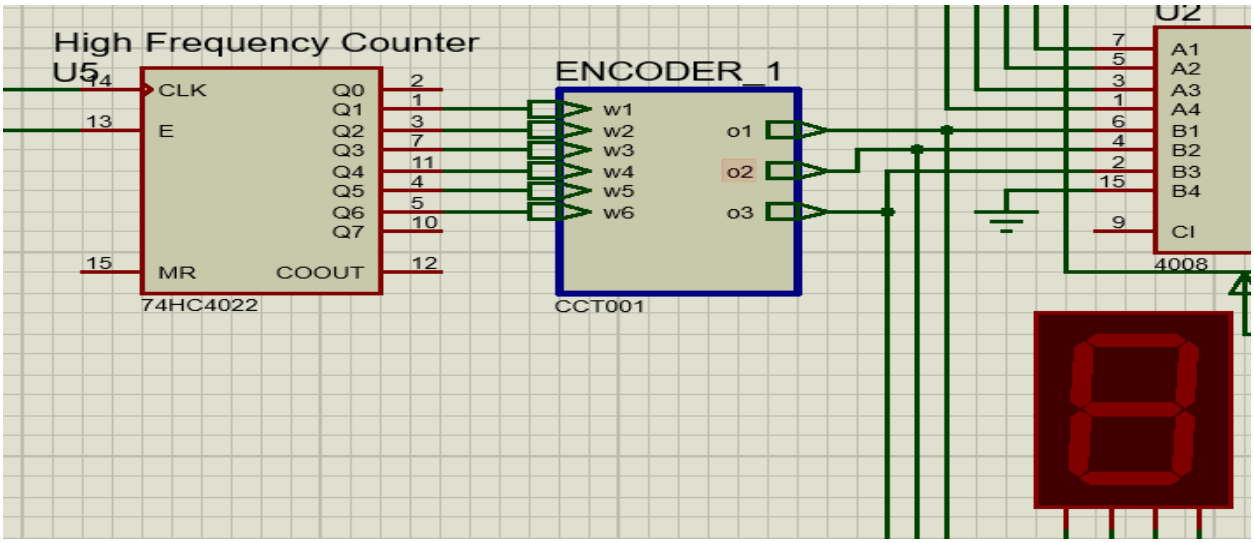
*We will break down the **working principle** now along with the **Proteus Implementation.***

1. Counter + Encoder

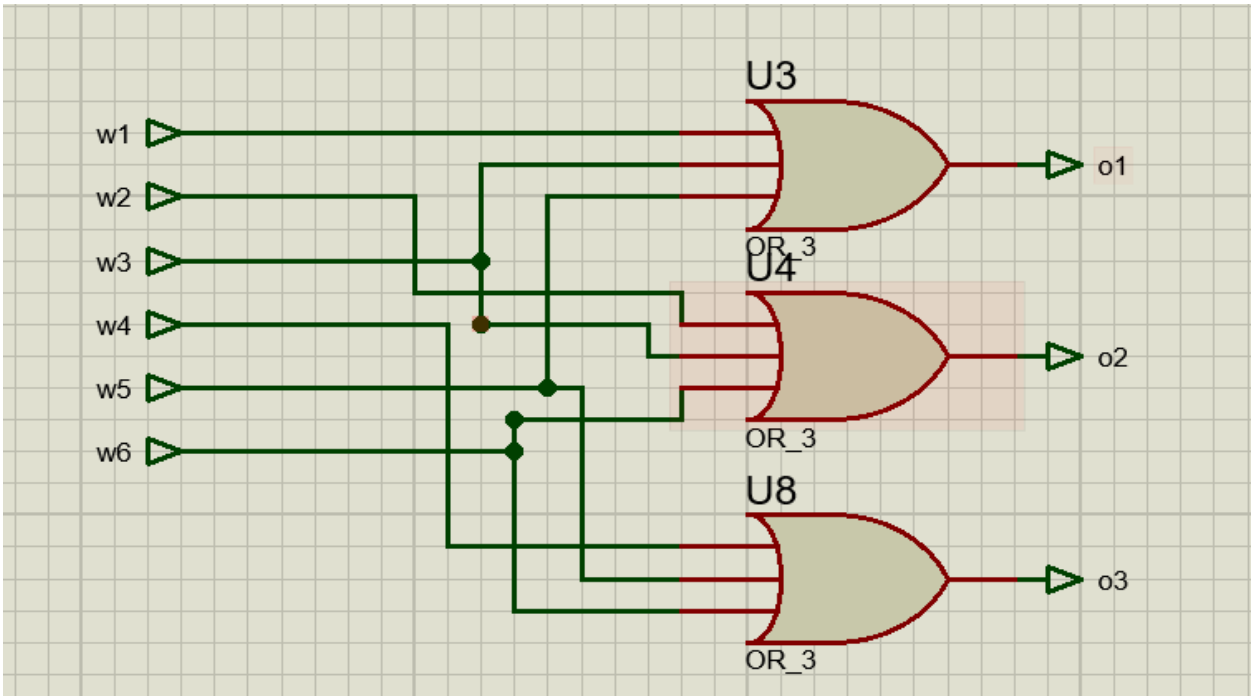
The switch is connected to the High Frequency Johnson Counter as its Enable Pin. So, When switch is on High State, the enable pin of Counter is High and Counter runs at very high frequency. When Switch turns low, Enable Pin turns low, And Johnson counter is fixed to a random State. The output of counter is passed to an Encoder which encodes the random state to a 3 bit binary number corresponding between Deciman 0 to 6. The Dice state which comes from the Encoder is shown live to a 7 segment Display.

In proteus implementation, **74HC4022 IC** is used as the **counter** whose clock is manually set to very high frequency clock. To achieve **Encoder** functionality we make a **custom Sub circuit** according to our preference.

The counter+Encoeder part is shown below –



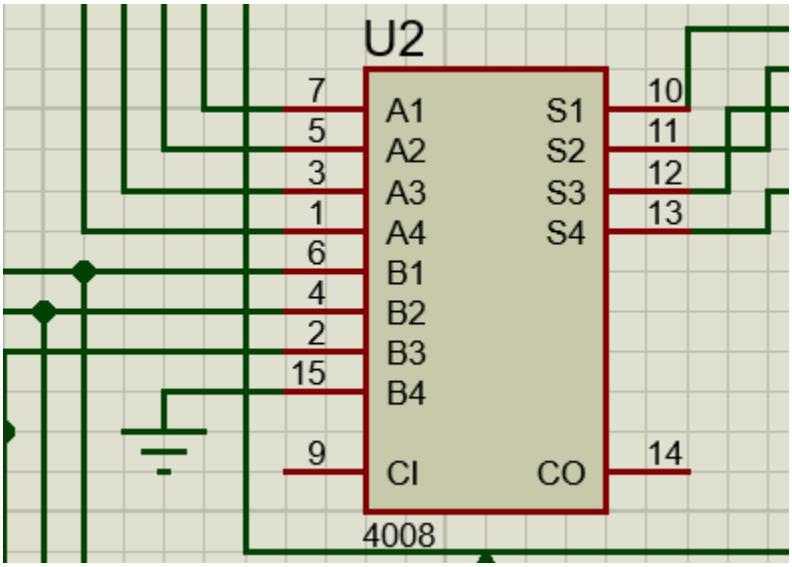
The Circuit inside Encoder is shown below –



2. Adder

Adder Produces 3 bit binary dice with the 4 bit binary state of Register R2. Register R2 holds the past state of the game i.e. which position the user in the LED matrix.

