Snake Ladder Game on 8*8 Dot Matrix LED

FPGA Project Report

Course: 2EC202CC23 : FPGA-based System Design

B. Tech. Semester IV

Submitted by:

Vansh Champaneri (23BEC027)

Department of Electronics and Communication Engineering,
Institute of Technology
Nirma University
Ahmedabad 382481

March 2025

Abstract

This report presents the design and implementation of a simplified Snake and Ladder game on an FPGA using an 8x8 LED matrix. The game logic is implemented in Verilog, incorporating a dice rolling mechanism, player movement, and predefined snake and ladder positions. The 8x8 LED matrix is directly controlled using Verilog, with active-low rows and active-high columns to display the game board and player position. This implementation demonstrates real-time control of an LED matrix for interactive gaming applications.

The primary focus of the design is to provide a clear and responsive game experience without incorporating a winner screen. Instead, the module emphasizes real-time gameplay, including accurate dice simulation, button debouncing, LED matrix scanning, and a blinking effect to highlight the player's current position. Operating on a 50 MHz clock, the design demonstrates the integration of state machines and timing counters to ensure smooth board refresh and precise control over game events.

Peripheral Used :

- ✓ 8*8 LED Matrix – For displaying Snake Ladder Board
- ✓ 7 – Seg LED – For displaying Dice Value
- ✓ 3 Swithces:
    - o 1 - Reset Button
    - o 2 - Dice Button
    - o 3 – Button Roll

Literature Survey:

Modern gaming applications often rely on high-resolution graphical displays, microcontrollers, or dedicated gaming consoles. However, implementing a game on an FPGA with an LED matrix presents a unique hardware-based approach. Existing implementations utilize MAX7219 drivers for simplified LED matrix control; however, this project directly manipulates the LED matrix using FPGA logic for greater precision and flexibility, as the SPI protocol does not function effectively in FPGA.

Limitations of Currently Available Technology :

1. Microcontroller-Based Solutions:
   Traditional implementations use microcontrollers (e.g., Arduino) to control LED matrices, which introduces software latency.
2. High Resource Consumption:
   Some FPGA-based designs use excessive resources due to inefficient matrix control.

3. Limited Interactivity:
   Many FPGA games focus on 5static displays rather than interactive movement and dynamic updates.

Proposed Solution :

- **Direct LED Matrix Control:**
  The 8x8 LED matrix is driven directly by the FPGA without additional ICs like MAX7219.

- **Snake and Ladder Logic:**
  The game logic incorporates a predefined lookup table for snake and ladder positions.

- **Button-Based Dice Mechanism:**
  Two push buttons control dice rolling and movement.

- **Player Indication:**
  The player's position blinks at a fixed rate for clear visibility.

- **Clock Handling:**
  The system operates using an FPGA clock with frequency division for display refresh and blink effects.

Design Overview:

Game Board Representation:

The board is represented using an 8×8 LED matrix. A ROM holds the static board pattern, while dynamic overlays indicate the player's current position. A blinking effect is applied to the player's LED for visibility.

Dice and Movement:

The dice value (1-6) is cycled using a dedicated button. The player's movement is updated only upon pressing the confirmation button. The new position is computed as the sum of the current position and dice value. Special squares trigger snake or ladder logic only if landed on exactly.

Special Square Logic (Snake & Ladder):

Specific squares are designated for snakes or ladders. There are 2 Snakes and 2 Ladders,

Snake 1: Square 60 moves the player to square 5
Snake 2: Square 25 moves the player to square 7
Ladder 1: Square 19 moves the player to square 50
Ladder 2: Square 10 mov\es the player to square 32

8x8 LED Matrix Working Principle:

The 8x8 LED matrix consists of 8 rows and 8 columns, forming a grid of LEDs. To control an individual LED:

Rows are Active-Low: A row is activated by setting it to logic low.

Columns are Active-High: A column is activated by setting it to logic high.

To turn on an LED at a specific (row, column) position, the corresponding row is pulled low, and the column is pulled high.

To display dynamic content, row scanning is performed rapidly while updating column data accordingly.

Implementation Details :

Clock and Timing:

A 50 MHz clock drives the design. This high-frequency clock is used for LED matrix refreshing (~1ms per row), blink signal generation (approximately 1 Hz), and game state updates.

