“LumiMaze: A Light-Guided Maze Game”
DLD Project Report

Muhammad Affan, Muhammad Hassan Shahzad, Hazeera Muhammad Hashim, Tanzeel Shahid, Fatima Tu Zahra

CS-130L - Digital Logic and Design

Section - T1

Semester - Fall 2024

Table of Contents

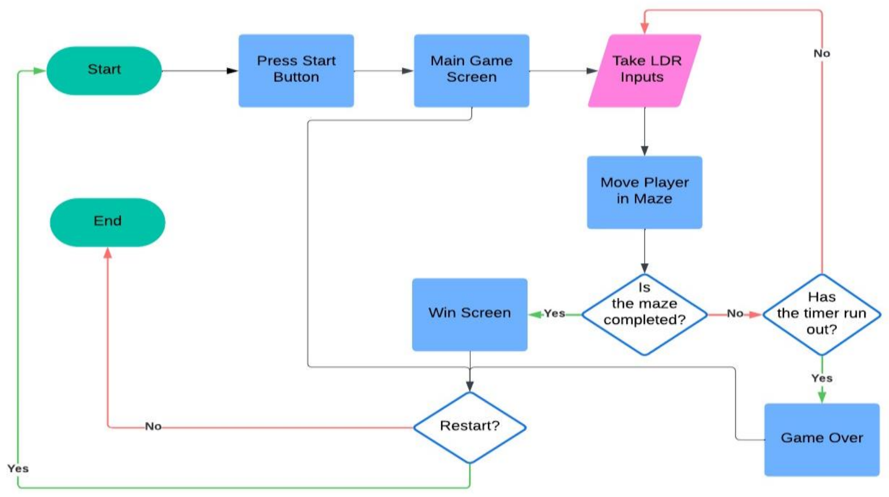
I. Project Description

LumiMaze adds a twist to the classic maze game by allowing you to navigate through the challenging maze through the power of light. Inspired by immersive design principles, the game employs Light Dependent Resistors (LDRs) as the primary mode of player input. The game comes to life on a VGA display, where the maze and the player's sprite are visually represented.

There are 4 different LDRs each dedicated to a direction the player wants to move in. The player will move in the direction represented by the LDR the user is shining the light on. Its movement is also time-bound, requiring swift navigation through the maze to reach the destination before the timer runs out.

The LDRs, placed strategically in 4 separate sections of a wooden box, respond to changes in light intensity, translating them into directional movements—shine light on the left LDR to move left, the right to move right, and so on. As the player progresses, the sprite dynamically updates its position on the screen, providing real-time feedback.

II. User-Flow Diagram



The figure above illustrates the user-flow experience

The user flow diagram for the game illustrates the sequence of actions and decisions a player experiences while playing LumiMaze.

1. Start Screen: The game begins at a start screen where the player can press "Enter" to begin.

2. Main Game Screen: Once the user clicks the start button the screen transitions to the main game view where the maze and the player's sprite (a UFO) is displayed.

3. Taking LDR Inputs: In the game state, LDR inputs are taken. These inputs determine the player's directional controls based on which LDR the light shines on.

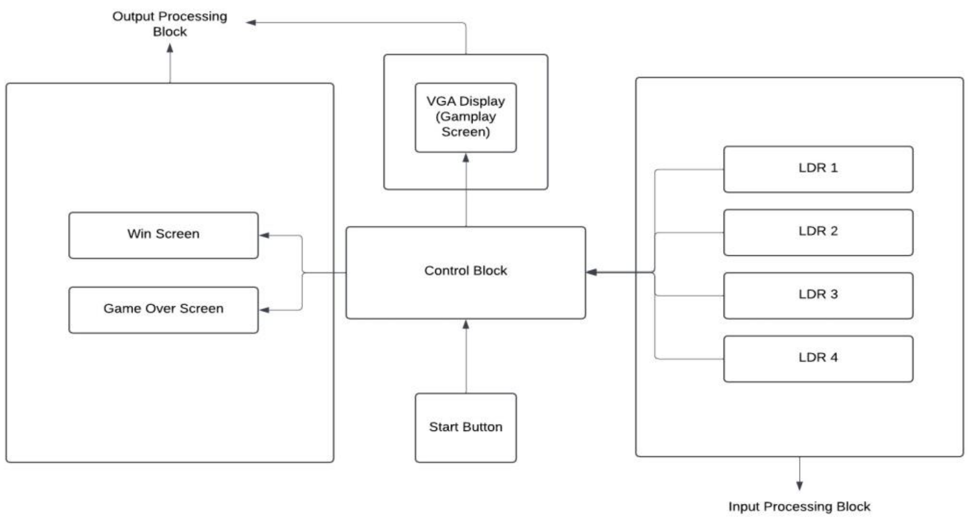
4. Gameplay - Movement in the Maze: The player navigates the UFO through the maze using the light-based LDR controls. The game actively checks for the following conditions:

- **Collision with Walls**:
  - The game continuously checks if the UFO collides with the maze walls, ensuring it stays within the open areas and cannot move through the walls.
- **Maze Completion**:
  - If the UFO successfully navigates through the maze (collision with the destination), the player is taken to the **Win Screen**.

5. Timer: If the timer ends before the UFO is able to reach the destination, the game ends, and the player is taken to a **Game Over** screen.

6. Restart: During the game, after completing the maze or encountering a game-over scenario, the player can choose to restart the game.

III. Module Block Diagram



Block Diagram connecting our input, control, and output blocks

3.1. Input Processing Block:

The Input Block uses **Light Dependent Resistors (LDRs)** as primary input peripherals to detect directional commands. By shining a laser on specific LDRs, the player can control the movement of a sprite in the maze game. Each LDR corresponds to a specific movement direction within the game (left, right, up, and down). The LDRs are connected to the BASYS 3 FPGA board, which processes the signals and translates them into movement commands.