To achieve adder functionality, we use IC 4008 4'bit binary adder. The adder circuit is shown below.



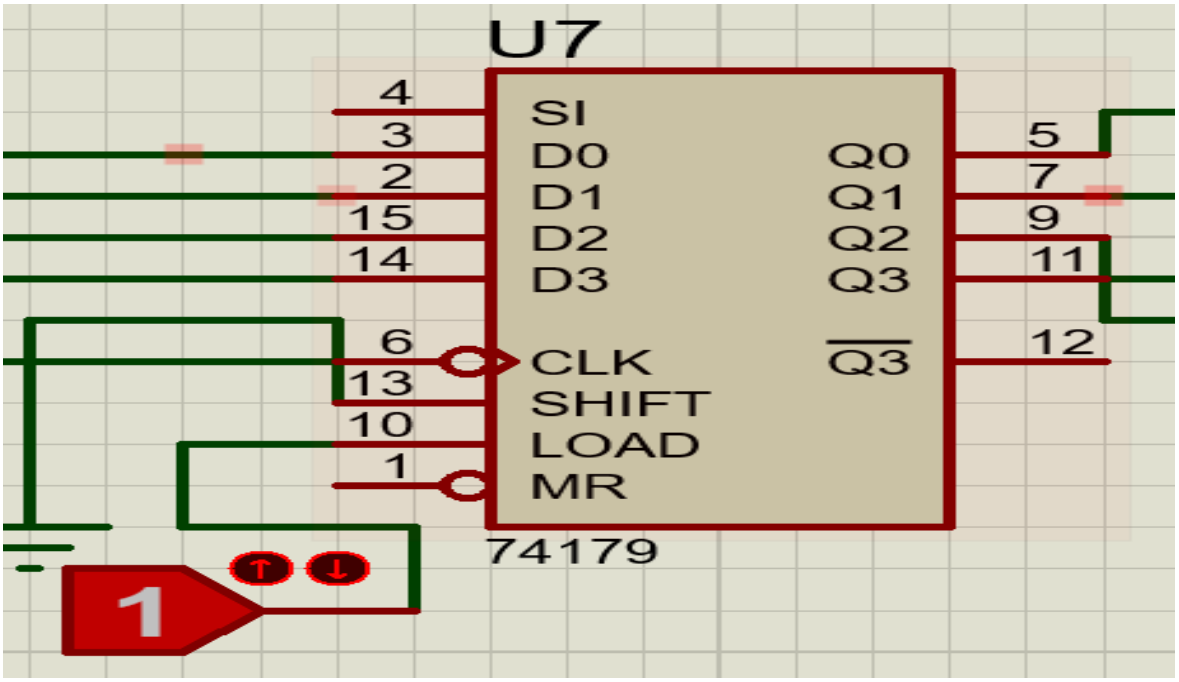
The adder operates Asynchronously.

3. Register R1

Register R1 takes the output of Adder as its input and The User interface Push button as its clock. Based on the Register R1's state, the position of User in LED matrix is determined as R1's state is feed to Decoder of LED Matrix by all Asynchronus Circuits.

So, When Push Button is operated, at its positive edge, Register R2 stores the next position of User which is Addition of last stage and Current dice as produced by Adder. In proteus we use IC 74179 to implement Register R1.

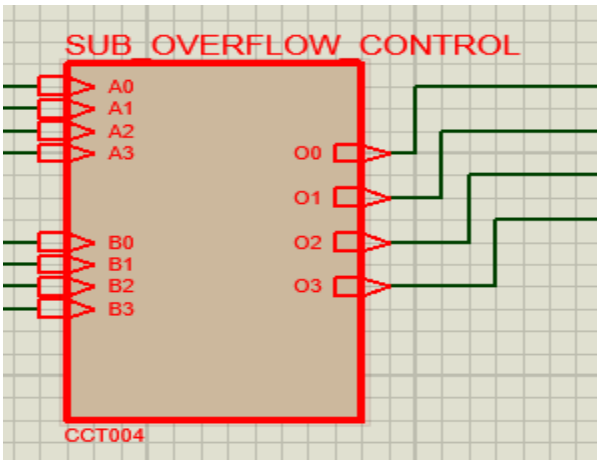
Proteus Implementation –



4. Over Flow Control Subcircuit

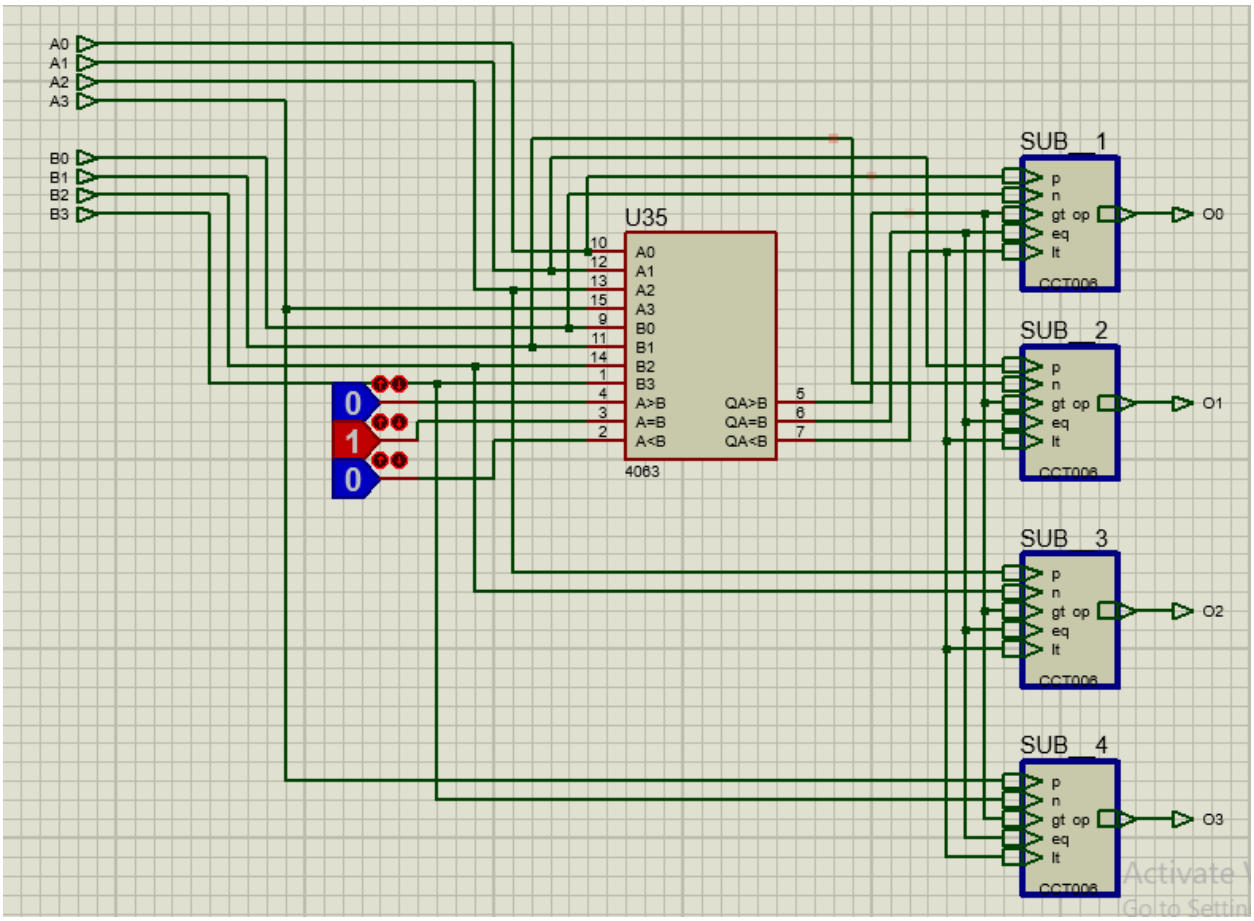
As we have seen so far, The Register R1 produces candidate next state for user in LED matrix. However, Consider the following case –

Currently User is on 11th LED and Current dice is 6. As we are operating on 4x4 Matrix, This move is redundant and can't be made. In such case the adder will go to overflow and produce modulo 16 of (11+6=17) = 1 as its state. We can easily detect this fault by Comparing past state which is 11 and Next candidate state which is 1. So, Overflow Control is basically a comparator which compares Past state and Next candidate state and chooses the bigger one in its output.