Button Debouncing and Edge Detection:

Two buttons (one for cycling the dice and one for confirming a move) are used. Debouncing and rising edge detection are implemented to avoid multiple triggers from a single press.

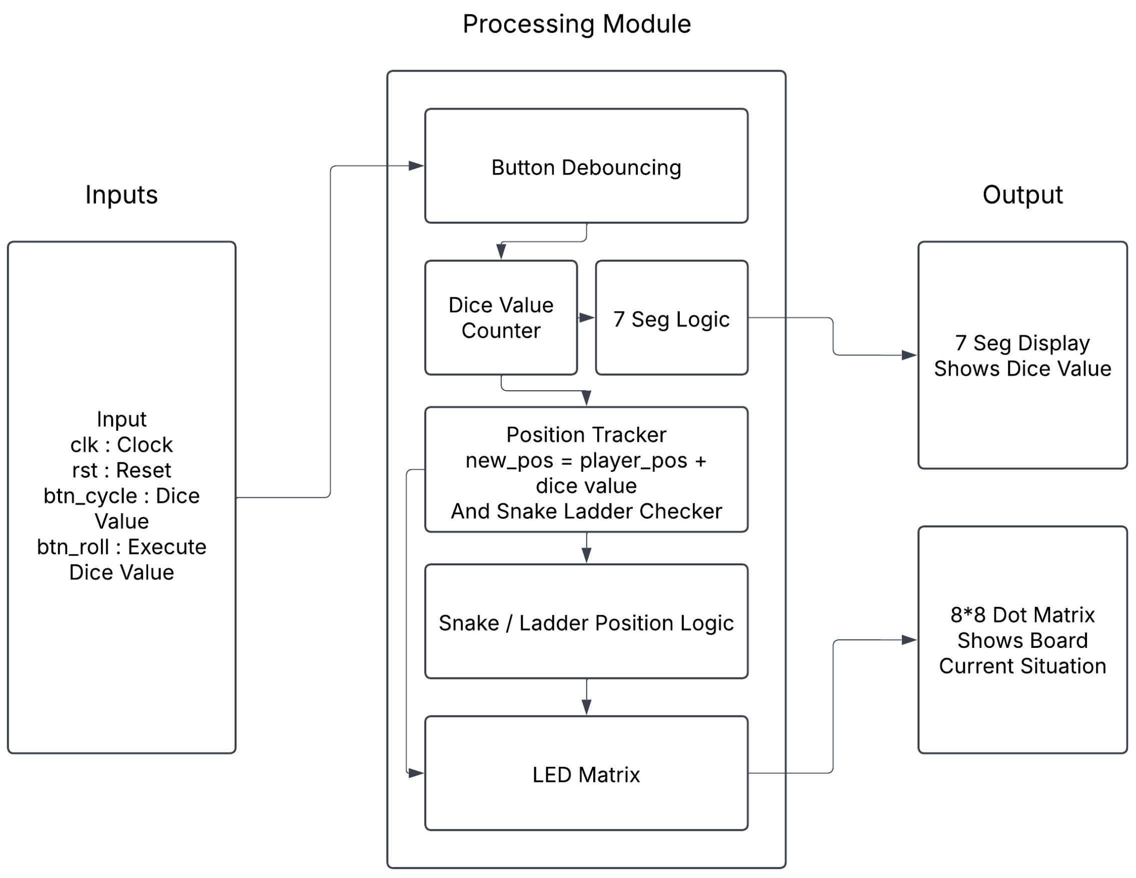
Game State Management:

The dice value, player position, and game over flag are updated in a single always block. This avoids multiple drivers for a signal and ensures correct state transitions. A temporary variable is used to compute the intended new position to correctly apply snake/ladder logic only on exact landing.

LED Matrix and 7-Segment Display:

The LED matrix scanning logic uses a refresh counter to cycle through rows. The board pattern is stored in a ROM, and a blinking overlay is applied for the player's position. The dice value is mapped to a 7-segment display using combinational logic.

Block Diagram:



The block diagram represents the FPGA-based Snake and Ladder game using an 8x8 LED matrix. It highlights the core processing modules, input controls, and output displays.

1. Inputs Section:

- clk (Clock): Provides the system clock for synchronizing all operations.
- rst (Reset): Resets the game state, including player position and dice value.
- btn_cycle (Dice Value Button): Cycles through dice values (1→2→3→4→5→6→1…).
- btn_roll (Execute Dice Value Button): Confirms the dice roll and moves the player.