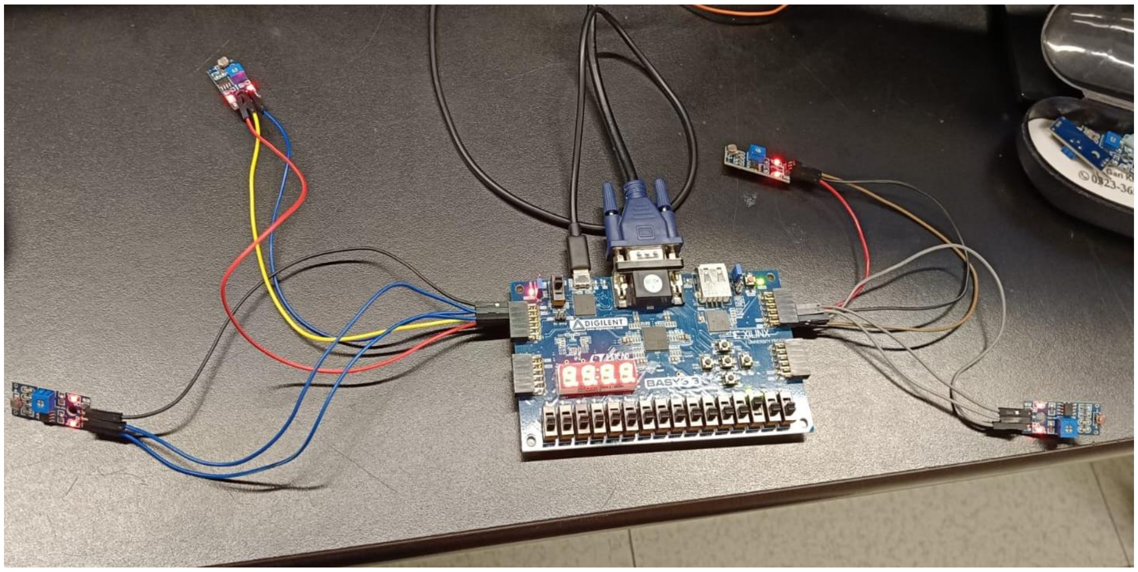
3.1.1 LDR Setup

LDRs are light-sensitive resistors that change their resistance based on the intensity of light falling on them. In this project:

- **Laser Interaction**: The player shines a laser pointer on an LDR to trigger movement in the corresponding direction.
- **Calibration**: Each LDR is calibrated to detect intentional laser signals while ignoring ambient light.
- **Resistance Mechanism**: The resistance of the LDR decreases with increased light intensity. When the resistance falls below a set threshold, the FPGA recognizes the input as active.
- **Direction Mapping**:
  - Left LDR → Moves sprite left.
  - Right LDR → Moves sprite right.
  - Upper LDR → Moves sprite up.
  - Lower LDR → Moves sprite down.

Four LDR modules are connected to the BASYS 3 FPGA, each representing one of the directional inputs.

Below is an image of the physical setup, showing the BASYS 3 FPGA board connected to the LDR modules. Each module is wired to the FPGA to detect signals from the laser and translate them into movement commands.



The above image of the BASYS 3 FPGA board connected to the LDR modules

3.1.2 LDR to FPGA Connection

Each **Light Dependent Resistor (LDR) module** in this project has three pins:

1. **VCC**: Provides power to the LDR module.
2. **GND**: Ground connection.
3. **D0 (Output)**: Digital output signal from the LDR module, indicating whether the light threshold has been exceeded.

Sample LDR used in the project, three pins mentioned can be seen

- The **VCC pin** of each LDR module is connected to the power supply pin on the BASYS 3 FPGA board.
- The **GND pin** of each LDR module is connected to the ground (GND) pin on the FPGA board.

- The **D0 pin** of each LDR is connected to one of the GPIO (General Purpose Input/Output) pins on the FPGA. This pin transmits a digital signal (HIGH or LOW) to the FPGA, representing whether the light intensity on the LDR has exceeded the threshold. In this project, the **D0 pins** of the four LDR modules are connected to four separate GPIO pins on the FPGA to differentiate directional inputs:
    - LDR1 → Pin G2 on FPGA [input 1]
    - LDR2 → Pin G3 on FPGA [input 2]
    - LDR3 → Pin B16 on FPGA [input 3]
    - LDR4 → Pin C16 on FPGA [input 4]

The connections are configured in the constraints file to map the physical pins of the FPGA to the LDR inputs.

Inputs

- **Input1** → Up LDR signal.
- **Input2** → Down LDR signal.
- **Input3** → Left LDR signal.
- **Input4** → Right LDR signal.
- **Clk** → Clock signal for synchronous processing.

Outputs

- **Out** → A 4-bit binary output, where each bit corresponds to one direction:
    - 0001: Upward movement.
    - 0010: Downward movement.
    - 0100: Left movement.
    - 1000: Right movement.

Direction Encoding

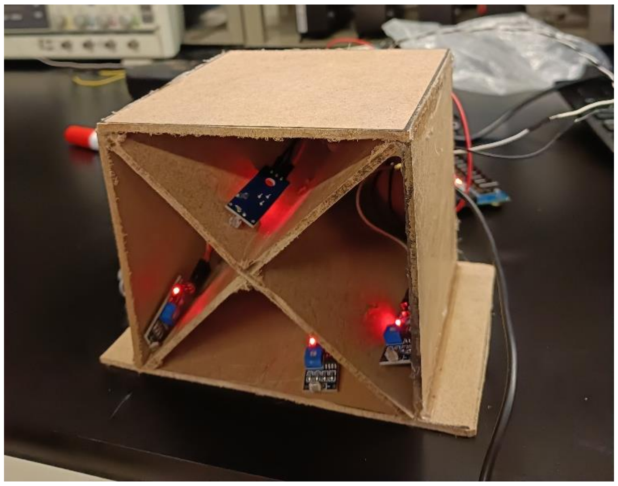
- Each LDR input has a specific priority:
    - **input1 (Up)** → Highest priority
    - **input2 (Down)**
    - **input3 (Left)**
    - **input4 (Right)**
- The module encodes the detected input into a 4-bit output signal.