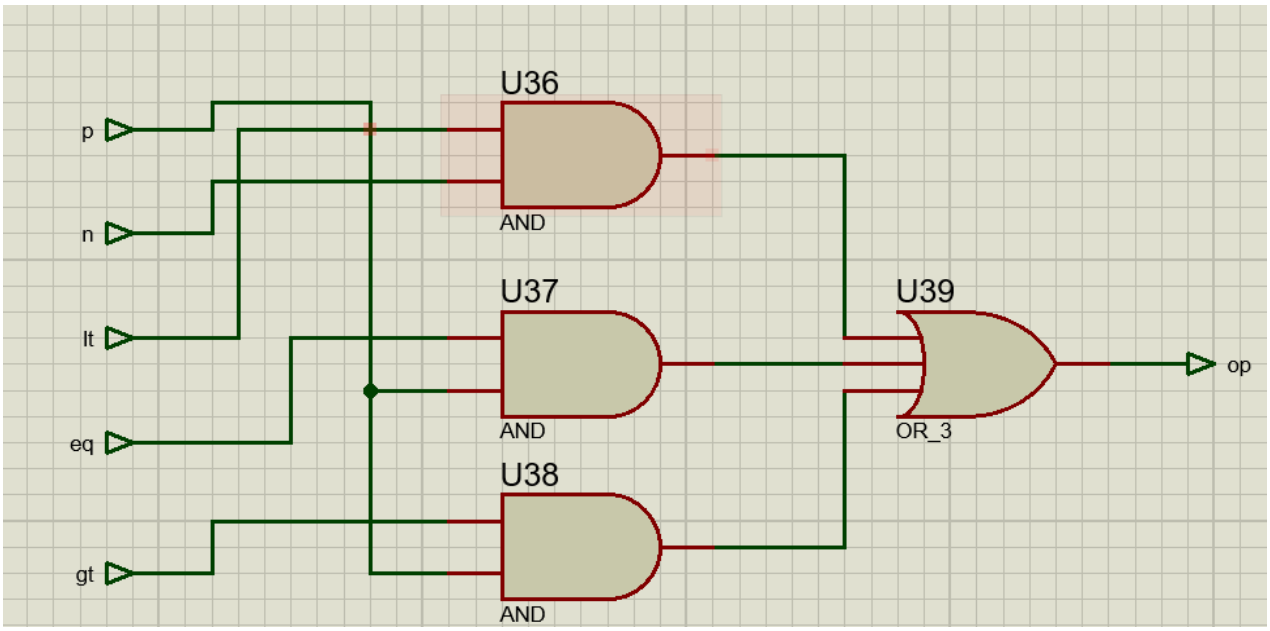
In proteus, Overflow Control is a subcircuit made customed by us.

Inside Subcircuit:



We use 4083 comparator IC to first compare between two past state and next candidate state. Based on result, we choose 4 output bits using a Multiplexer.

Inside the multiplexer subcircuit –

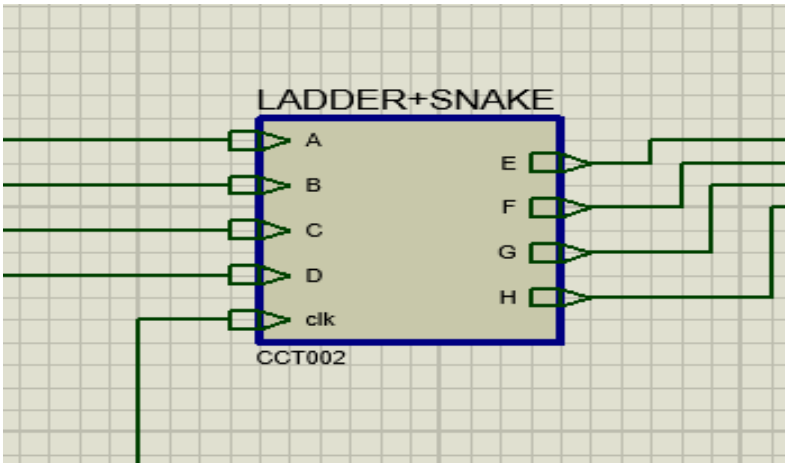


So based on output of Comparator (AeqB, AgtB or AltB) , One bit among past state's bit and candidate next state's bit is chosen .

5. Snake + Ladder Logic

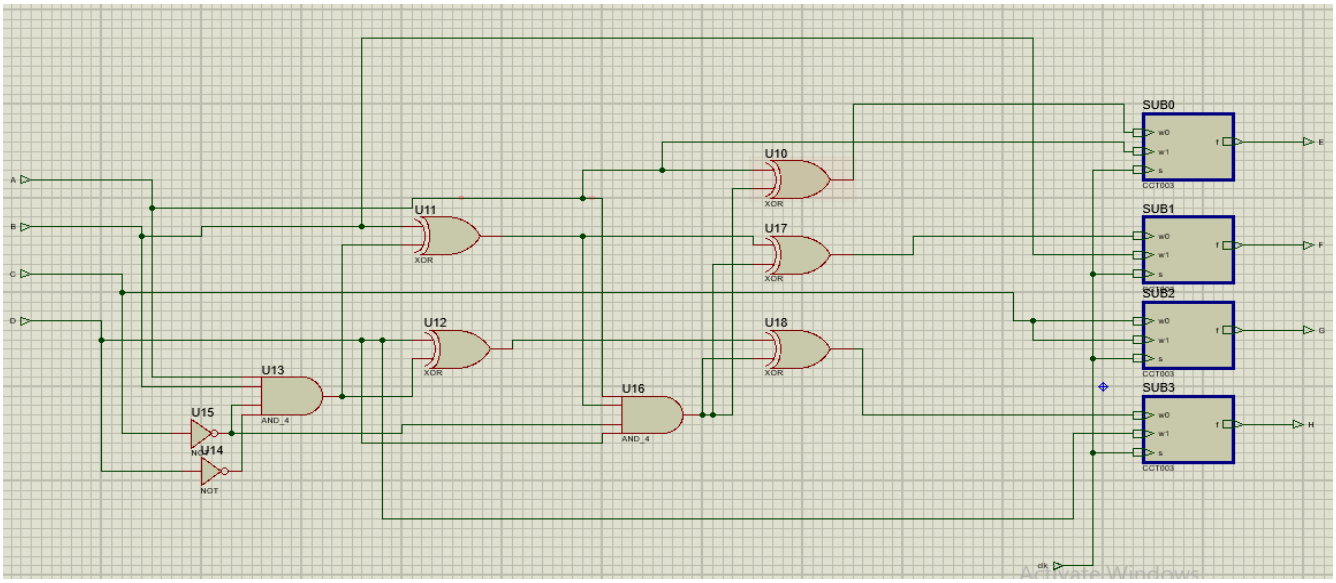
Next is Snake + Ladder Logic. We implement 1 snake and 1 Ladder. This subcircuit is manually created by us.