2. **Processing Module:**

This section contains multiple functional blocks that execute the game's logic.

- Button Debouncing: Ensures stable button inputs, removing unwanted glitches due to mechanical button presses.
- Dice Value Counter: Cycles through the dice values when btn_cycle is pressed.
- 7-Segment Logic: Displays the current dice value on a 7-segment display.
- Position Tracker: Updates the player's position based on the dice roll. Checks for snakes and ladders, modifying the position accordingly.
- Snake / Ladder Position Logic: Stores predefined locations of snakes and ladders and adjusts player movement if needed.
- LED Matrix Driver: Controls the 8x8 LED matrix, ensuring proper row-column activation.

3. **Output Section:**

- 7-Segment Display: Shows the currently selected dice value.
- 8x8 LED Matrix:

Displays the board layout with predefined snake and ladder positions (static display).

The player's current position blinks for visibility.

This architecture ensures efficient real-time game execution while maintaining accurate display control.

Blink Logic:

FPGA clock is 50 MHz, each clock cycle is 1 / 50M = 20 ns.

BLINK_THRESHOLD = 25,000,000 means:

Blinking period = 25,000,000 × 20 ns = 0.5 seconds per ON/OFF.

Total blink cycle (ON + OFF) = 1 second.

LED Matrix :

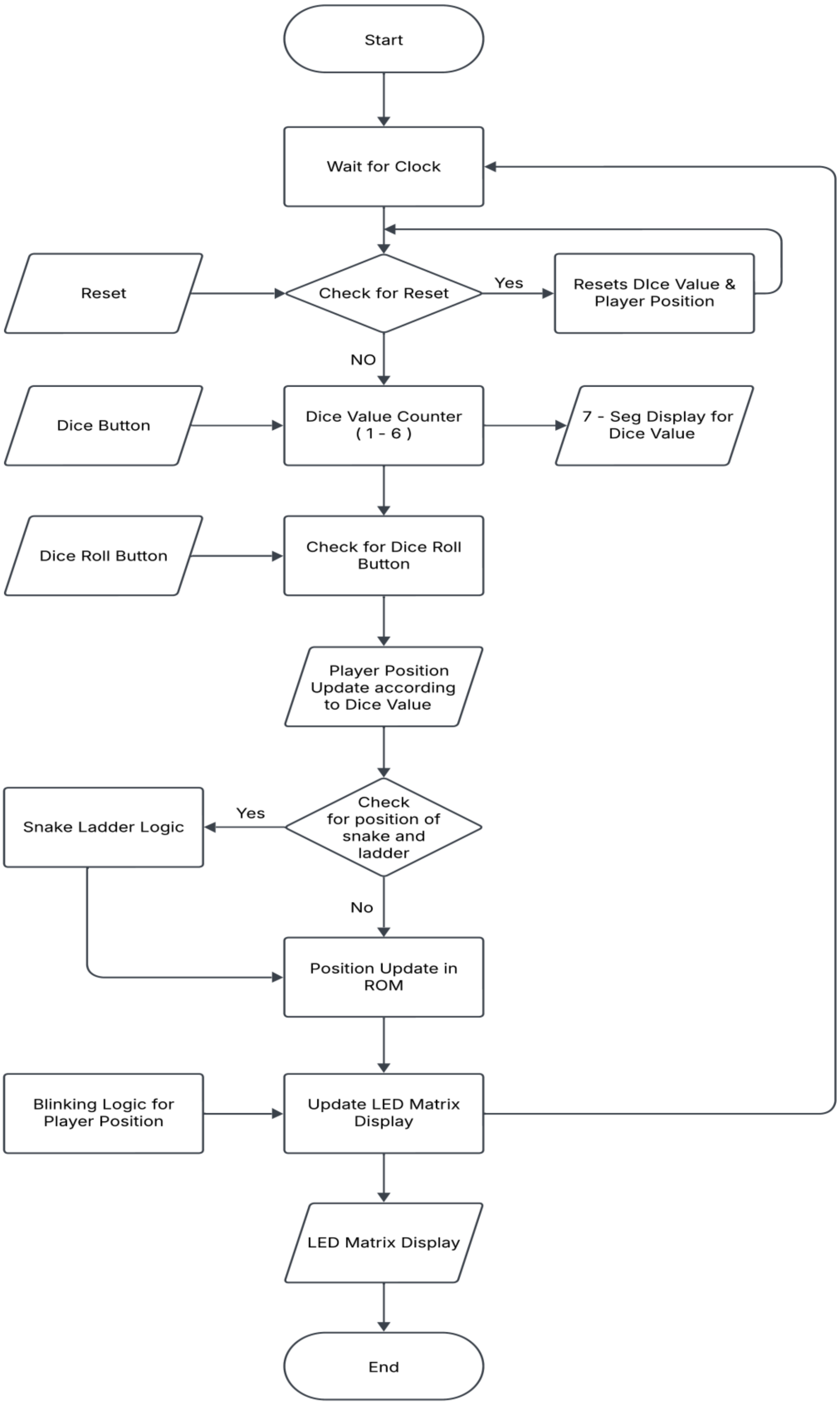
As 8*8 LED can only show one row at a time so we will scroll through all row at 120 Hz so screen doesn’t seem to be flicker.

```
if (refresh_counter >= 16'd50000)
```