Priority-Based Output Assignment

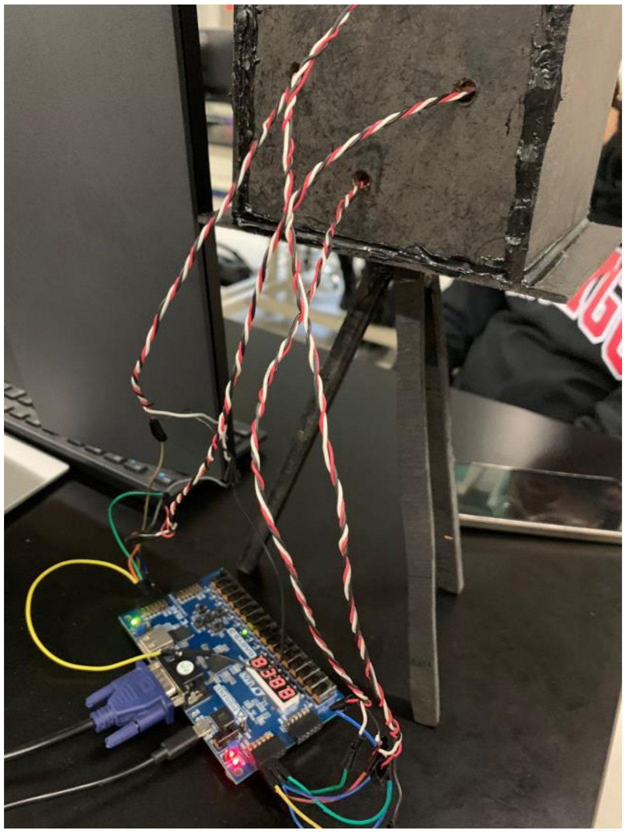
- If **input1** is active, Out is set to 0001.
- If **input1** is inactive but **input2** is active, Out is set to 0010, and so on.

The following Verilog module processes the LDR inputs and generates a 4-bit output:

See Appendix A for the full code of the LDR Integration Module.



Box containing four LDRs for each direction



LDRs Box (painted black, with a stand)

Backside of the LDR box

Above two images display the final version of the project's hardware component. The LDRs are strategically placed within the inside of the box, to prevent the surrounding light exposure from hindering the input values. We have also included a stand to place the box at an appropriate level with the maze game, aiding in a smooth User Experience.

Wires from each LDR are curled at the back of the box and connected with the jumper wires (displayed on the picture) which are then linked with the FPGA board.

3.1.3 Conclusion

The Input Module effectively processes signals from the LDR modules and generates directional commands for controlling the maze game sprite. By prioritizing inputs and ensuring robust signal calibration, the module ensures smooth and responsive gameplay.

3. 2. Output Processing Block:

3.2.1 Key Modules and Their Functions

1. **TOP_LEVEL**

The top-level module integrates all submodules for a VGA-based system with interactive control using Light Dependent Resistors (LDRs). It manages clock division, VGA synchronization, input integration from LDRs, and pixel generation for the display.

Inputs:

- Start: Signal to initiate the functionality of the system.
- Restart: Signal to reset or restart the system functionality.
- Clk: Master clock signal (typically 100 MHz or 50 MHz).
- LDR1: Signal from the first Light Dependent Resistor.
- LDR2: Signal from the second Light Dependent Resistor.
- LDR3: Signal from the third Light Dependent Resistor.
- LDR4: Signal from the fourth Light Dependent Resistor.

Outputs:

- clk_d: Divided clock signal for driving the VGA display.
- h_sync: Horizontal sync signal for VGA timing.
- v_sync: Vertical sync signal for VGA timing.
- red [3:0]: 4-bit signal controlling the red color intensity.
- green [3:0]: 4-bit signal controlling the green color intensity.
- blue [3:0]: 4-bit signal controlling the blue color intensity.

Functionality:

The TOP_LEVEL module orchestrates the interaction between its submodules to:

- Clock Management: Use the clk_div module to generate a divided clock (clk_d) suitable for VGA timing.
- LDR Signal Processing: Convert input signals from the LDRs into a binary signal (keys) using the LDR_integration module.
- VGA Synchronization: Generate horizontal (h_sync) and vertical (v_sync) synchronization signals and track the pixel positions (x_loc, y_loc) with the h_counter and v_counter modules.
- Pixel Generation and Game Logic: Control the RGB color signals (red, green, blue) for the VGA display based on the current pixel location, the keys signals from the LDRs, state transitions, the start and restart inputs. This is managed by the pixel_gen module.

Pin Configuration

Signals	Pin Configuration on FPGA
LDR1	G2
LDR2	G3
LDR3	B16
LDR4	C16
clk	W5
clk_d	U14
h_sync	P19
v_sync	R19
start	T18
restart	T17
red [3]	N19
red [2]	J19
red [1]	G19
red [0]	H19
green [3]	D17
green [2]	G17
green [1]	H17

green [0]	J17
blue [3]	J18
blue [2]	K18
blue [1]	L18
blue [0]	N18

2. **PIXEL_GEN**

The pixel_gen module generates RGB color signals (red, green, blue) based on the current pixel position (pixel_x, pixel_y) and the state of the game. It handles transitions between a start screen and the game screen as well as the win and lose screens. It uses an FSM (Finite State Machine) to control these transitions.

Inputs:

- Clk_d: Divided clock signal, used as the timing source for state transitions and logic.
- Pixel_x: Horizontal pixel position on the display (10-bit).
- Pixel_y: Vertical pixel position on the display (10-bit).
- Video_on: Indicates whether the current pixel is within the active video region.
- Keys: 4-bit input signal controlling the player's position or actions.
- Start: Signal to start the game and transition from the start screen to the game screen.
- Restart: Signal to reset the game and return to the start screen.

Outputs:

- Red [3:0]: 4-bit signal controlling the red color intensity of the pixel.
- Green [3:0]: 4-bit signal controlling the green color intensity of the pixel.
- Blue [3:0]: 4-bit signal controlling the blue color intensity of the pixel.