The Ladder is implemented at LED no 3 with a jump of 6. So , When user comes to LED 3 by state and dice addition , As long as Push button is operating the 3rd LED will be high but as Push button action is completed and move is complete the user will move to 3+6=9th LED .



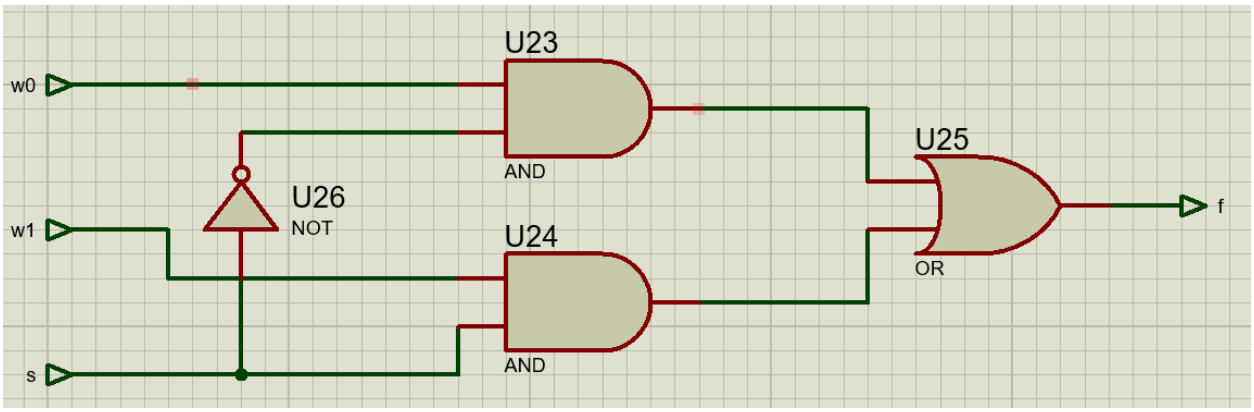
The Snake is implemented at LED no 11 and with a jump of 11. So , When someone user to LED 11 by state and dice addition , , As long as Push button is operating the 11th LED will be high but as Push button action is completed user will move to 11-11 = 0th LED .

Inside the Subcircuit:



So if 0011 is detected (By U13) or 1001 is detected (By U16), Corresponding bit changes are done. Next the bits are passed to a synchronous subcircuit.

Inside the synchronous subcircuit:

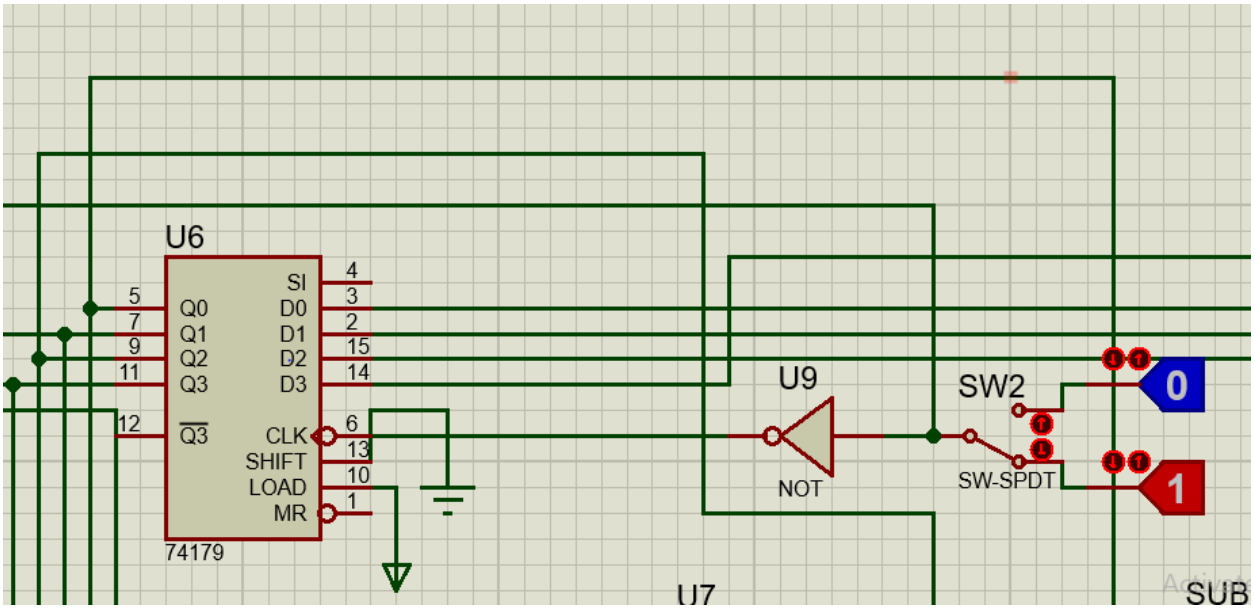


So, The Ladder/Snake modified result is shown only after the Push button action is over i.e. one complete move is done.

6. Register R2

Register R2 Holds last state of User in LED matrix as its state. It takes output from Snake+Ladder subcircuit as its input and its clock is the compliment of Switch. So, When we Make switch from High to Low to make dice rolling i.e. When we move to next step , The register is activated and Stored the last state in LED Matrix as its state . This state is later fed to adder to produce next state with addition of Dice.