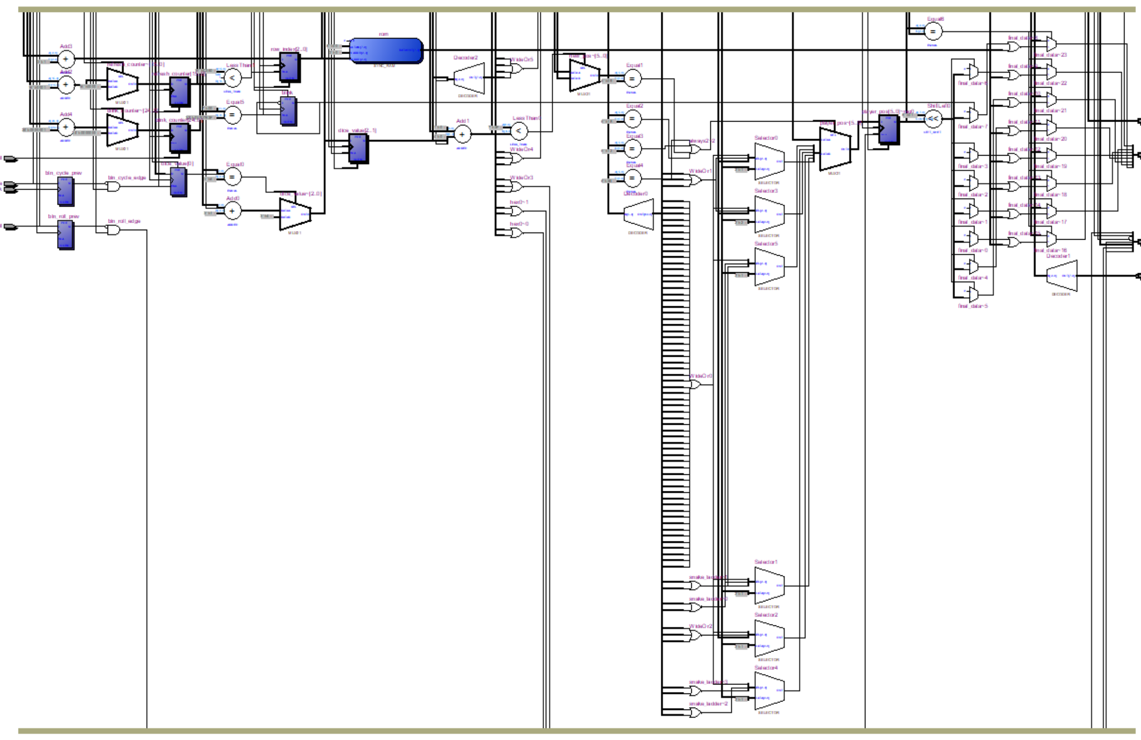
Here With a 50 MHz clock, this gives a refresh rate of ~125 Hz per row (50M / (50k × 8) ≈ 125 Hz).

The entire matrix refreshes ~125 times per second, ensuring smooth visibility.