Functionality:

- FSM Behavior:

  START State: Displays the start screen:
  - If a pixel belongs to the start screen (s1_pix is high), the color is set to white.
  - Otherwise, the background is set to black
  - Sets framecount to 0 and timer digits to 5 and 9

  GAME State: Displays the game screen:

    - Checks if the player has reached destination by taking player’s x_position and y_position from the player module.
    - If a pixel corresponds to the player's position (player_pix is high), the color is set to white.
    - If a pixel belongs to timer (firstnum_flag or secondnum_flag is high) set color to white.
    - If a pixel belongs to maze’s wall (maze_pix is high) set color to red.
    - If a pixel belongs to the destination (dest is high) set color to yellow.
    - Otherwise, the background is dark blue.

  WIN State: Displays the winning screen.
    - If a pixel belongs to the screen letters (win_pix is high) set color to black.
    - The background color is set to blue.

  LOSE State: Displays the losing screen.
    - If a pixel belongs to the screen letters (lose_pix is high) set color to black.
    - The background color is set to red.

- Transition Logic:
    - The FSM transitions from START to GAME on the startgame signal.
    - It transitions from GAME to START state on the restart signal.
    - It transitions from GAME to WIN state upon reaching the destination.
    - It transitions from GAME to LOSE state if the timer ends.
    - It transitions from WIN/LOSE to START state on the restart signal.

- Color Logic:
    - During active video regions (video_on is high), colors are updated based on the FSM state and submodule outputs (s1_pix, maze_pix, lose_pix, win_pix, dest and player_pix).
    - During inactive video regions (video_on is low), all colors are set to black.

3. **START SCREEN**

The start screen module generates a 30x30 pixel static bitmap for a start screen. This bitmap defines a border and a graphical design that is used as a starting display in our VGA system.

Inputs:

- Pixel_x: Horizontal pixel position on the screen (10-bit).
- Pixel_y: Vertical pixel position on the screen (10-bit).

Outputs:

- Flag: High if the current pixel belongs to the defined pattern, low otherwise.

Functionality:

It determines if the current pixel (pixel_x, pixel_y) is part of the start screen based on scaled-down coordinates and outputs the flag signal.

1. **Grid Definition:**

- The 30x30 grid is defined as a 2D register array where each row contains 30 bits.
- Each bit represents whether a specific pixel is part of the start screen (1) or the background (0).

2. **Coordinate Scaling:**

- The pixel_x and pixel_y inputs (10 bits wide) are scaled down to match the 30x30 grid using the following logic:
  x = pixel_x[9:4] - 5: Maps horizontal pixel position to the 30x30 grid with an offset of 5.
  y = pixel_y[9:4]: Maps vertical pixel position to the grid.
- This scaling reduces the full pixel resolution (e.g., 640x480) to match the smaller grid.

4. **PLAYER MODULE**

The player module manages the player's position on the screen, handles collision detection with maze pixels, and determines whether the current pixel belongs to the player's 16x16 graphical representation. It supports position updates based on input keys, resets functionality, and incorporates collision handling for a more interactive movement system.

Inputs:

- clk: Clock signal to drive the logic for animation and position updates.
- pixel_x: Horizontal pixel position on the display (10-bit).
- pixel_y: Vertical pixel position on the display (10-bit).
- keys: 4-bit input signal controlling the player's movement
- reset: Signal to reset the player's position to its initial coordinates
- maze_pix: Input signal indicating if a pixel belongs to a maze wall (used for collision detection).

Outputs:

- player_flag: High if the current pixel belongs to the player’s 16x16 sprite, low otherwise
- player_x_pos: current x position of the player sprite

- Player_y_pos: current y position of the player sprite

Functionality:

- Player Position
    - Initial Position:
        - When the reset signal is high, the player's position is reset to its starting coordinates:
          x_pos = 511
          y_pos = 390

    - Movement Logic and Collision Detection:
      The player's position updates based on the keys input:
        - 4'b0001: Move up by decrementing y_pos.
        - 4'b0010: Move down by incrementing y_pos.
        - 4'b0100: Move left by decrementing x_pos.
        - 4'b1000: Move right by incrementing x_pos.

    - Collision detection is based on maze_pix. If a collision is detected in a direction, the movement in that direction is restricted or reversed:
        - Up: Decrement y_pos unless collision (collision [0]), otherwise increment.
        - Down: Increment y_pos unless collision (collision [1]), otherwise decrement.
        - Left: Decrement x_pos unless collision (collision [2]), otherwise increment.
        - Right: Increment x_pos unless collision (collision [3]), otherwise decrement.

    - The maze_pix input is used to detect potential collisions with the maze. Collisions are tracked using a 4-bit collision signal:
    - collision [0]: Collision detected above the player.
    - collision [1]: Collision detected below the player.
    - collision [2]: Collision detected to the left of the player.
    - collision [3]: Collision detected to the right of the player.

  Collisions are reset at the start of each VGA frame (pixel_y == 481 and pixel_x == 0).

- New Frame Control:

  New_frame updates occur when pixel_y == 480 and pixel_x == 0, ensuring synchronization with the VGA frame.

- Pixel Ownership:
    - The player_flag signal determines if the current pixel belongs to the player's sprite
    - The player is rendered as a 16x16 pixel array defined in player_arr:
        - 1 indicates the presence of the sprite.
        - 0 indicates transparent space.

5. **MAZE MODULE**

The maze module defines a 30x30 grid representing a maze layout. It determines whether a given pixel on the screen belongs to a maze wall based on its position in the grid. The module uses a binary matrix to encode the maze structure, where 1 indicates a wall and 0 represents empty space.

Inputs:

- pixel_x: Horizontal pixel position on the display (10-bit).
- pixel_y: Vertical pixel position on the display (10-bit).

Outputs:

- flag: High if the current pixel corresponds to a wall in the maze, low otherwise.