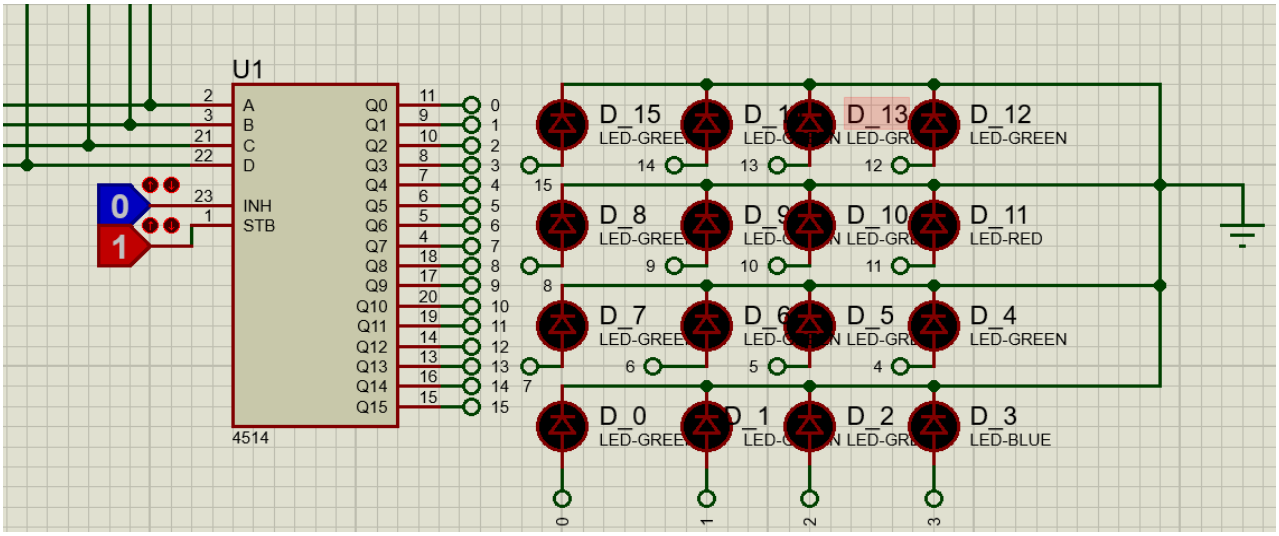
R2 In proteus is implemented as shown –



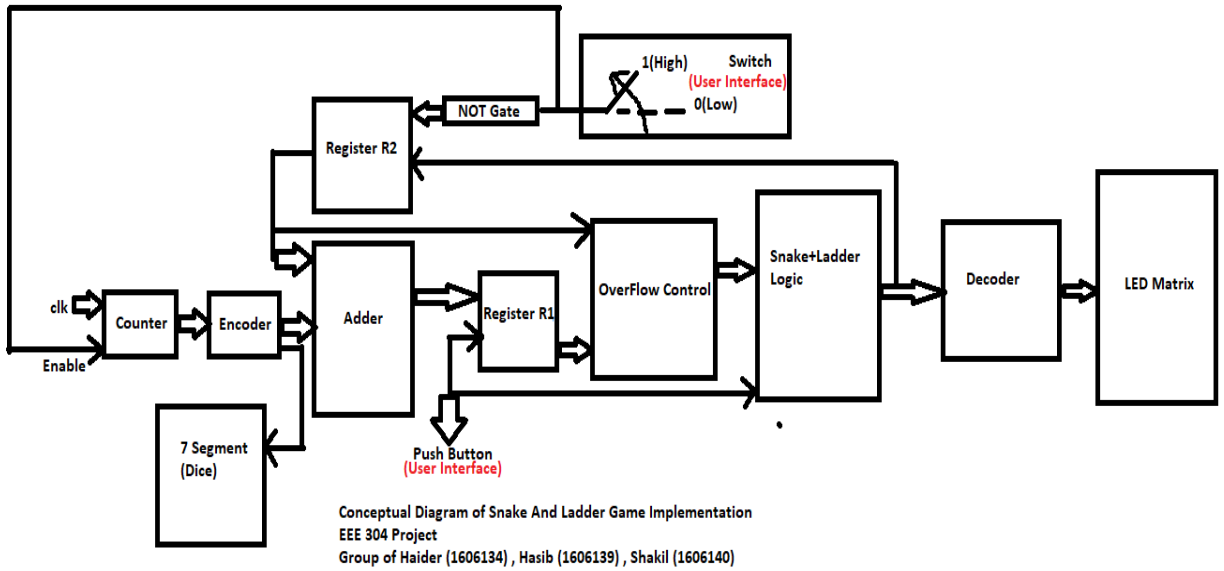
7. Decoder+LED Matrix

Snake+Ladder logic creates a 4 bit binary number as next state. Corresponding to the number, we have to make one LED on i.e. we have to One Hot Encode the output of Snake+Ladder logic . For this purpose a 4-to-16 Decoder is used which takes 4-bit binary state as its input and produces 16 signals which feeds 16 LED's and only one of them are high at any moment .

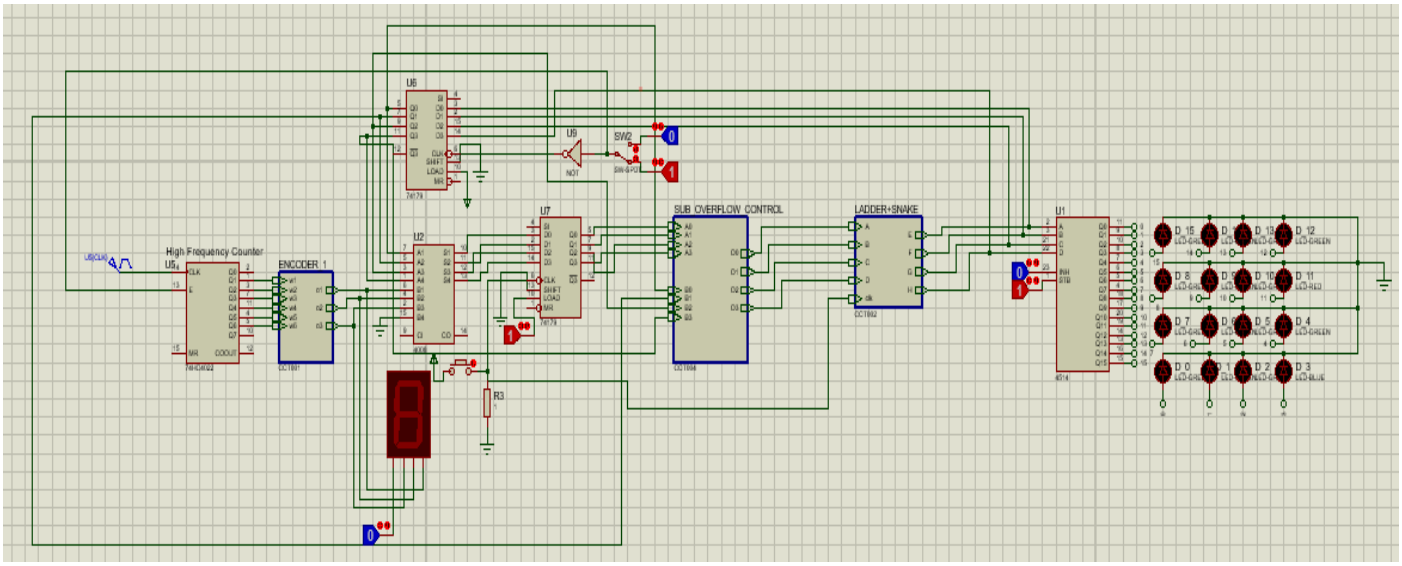
Decoder + LED Matrix in proteus is implemented as below –



The Full Diagram in a simplified way is shown below –



Actual Proteus Implemented Full circuit is shown below –



The Basic 4 × 4, 1 player Game Needs –

- 2 Registers
- 1 Encoder
- 1 Decoder
- 1 Counter
- 1 Adder
- 1 OverFlow Control SubCircuit
- 1 Snake+Ladder Logic SubCircuit
- 1 7 Segment Display .

Verilog Implementation

In this Section, Verilog code for two players in Ladder-Snake game implement is discussed. Here we use a high frequency clock pulse as input to run the game smoothly and generate random numbers. We have also two inputs: a switch to roll the dice (Random numbers) and a push button to pass the dice value to Ladder-Snake module to advance/ place the tokens. And we get six outputs: current state of dice, current state and LED state of two players & winner.