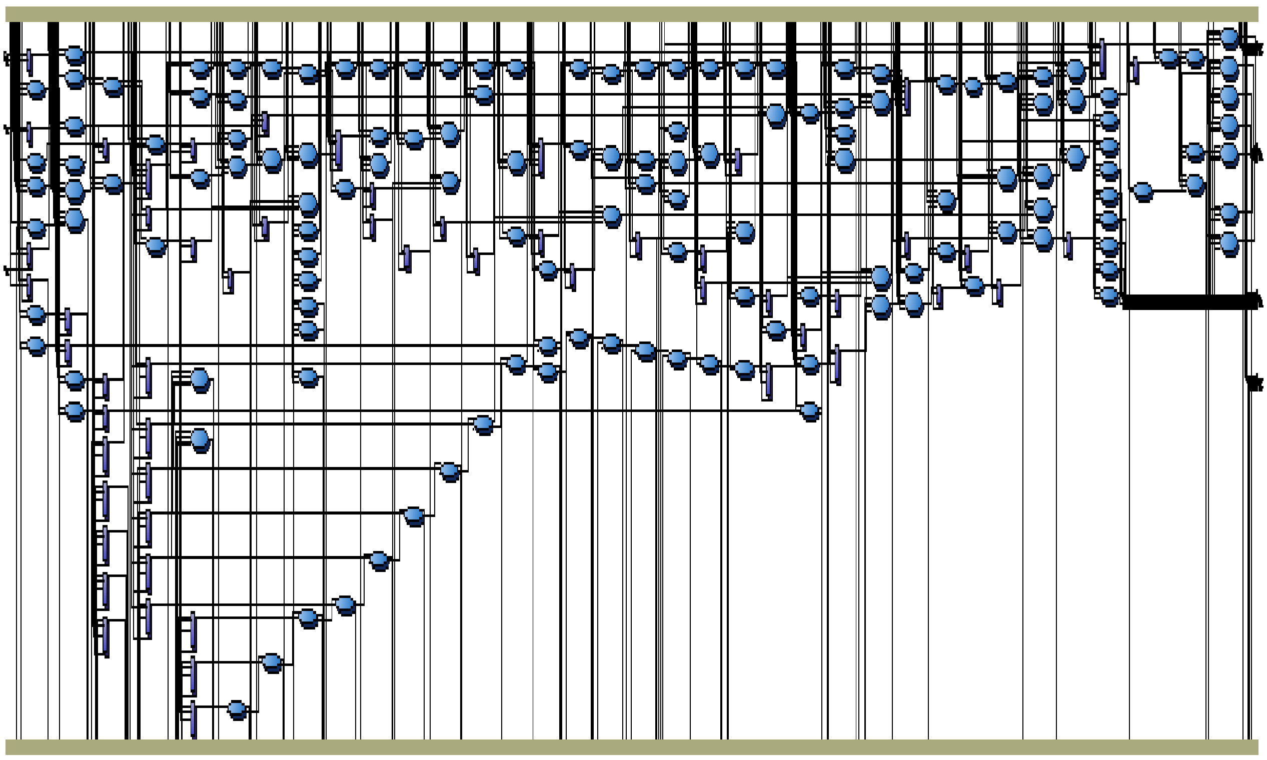
Flowchart:


Start
Wait for Clock
Reset
Check for Reset
Yes
Resets DIce Value & Player Position
NO
Dice Button
Dice Value Counter (1 - 6)
7 - Seg Display for Dice Value
Dice Roll Button
Check for Dice Roll Button
Player Position Update according to Dice Value
Snake Ladder Logic
Yes
Check for position of snake and ladder
No
Position Update in ROM
Blinking Logic for Player Position
Update LED Matrix Display
LED Matrix Display
End