Functionality:

- Maze grid:
    a. The maze is represented by a 30x30 matrix (m), where:
        i. Each row (m[y]) represents a horizontal section of the win screen.
        ii. Each bit within a row corresponds to a 16x16 pixel block in the display.
- Mapping Pixel to Maze Block:
    b. The horizontal (x) and vertical (y) indices within the maze are derived from the pixel coordinates:
        i. x = pixel_x[9:4] - 5
        ii. y = pixel_y[9:4]
    c. This maps each 16x16 pixel block in the display to a single cell in the maze grid.
- Flag Assignment:
    d. The flag output is assigned based on the value of the corresponding cell in the maze grid:
        i. flag = m[y][x]

Maze Layout:

- The maze matrix (m) is initialized as follows: 1 represents a maze wall, and 0 represents open space.

6. **WIN SCREEN MODULE**

This module draws a pattern of the win screen inside a 30 by 30-bit array (win).

Inputs:

- pixel_x (10-bit): Horizontal pixel position on the display.
- pixel_y (10-bit): Vertical pixel position on the display.

Outputs:

- flag: High if the current pixel corresponds to the patter of the win screen

Functionality:

- Win screen Grid:
  - The maze is represented by a 30x30 matrix (win), where:
    - Each row (win[y]) represents a horizontal section of the maze.
    - Each bit within a row corresponds to a 16x16 pixel block in the display.
- Mapping Pixel to win screen block:
  - The horizontal (x) and vertical (y) indices within the win matrix are derived from the pixel coordinates:
    - x = pixel_x[9:4] - 5
    - y = pixel_y[9:4]
- Flag Assignment:
  - The flag output is assigned based on the value of the corresponding cell in the maze grid:
    - flag = win[y][x]

7. **LOSE SCREEN MODULE**

This module has the exact same functionality as the win screen module, just the pattern in the bitmap is different and the FSM calls it when the player loses the game and state shifts from GAME to LOSE

8. **DESTINATION MODULE**

The destination module is responsible for determining whether a pixel on the VGA display corresponds to the destination sprite defined using a 16x16 bit array. It outputs high flag if the pixel falls within the bounds of the destination sprite

Inputs:

- pixel_x (10-bit): Horizontal pixel position on the display.
- pixel_y (10-bit): Vertical pixel position on the display.

Outputs:

- flag: High if the current pixel corresponds to the pattern of the destination module

Functionality:

The arrow is represented by a 16x16 matrix (Dest), where:

- Each row represents a horizontal slice of the arrow sprite.
- Each bit within a row corresponds to a single pixel in the 16x16 block of the display.

Flag Assignment:

- If the current pixel is within the bounds of the destination sprite from (200, 40) to (216, 56) the flag is set high
- Else it is set to low

9. **COUNTER_NUMS MODULE**

The module *Counter_nums* is responsible for rendering a single digit (0-9) as a sprite on a VGA display. Each digit is represented by a unique pattern of pixels.

Inputs:

- Clk_d: clock signal
- xloc (10-bit): Horizontal pixel position on the display.
- yloc (10-bit): Vertical pixel position on the display.
- charx: x pixel of the location to display the number
- chary: y pixel of the location to display the number
- num: number to be displayed from 0-9

Output:

- sprite_on: indicates if current pixel corresponds to the number being rendered

Functionality:

Digit representation:

- Each digit (0-9) is defined as a combination of pixel regions forming its shape
- These regions define the on-pixels for each digit relative to the top-left corner of the digit (charx, chary).

Flag Assignment:

- At each clock cycle, the sprite_on flag is activated if the current pixel corresponds to the selected digit (num).
- The module uses a sequential block to assign the corresponding sprite (e.g., zero, one, etc.) based on the num input.
- If num is not in the range 0-9, the sprite_on flag is deactivated (0).

10. **OTHER MODULES**

In this project, we have used several Verilog modules to handle essential and synchronization for the sprite movement control. The modules not only include the ones we have mentioned above, but also the ones we used in previous Labs. These include:

1. clk_div (Clock Divider): This module divides the main system clock by a specified value to generate a slower clock signal (clk_d). It ensures that the FPGA processes signals at appropriate intervals, preventing excessive speed and allowing for precise control over timing-related operations.

2. h_counter (Horizontal Counter): The h_counter module counts horizontal pixel positions for display synchronization. It generates a 10-bit horizontal count (h_count) and a trigger signal (trig_v) that is used to manage horizontal refresh cycles. When the count reaches 799, it resets and triggers the vertical counter.

3. v_counter (Vertical Counter): Like the h_counter, the v_counter tracks vertical pixel positions. It operates in conjunction with the h_counter to form a complete frame by counting to 524 and resetting. It also controls the enabling of vertical synchronization signals to properly display the game's graphics.

3.3. Control Block:

1. FSM for Game Logic

The FSM will control the transitions between different game states, such as "Start Screen," "Gameplay," "Game Over," and "Win Screen." It is **Mealy machine**, since the outputs depend on both the current state and the input. For example, in the **Gameplay** state (S1), the FSM reacts to inputs such as LDR (Light Dependent Resistor) inputs for directional movement and outputs actions like updating the player's position, checking for collisions, and detecting maze completion. The output action is tied to the input (e.g., player movement) and the current state (Gameplay).

States:

1. **Start Screen** (S0): The initial state where the game waits for the player to press a button (e.g., Enter) to start.
   Waits for player input. Transitions to Gameplay when the "Startgame" signal is received.

2. **Gameplay** (S1): The state where the player navigates the maze, and the sprite moves according to LDR inputs. Collision detection and maze completion checks occur here.

   Responds to LDR inputs, updates sprite position, checks for collisions, and detects maze completion. Transitions to Game Over if the timer ends or to Win Screen if the player reaches the destination.