Main Code

```
module trial_2(input clk, push, switch, output [2:0] dice_live, output [3:0] state_1_live, state_2_live ,
output [15:0] led_16_1,led_16_2 , output reg [1:0] winner);

    wire [2:0] dice;
    wire [3:0] added_1, added_2, state_1, state_2, o_11, o_12, o_21, o_22, o_31, o_32;
    reg T;
    always@(negedge switch) T<=~T;

    counter  produce_dice(clk, switch, dice);
    adder  do_addition(state_1, state_2, dice,T, added_1, added_2);
    register  R_11(push, added_1, o_11);
    register  R_12(push, added_2, o_12);

    overflow_control  OF_1(state_1, o_11, o_21);
    overflow_control  OF_2(state_2, o_12, o_22);

    ladder_snake  LS_1(o_21, push, o_31);
    ladder_snake  LS_2(o_22, push, o_32);

    decoder  DEC_1(o_31, led_16_1);
    decoder  DEC_2(o_32, led_16_2);

    register  R_21(~switch, o_31, state_1);
    register  R_22(~switch, o_32, state_2);


    assign dice_live = dice;
    assign state_1_live = state_1;
    assign state_2_live = state_2;
```

```
        always@(negedge push) begin
                if(!(o_31 == 4'b1111) || !(o_32 == 4'b1111)) begin
                        if( o_31 == 4'b 1111 ) winner = 2'b01;
                        else if(o_32 == 4'b 1111) winner= 2'b10;
                        else winner = 2'b00;
                end
        end
endmodule



module counter(input clk, E, output reg [2:0] count);
        always @(posedge clk) begin
                if(!E) begin
                count = count+1;
                if (count==7)
                count = 0;
                end
        end
endmodule



module adder(input [3:0] state0, state1, input [2:0] dice, input T, output reg [3:0] result0,
result1);
        always@(*) begin
                if(T) begin
                result1 = state1 + dice ;
                result0 = state0;
                end
                else if(T == 0) begin
                result0 = state0 + dice;
                result1 = state1;
                end
        end
endmodule
```

```
module register(input clk, input [3:0] D, output reg [3:0] Q);
        always@(posedge clk)
                Q<=D;
endmodule
```

```
module overflow_control(input [3:0] state, added, output reg [3:0] result);
        always @(*) begin
            if (state>added) result = state;
            else if(state<added) result=added;
            else result = added;
        end
endmodule
```

```
module ladder_snake(input [3:0] inn, input push, output reg [3:0] outt);
        always @(*) begin
            if(inn==11) outt=0;
            else if(inn==3) outt=9;
            else outt=inn;
        end
endmodule
```

```
module decoder(input [3:0] decoded, output reg [15:0] led);
        always@(*)begin
                led=0;
                   led[decoded] = 1;
           end
endmodule
```

*We will break down the **working principle** now along with the **Verilog Implementation.***

1. Function of 'T'

```
reg T;
always@(negedge switch) T<=~T;
```

T is a variable with changes with each negative edge of the switch and keeps the value till the next negative edge. This helps to roll the dice for each player by rotation.

2. Counter

```
counter  produce_dice(clk,switch,dice);

module counter(input clk, E, output reg [2:0] count);
    always @(posedge clk) begin
        if(!E) begin
        count = count+1;
        if (count==7)
        count = 0;
        end
    end
endmodule
```

Here, counter is used to produce dice randomly. It takes a high frequency clock & switch as input and gives a dice value. When switch is off, the counter counts 0 to 6 periodically at the clock frequency and when the switch is pressed, the dice takes the value of counter at that moment. As the frequency is very high, the dice is almost random.

3. Adder

```
adder  do_addition(state_1, state_2, dice, T, added_1, added_2);

module adder(input [3:0] state0,state1, input [2:0] dice, input T, output reg [3:0] result0,
result1);
    always@(*) begin
        if(T) begin
        result1 = state1 + dice ;
        result0 = state0;
        end
        else if(T == 0) begin
        result0 = state0 + dice;
        result1 = state1;
        end
```

```
    end
endmodule
```

The adder module takes current states of two players, dice value & T. It adds the dice with the current state depending on the T value (0 for first player, 1 for second player) and keeps the other player's state unchanged.

4. Register R1

```
    register  R_11(push, added_1, o_11);
    register  R_12(push, added_2, o_12);

module register(input clk, input [3:0] D, output reg [3:0] Q);
    always@(posedge clk)
        Q<=D;
endmodule
```

When the push button is on, register module takes the adder output of a player and stored it in output variable (o_11/ 0_12) at each positive edge of the clock. Otherwise the output of register remains unchanged.

5. Overflow Control

```
    overflow_control  OF_1(state_1 , o_11 , o_21);
    overflow_control  OF_2(state_2 , o_12 , o_22);

module overflow_control(input [3:0] state, added, output reg [3:0] result);
    always @(*) begin
        if (state>added) result = state;
        else if(state<added) result=added;
        else result = added;
    end
endmodule
```

The overflow_control module takes register R1 output & current state of a player and store the bigger value in the output (o_21/ o_22). If the adder output automatically becomes small, the sum overflows.

6. Snake + Ladder Logic

```
        ladder_snake  LS_1(o_21,push,o_31);
        ladder_snake  LS_2(o_22,push,o_32);

module ladder_snake(input [3:0] inn, input push, output reg [3:0] outt);
        always @(*) begin
            if(inn==11) outt=0;
            else if(inn==3) outt=9;
            else outt=inn;
        end
endmodule
```

The ladder_snake module takes overflow_control output as input. If it is 3, it is a ladder & the token jumps to 9 and it is 11, it is a snake & the token goes to 0. Otherwise the token goes to the place indicated by overflow_control output

7. Decoder

```
        decoder  DEC_1(o_31,led_16_1);
        decoder  DEC_2(o_32,led_16_2);

module decoder(input [3:0] decoded, output reg [15:0] led);
        always@(*)begin
                led=0;
                    led[decoded] = 1;
           end
endmodule
```

It is a 4 to 16 decoder. The decoder module takes the ladder_snake module's output and decode it for 4 × 4 LED array to show the token's current position. It is output user interface or screen for the game.

8. Register R2

```
        register  R_21(~switch,o_31,state_1);
        register  R_22(~switch,o_32,state_2);

module register(input clk, input [3:0] D, output reg [3:0] Q);
        always@(posedge clk)
                Q<=D;
endmodule
```

When the switch is off, register module takes the decoder output of a player, and stored it in output variable (state_1/ state_2) at each positive edge of the clock. Otherwise the output of register remains unchanged.

9. Winner Logic

```
always@(negedge push) begin
        if(!(o_31 == 4'b1111) || !(o_32 == 4'b1111)) begin
                if( o_31 == 4'b 1111 ) winner = 2'b01;
                else if(o_32 == 4'b 1111) winner= 2'b10;
                else winner = 2'b00;
        end
end
```

This part of code shows the winner. If any of the player's token reaches place-15, it shows the player as winner. It shows 01 for first player & 10 for second player. And the game is over. Otherwise it shows 00 & the game is still on.

Simulation Results

After Simulating the verilog file and then creating the vector waveform file , We get the result of the game. This is the full game shown in figure 1.