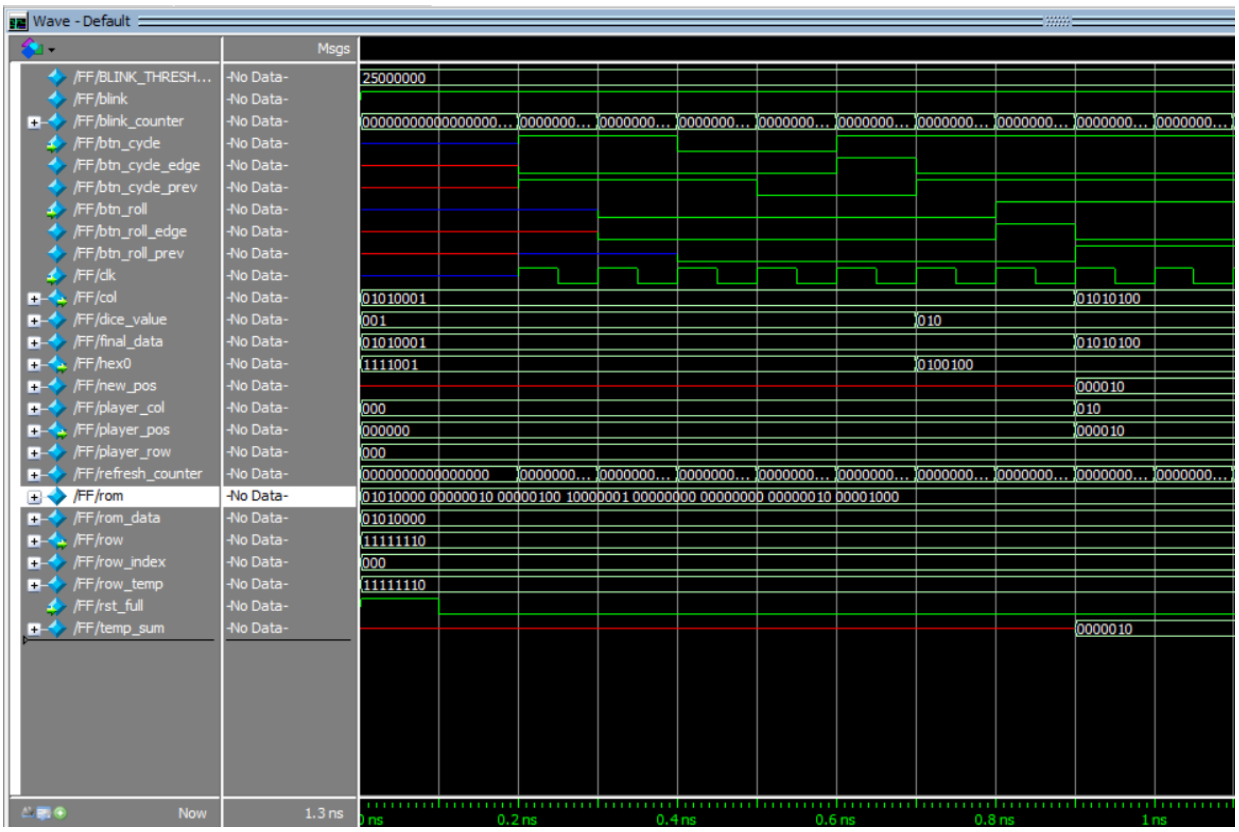
RTL :



TTL :