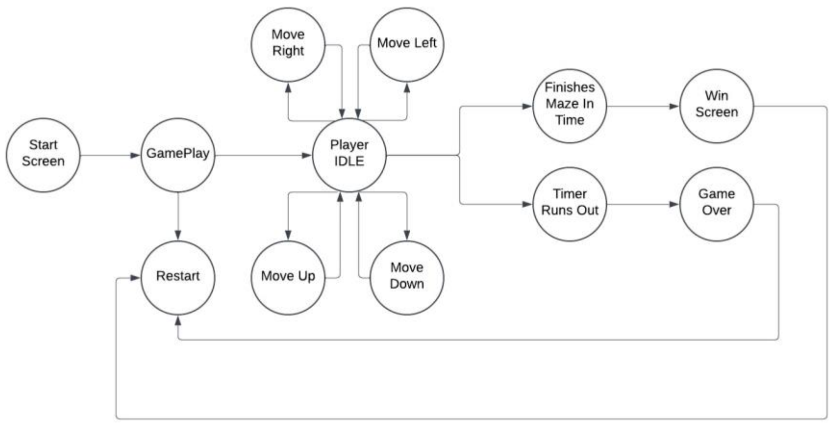
   Transitions to Start Screen if player clicks the restart button.

3. **Game Over** (S2): Activated if the timer ends before the player reaches the destination. Displays the 'You Lose' text and waits for a restart.

4. **Win Screen** (S3): Triggered when the player successfully completes the maze. Celebrates the win and waits for a restart.

FSM Transitions:

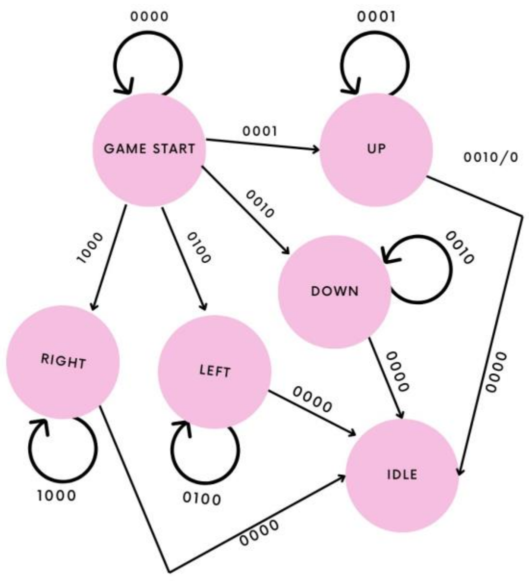
Current State	Inputs	Next State	Output (Action)
Start Screen	Start Button Press	Gameplay Screen	Display Maze Screen, Timer, Initialize Game
Gameplay Screen	LDR Inputs (Directional Movement)	Gameplay Screen	Update Player's Position on Screen
Gameplay Screen	Timer Ends	Game Over Screen	Display Game Over Screen
Gameplay Screen	Maze Completion (Player Reaches End)	Win Screen	Display Win Screen
Gameplay Screen	Reset Button Press	Start Screen	Reset Game State, Return to Start Screen
Game Over Screen	Reset Button Press	Start Screen	Reset Game State, Return to Start Screen
Win Screen	Reset Button Press	Start Screen	Reset Game State, Return to Start Screen



State Transition Diagram

2. FSM for Player Movement (LDR-based Input)

For controlling sprite movement based on LDR inputs, a **Mealy FSM** is ideal. Each LDR corresponds to a direction, and the FSM will manage the sprite’s position in real-time based on which LDR the light shines on.



Player FSM

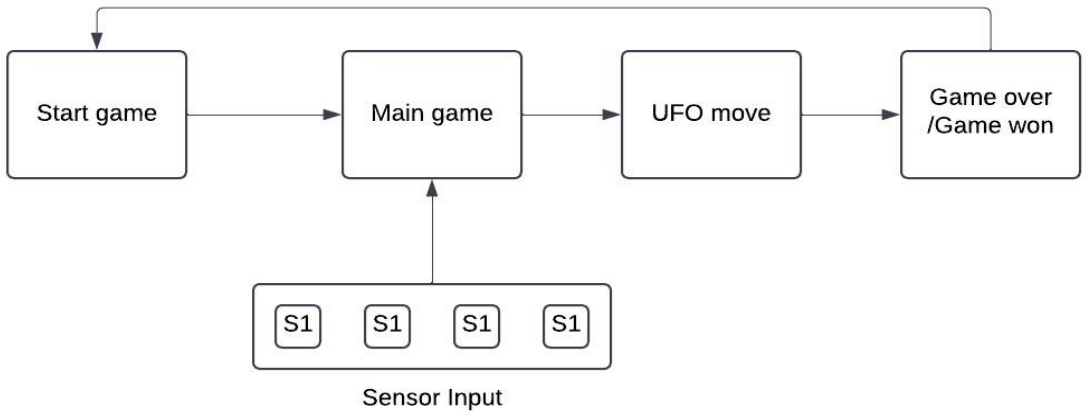
States:

1. **Idle**: No movement (when no LDR is activated).
2. **Move Up**: Player moves upward when the "Up LDR" (input1) is activated.
3. **Move Down**: Player moves downward when the "Down LDR" (input2) is activated.
4. **Move Left**: Player moves left when the "Left LDR" (input3) is activated.
5. **Move Right**: Player moves right when the "Right LDR" (input4) is activated.

Transition Logic:

- When a light is detected on an LDR, the FSM will transition to the corresponding state and update the player's position.
- The FSM will return to the **Idle** state when no LDR input is active.
- **Idle to Move Up**: If input1 (Up LDR) is active, the state transitions to Move Up.
- **Move Up to Idle**: If no LDR input is active, the state transitions to Idle.