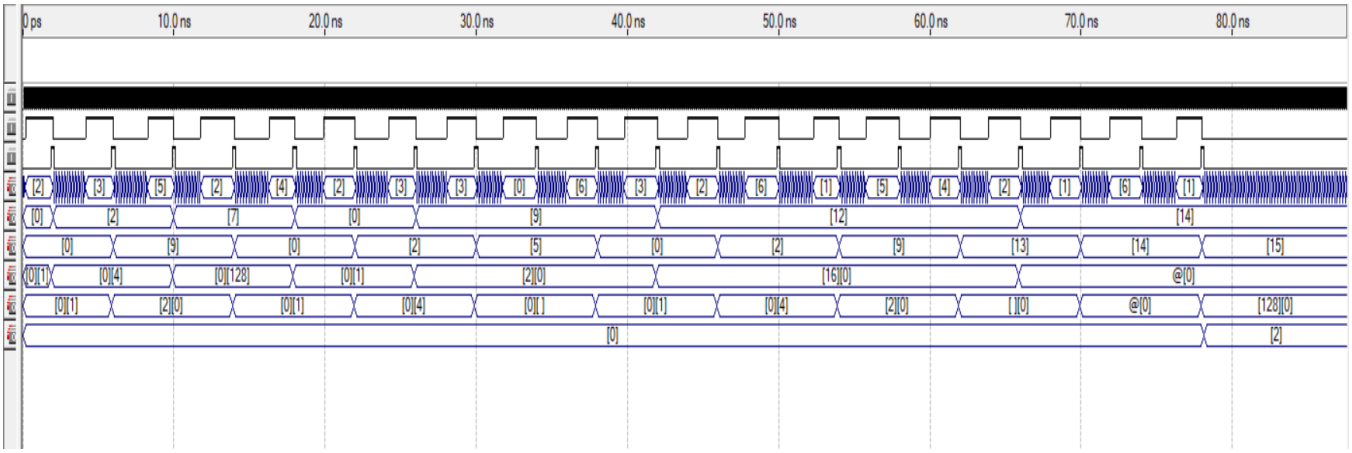


Figure 1

Here, clk variable is a random number generator that gives us a random number for the dice every time the switch button is positive. Then the push button assigns the dice number to the register of the player.

There are two player in this game, player 1 and player 2. Their progress is added to the variable state_1_live and state_2_live. At first they both are clear/zero. After the first dice is rolled, the dice value is (2). So it is added to 1st player. Then the next dice value is (3). It is added to 2nd player while the first player retains it's value. Then the dice goes to 1st player again. Thus this game is played until any player reaches destination, that is 15 in this game.

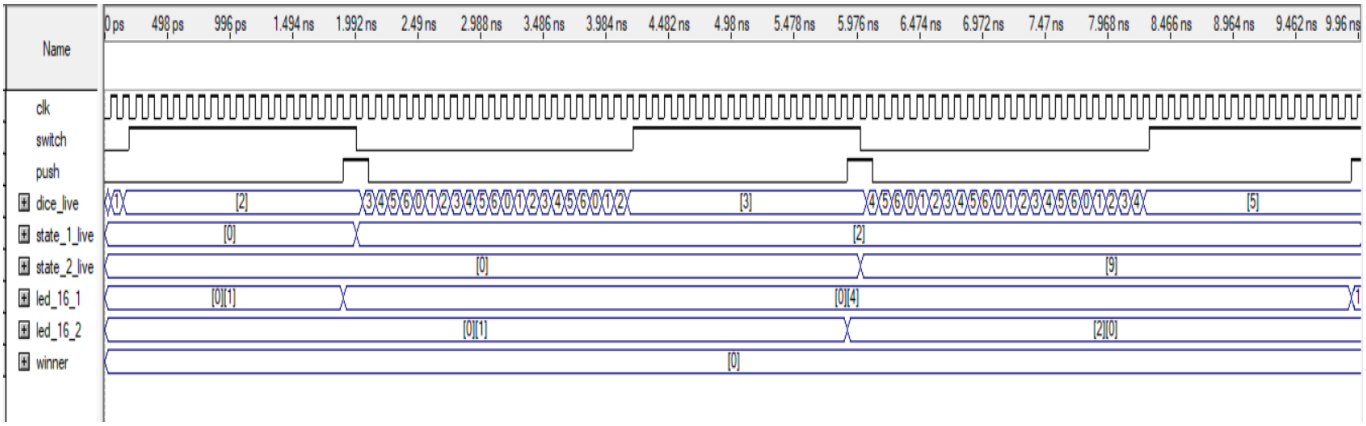


Figure 2

But there is a ladder in this game in (3) position thar takes the player in (9) position. So when 2nd player hits 3 in his 1st dice, It takes him to (9) point as seen in figure 2.
The subsequent moves are illustrated in the next figures.

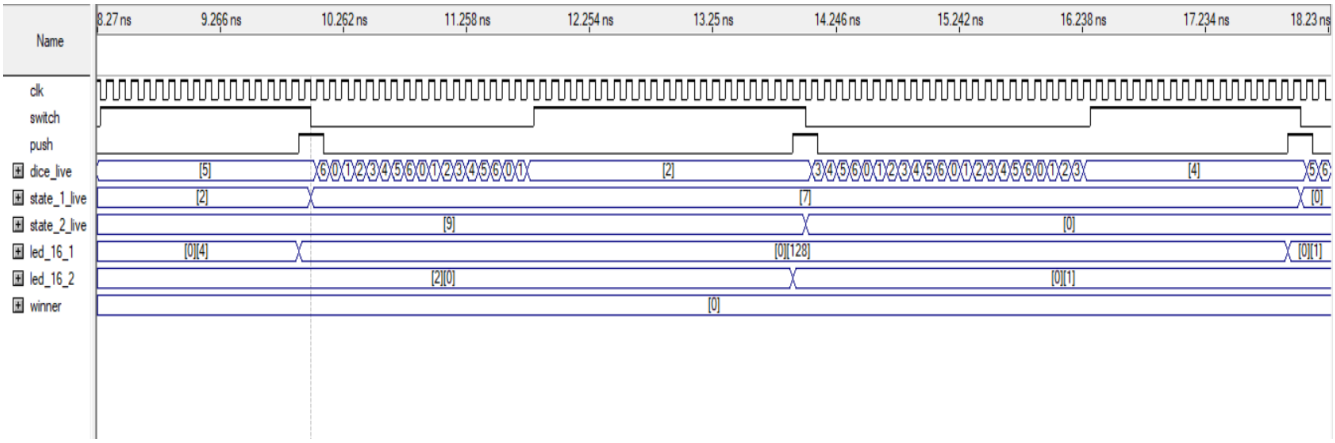


Figure 3

Now There is no snake ladder game without a snake! We have a snake at (11) position. This snake takes player from 11 to 0. From figure 3, we can see that player 2 has fallen to the trap and gone all the way to 0 in 2nd dice roll.

But 1st player steps into the snake and goes back to 0 points in it's 3rd dice roll in figure 4.

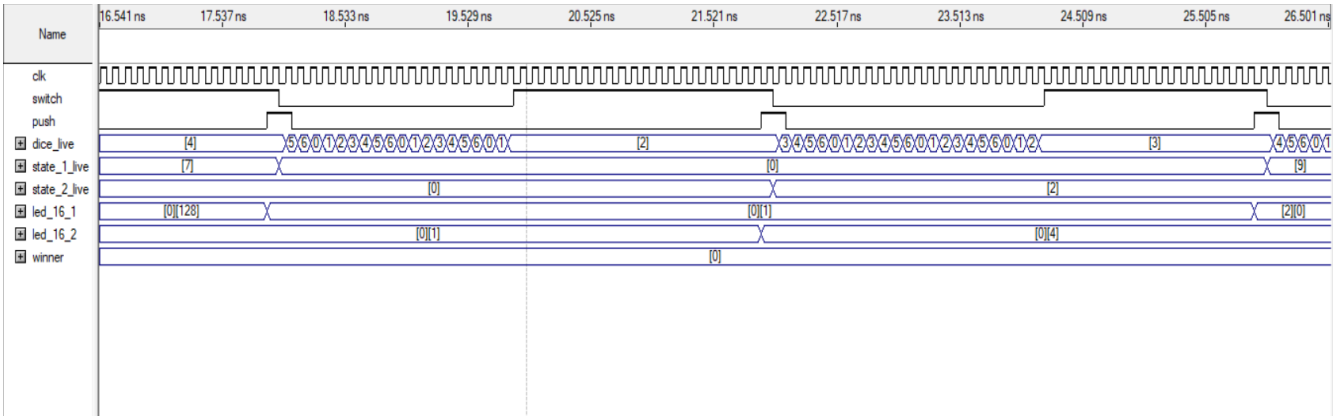


Figure 4

The 2nd player is unfortunate again and falls to the snake again in his 5th dice roll as seen in figure 5.