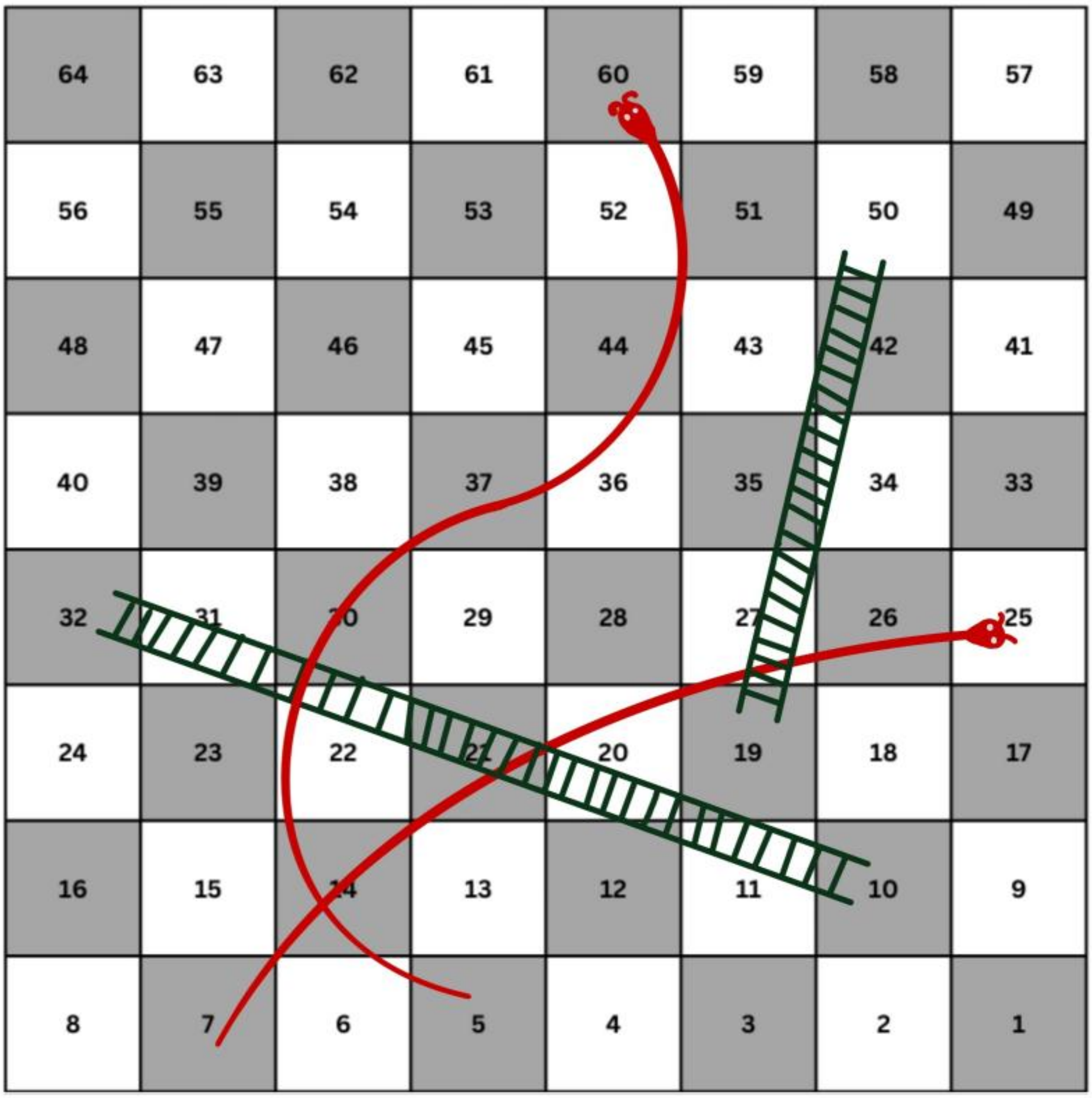
Modelsim Simulation:



Debouncing Observation from Above simulation:

Clock Cycle	btn_cycle (Raw Button Input)	btn_cycle_prev (Previous State)	btn_cycle_edge (Edge Detection Output)	Comment
t0	0	0	0	Button not pressed
t1	1 (Bounce)	0	1	Detected as a **new press**
t2	0 (Bounce)	1	0	Glitch (bounce) ignored
t3	1 (Bounce)	0	1	Another **new press** detected (wrong!)
t4	1 (Stable)	1	0	Button held down (no false triggers)

Snake and Ladder Board:



Conclusion:

This project successfully demonstrates the implementation of a Snake and Ladder game using FPGA-based control of an 8x8 LED matrix. By eliminating external driver ICs and directly manipulating the LED matrix through Verilog logic, the design achieves precise control, efficient resource utilization, and real-time responsiveness. The use of a button-based dice mechanism and predefined game rules ensures an interactive and engaging gameplay experience.

Additionally, the integration of an optimized lookup table for snake and ladder positions streamlines the movement logic. Future enhancements could include expanding the game board size, implementing multi-player functionality, and incorporating additional game mechanics like power-ups or animations. This project highlights the potential of FPGA-based game development and the versatility of hardware-implemented logic in real-time applications.

Reference:

Verilog Book :

Palnitkar, S. (2003). Verilog HDL: A Guide to Digital Design and Synthesis. Pearson Education.

8*8 Dot Matrix LED Datasheet:

https://www.futurlec.com/LED/LEDM88R.shtml

Reference Video:

https://youtu.be/OQMlbOM7Too?si=T1yMDiN73P1-clVP

- **Appendix (Verilog Code):**

```
module PLS(
    input clk,
    input rst_full,
    input btn_cycle,
    input btn_roll,
    output reg [7:0] row,
    output reg [7:0] col,
    output reg [6:0] hex0,
    output reg [5:0] player_pos
);

    // Button Debouncing
    reg btn_cycle_prev, btn_roll_prev;
    wire btn_cycle_edge, btn_roll_edge;

    assign btn_cycle_edge = btn_cycle & ~btn_cycle_prev;
    assign btn_roll_edge  = btn_roll  & ~btn_roll_prev;

    always @(posedge clk) begin
        btn_cycle_prev <= btn_cycle;
        btn_roll_prev  <= btn_roll;
    end

    // Dice Value Counter
    reg [2:0] dice_value = 3'b001;

    always @(posedge clk or posedge rst_full) begin
        if (rst_full)
            dice_value <= 3'b001;
        else if (btn_cycle_edge) begin
            if (dice_value == 3'b110)
                dice_value <= 3'b001;
            else
                dice_value <= dice_value + 1;
        end
    end
    reg [5:0] new_pos;
   reg [6:0] temp_sum;

 always @(posedge clk or posedge rst_full) begin
    if (rst_full)
       player_pos <= 6'd0;
    else if (btn_roll_edge) begin
       temp_sum = player_pos + dice_value;
       if (temp_sum > 6'd63)
          new_pos = player_pos;
       else
          new_pos = temp_sum[5:0];


       if ((new_pos == 6'd9) || (new_pos == 6'd24) || (new_pos == 6'd18) ||
 (new_pos == 6'd59))
          player_pos <= snake_ladder(new_pos);
       else
          player_pos <= new_pos;
    end
 end


    // Snake & Ladder Function
    function [5:0] snake_ladder;
       input [5:0] pos;
       begin
          case (pos)
             6'd9: snake_ladder = 6'd31; // Ladder: position 10 jumps to 32
             6'd24: snake_ladder = 6'd6;  // Snake: position 25 falls to 7
             6'd18: snake_ladder = 6'd49; // Ladder: position 19 jumps to 50
             6'd59: snake_ladder = 6'd4; // Snake: position 60 falls to 5
             default: snake_ladder = pos;
          endcase
       end
    endfunction

    // Derive Player's LED Matrix Coordinates
    reg [2:0] player_row;
    reg [2:0] player_col;
```

```
always @(*) begin
   player_row = player_pos[5:3];  // Upper 3 bits: row
   player_col = player_pos[2:0];  // Lower 3 bits: column
end

// LED Matrix Refresh and Scanning Logic
reg [15:0] refresh_counter;
reg [2:0] row_index;
always @(posedge clk or posedge rst_full) begin
   if (rst_full) begin
      refresh_counter <= 16'd0;
      row_index <= 3'd0;
   end else begin
      refresh_counter <= refresh_counter + 1;
      if (refresh_counter >= 16'd50000) begin
         refresh_counter <= 16'd0;
         row_index <= row_index + 1;
      end
   end
end

// Blink Logic for Player Position
reg blink;
reg [24:0] blink_counter;
localparam BLINK_THRESHOLD = 25000000;

always @(posedge clk or posedge rst_full) begin
   if (rst_full) begin
      blink_counter <= 0;
      blink <= 1'b1;
   end else begin
      if (blink_counter == BLINK_THRESHOLD - 1) begin
         blink_counter <= 0;
         blink <= ~blink;
      end else begin
         blink_counter <= blink_counter + 1;
      end
   end
end
```

```
    //  ROM for Static Board
    reg [7:0] rom [0:7];
    initial begin
        rom[0] = 8'b01010000;
        rom[1] = 8'b00000010;
        rom[2] = 8'b00000100;
        rom[3] = 8'b10000001;
        rom[4] = 8'b00000000;
        rom[5] = 8'b00000000;
        rom[6] = 8'b00000010;
        rom[7] = 8'b00001000;
    end

    // Blinking and static rom integrating.
    reg [7:0] rom_data;
    always @(*) begin
        rom_data = rom[row_index];
    end

    reg [7:0] final_data;
    always @(*) begin
        if (row_index == player_row)
            final_data = rom_data | ((blink) ? (8'b00000001 << player_col) :
8'b00000000);
        else
            final_data = rom_data;
    end

    // LED Matrix Output Generation
    always @(*) begin
        reg [7:0] row_temp;
        row_temp = 8'hFF;
        row_temp[row_index] = 1'b0;
        row = row_temp;
        col = final_data;
    end


    always @(*) begin
        case (dice_value)
```

```
            3'b001: hex0 = 7'b1111001;
            3'b010: hex0 = 7'b0100100;
            3'b011: hex0 = 7'b0110000;
            3'b100: hex0 = 7'b0011001;
            3'b101: hex0 = 7'b0010010;
            3'b110: hex0 = 7'b0000010;
            default: hex0 = 7'b1000000;
        endcase
    end

endmodule
```