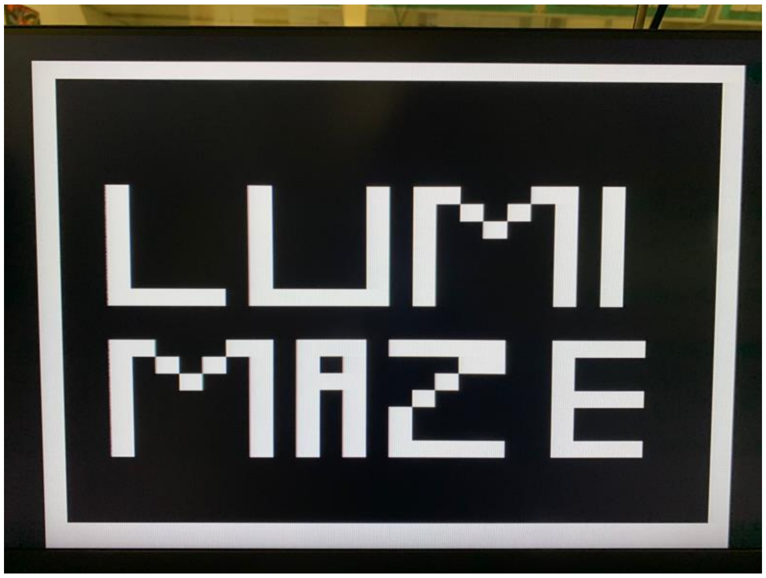
IV. The Game Logic



The above illustrates the GUI Block Diagram

Display Screen (Start Game):

- Visual Layout: A black screen appears with the word 'LUMIMAZE' on it written in white.
- Pixel Logic: The bit value is set to 1 where white color needs to be rendered and 0 for black.

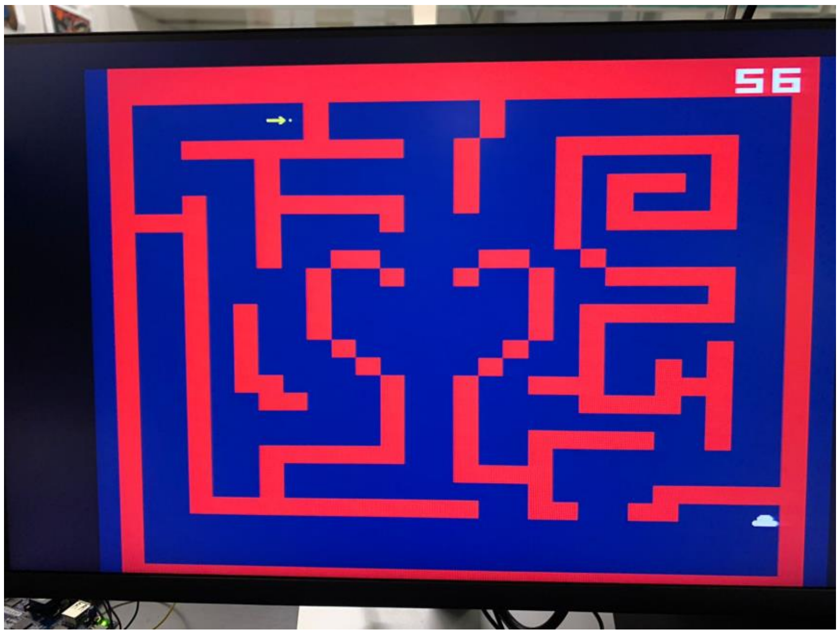


Display Screen

Main screen (Main Game):

- Visual Layout: A dark blue screen appears with a maze structure displayed in red. The maze contains paths for the player to navigate, with the UFO sprite visible as a white object within the maze.

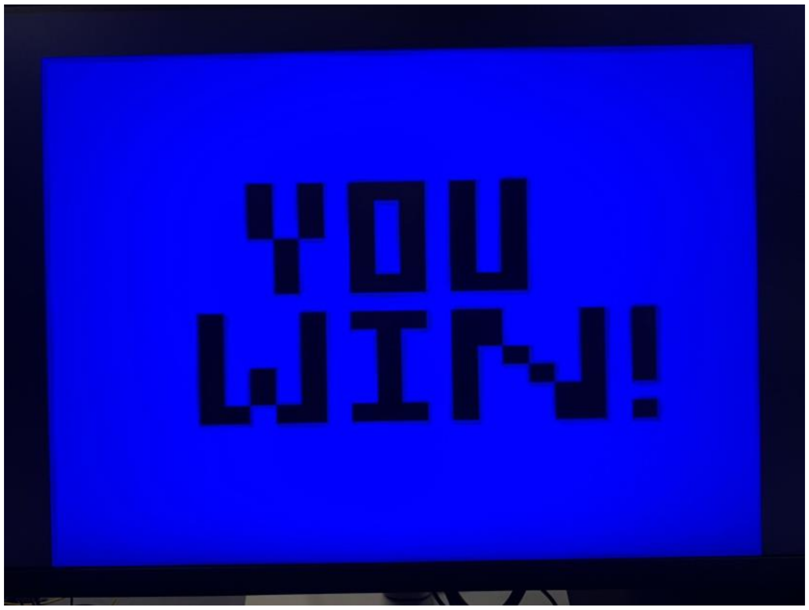
- Pixel Logic: The bit value is set to 1 where red color (maze walls) needs to be rendered and 0 for dark blue (open paths). The UFO sprite's position is dynamically updated based on movement inputs.



Game Screen

Win screen:

- Visual Layout: A blue screen appears with the words ‘You Win!’ displayed on it.
- Pixel Logic: The bit value is set to 1 where black color (word) needs to be rendered and 0 for blue (background).