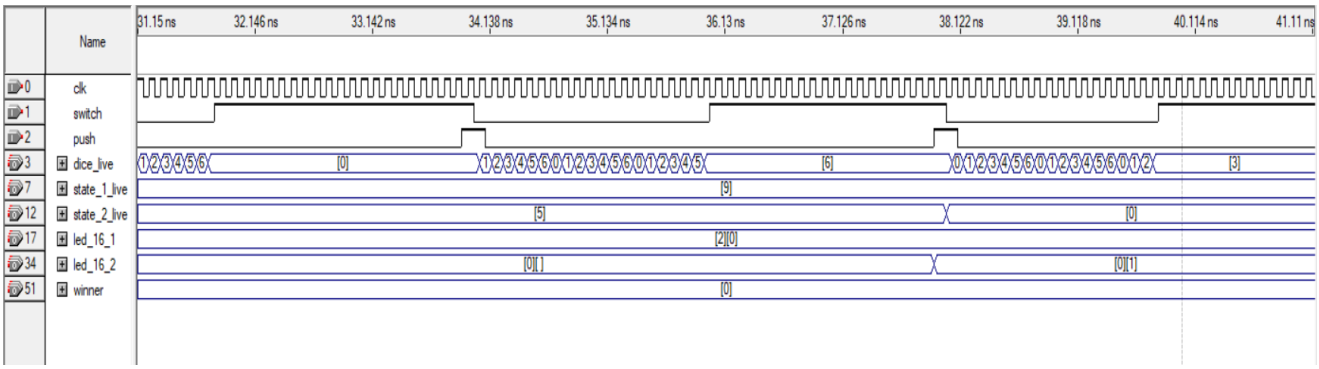


Figure 5

The 1st player steadily gains the ground. But 2nd player gets lucky and gets the ladder in his 7th dice and climbs all the way up to 9 in figure 6. Thus the game goes on.

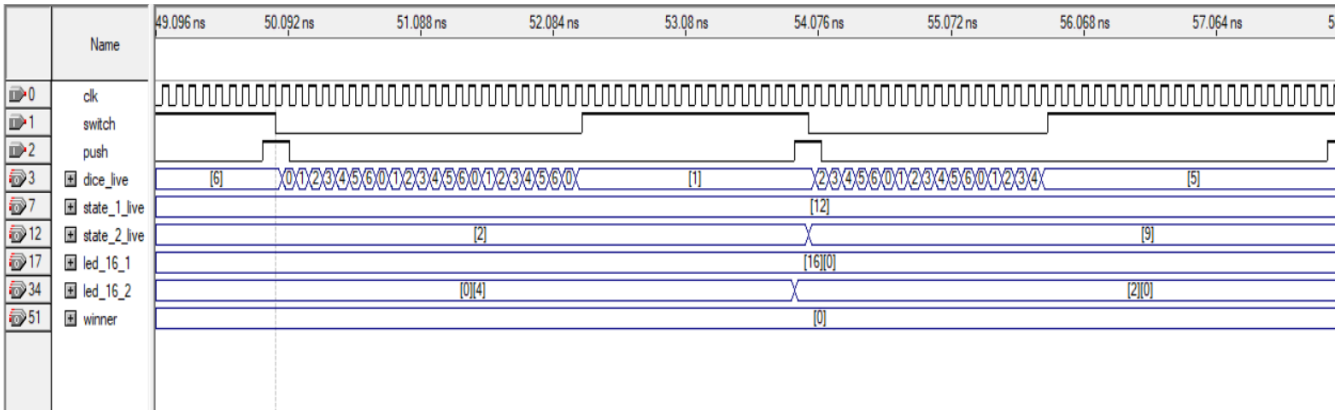


Figure 6

The difference between 2 player becomes minimal as the game goes on. But 2nd player gets the win in 11th dice roll and reaches 15 points as seen in figure 7.

Thus we have declared player 2 as winner of this round.

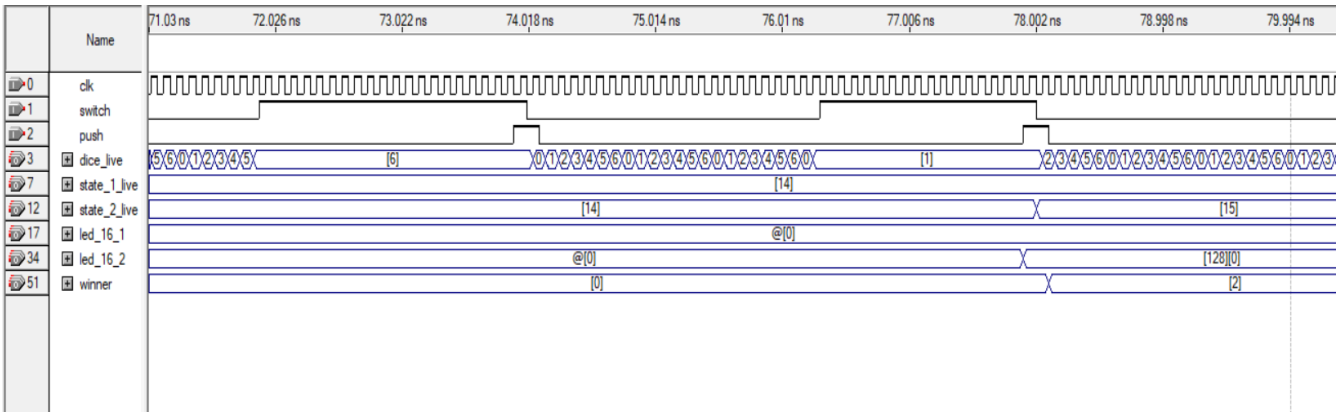


Figure 7

Thus we ended a delightful game of snake and ladder between two player, and the final winner is player 2.

We can play this game again by changing the frequency of the high frequency clock, producing a new set of random numbers and creating new scenerios in this game.

Conclusion

The snake and ladder game is certainly a very entertaining game. We have created the verilog implementation of the game. Here in verilog, we have implemented a two player system playing in virtual world and competing with each other. Here we are using random number generator for a fair game. Thus we made the game in verilog and tested the game through simulation. Thus we have found the winner from simulated clock diagram.

We have also implemented the game in circuits in proteus. Where we have implemented a 4*4 grid of snake ladder game just like the verilog. The input is represented by a switch for rolling the dice and two Push button for move by player 1 & player 2. The outut is represented by a 4*4 LED Board. So it is a 16 point snake and ladder game with a snake in 11 and a ladder in 3. Thus we have circuit implemented the game in proteus.

This is a very fun game and we hope everyone will enjoy it too.

My Contribution to the Project

Name: SM Haider Ali Shuvo
Std ID: 1606134
Contribution: **Proteus Implementation**

I have contributed in proteus implementation mainly where I have built several subcircuits using knowledge of EEE 304 for correct synthesis , chose proper IC's through trial and error , clocked the registers accordingly using knowledge of synchronous circuits and interconnected different parts properly through wires to make the game working live .