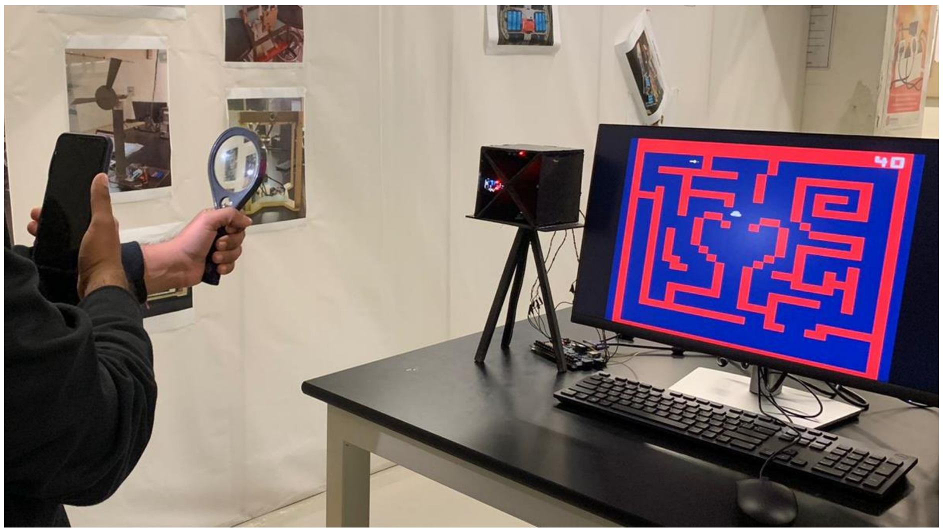
Win Screen

Lose screen:

- Visual Layout: A red screen appears with the words ‘You Lose!’ displayed on it.
- Pixel Logic: The bit value is set to 1 where black color (word) needs to be rendered and 0 for red(background).

Lose Screen

LDR Sensors (UFO Move):



Gameplay

- When light falls on the sensor, it sends a signal into the FPGA board which is later processed for sprite’s movement.

The shape of the sprite resembles a UFO.

- It can move in 4 directions based on the incoming LDR input. Each LDR input is assigned a 4-bit value which then determines the sprite’s position.
    - 4’b0001 (Upward movement)
    - 4’b0010 (Upward movement)
    - 4’b0100 (left movement)
    - 4’b1000(Right movement)

Reset Game:

- At any point during the game, if the reset button is pressed. The program changes its state back to the main screen and the game starts all over again.

V. Conclusion

The *LumiMaze* project was an exciting adventure that pushed our creativity and teamwork to new levels. We turned the classic maze game into something fresh and innovative by using Light Dependent Resistors (LDRs) as controls, letting players navigate the maze with a beam of light. Paired with a vibrant VGA display, the game delivered a fun and immersive experience that felt like steering a UFO through a glowing labyrinth. Our goal was simple: build a functional, interactive maze game on time—and we did it! From smooth movement to collision detection and a race-against-the-clock challenge, every piece came together as planned. Completing the project on schedule and hitting all our objectives was a win we’re proud of. But we’re not stopping here. We’re already thinking of ways to take *LumiMaze* to the next level. Imagine dodging obstacles, managing lives, and navigating even tougher mazes—these additions will make the game more challenging and fun, keeping players coming back for more.

Throughout this journey, teamwork was key. We learned how to divide tasks, stay organized, and turn our collective ideas into reality. It wasn't just about coding—it was about figuring things out together, tackling problems, and having each other's backs. In the end, *LumiMaze* isn't just a game—it's a project that showed us what we can achieve with innovation, collaboration, and a lot of determination. We're excited to build on this foundation and make future versions of the game even more thrilling!

VI. Project Division

The development of *LumiMaze* was a collaborative effort, with primary responsibilities assigned to specific team members, while all team members worked together to ensure the project's success.

- **Hardware Development**:

The tasks involved designing and implementing the Light Dependent Resistor (LDR) system, which served as the main control input for the game. Constructing the physical setup, ensuring that the LDRs were strategically placed within the wooden box for optimal functionality was one of the key tasks. This also included ensuring the smooth integration of the hardware with the game system and resolving any issues that arose during the testing phase.

Primary Responsibility taken by: Hazeera and Hassan

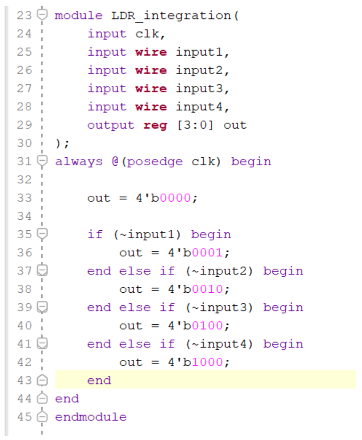
- **Software Development**:

Thus, focused on the software development, creating the core game logic, including movement controls based on LDR inputs, collision detection, and the timer system. Furthermore, developing the VGA display to visually represent the maze and player's sprite. The work ensured the dynamic display of the maze, transitions between different game states (start, game, win, and game-over), and the overall flow of the gameplay.

Primary Responsibility taken by: Tanzeel, Affan, and Fatima

While the primary responsibilities were divided as described, we all worked collaboratively, offering input, troubleshooting issues together, and ensuring that both hardware and software components seamlessly integrated to create a functional and immersive gaming experience. This collaboration was key to the timely and successful completion of the project.

VII. Appendix A: Verilog Code for Processing LDR Inputs



```
module LDR_integration(
    input clk,
    input wire input1,
    input wire input2,
    input wire input3,
    input wire input4,
    output reg [3:0] out
);
always @(posedge clk) begin

    out = 4'b0000;

    if (~input1) begin
        out = 4'b0001;
    end else if (~input2) begin
        out = 4'b0010;
    end else if (~input3) begin
        out = 4'b0100;
    end else if (~input4) begin
        out = 4'b1000;
    end
end
endmodule
```

Figure A1: Verilog module that processes 4-bit LDR inputs and generates corresponding 4-bit outputs based on the input patterns.

VIII. References

Affan002. (2024, December 12). *LUMIMAZE_DLDFinalProject: A maze game in Verilog with input from LDRs* [GitHub repository]. GitHub. Retrieved from https://github.com/affan002/LUMIMAZE_DLDFinalProject

[Our Project's GitHub Repository]

Cirkit Designer. (n.d.). *Module: LDR* [Documentation]. CirkitDesigner. Retrieved December 12, 2024, from https://docs.cirkitdesigner.com/component/9c93cd20-5ee8-44bd-b70e-f922819fa169/module-ldr

Digilent, Inc. (2019). *Basys 3™ FPGA board reference manual* (Rev. C). https://www.digilentinc.com