

# Digital Logic & Design - Final Project Report

Abdul Karim, Huzaifa Ali Khan, Sheikh Mohammad Muneeb,  
Syed Hamza Raza, Muhammad Zoraiz Azeem

Fall 2024

## CAR SMASH

### 1 Introduction

#### 1.1 Aim:

To depict the function of Car game using Verilog HDL, BASYS 3 Board, and Xilinx Vivado.

#### 1.2 About The Game:

This project aims to develop a fast-paced, real-time car racing game using an FPGA. Designed for two players, the game features cars at the bottom of a three-lane road. Players must navigate their cars left or right to avoid randomly generated obstacles falling from the top. At least one lane will always remain clear, ensuring challenging gameplay. Both players start with three lives, losing one upon collision with an obstacle. The player who survives the longest wins.

#### 1.3 Layout of the game design:

- Design a 3-lane road displayed on the screen.
- Two cars, controlled by two players, are positioned at the bottom of the screen.
- Input from a keyboard (WASD keys for Player 1 and arrow keys for Player 2) allows players to move their cars to avoid obstacles.
- Obstacles are randomly generated in the three lanes every 3 seconds, with at least one lane always being clear.
- If a car collides with an obstacle, the player loses one life. Both players start with 3 lives.
- The game continues until one player has no lives remaining. At this point, the game transitions to the Game Over state, and the other player is declared the winner.
- A reset button allows us to return to main screen, from where the game can be restarted.
- The game loop includes car movement, obstacle generation, collision detection, and life tracking, ensuring continuous and challenging gameplay.

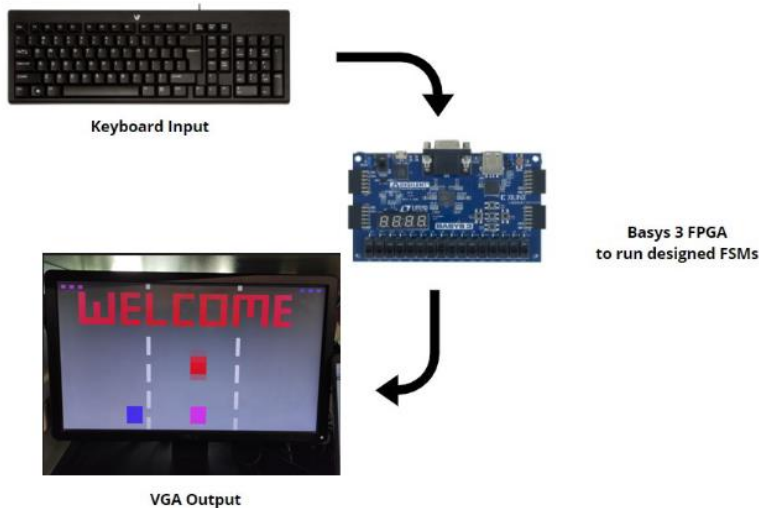


Figure 1: Game Layout

## 1.4 User Flow Diagram

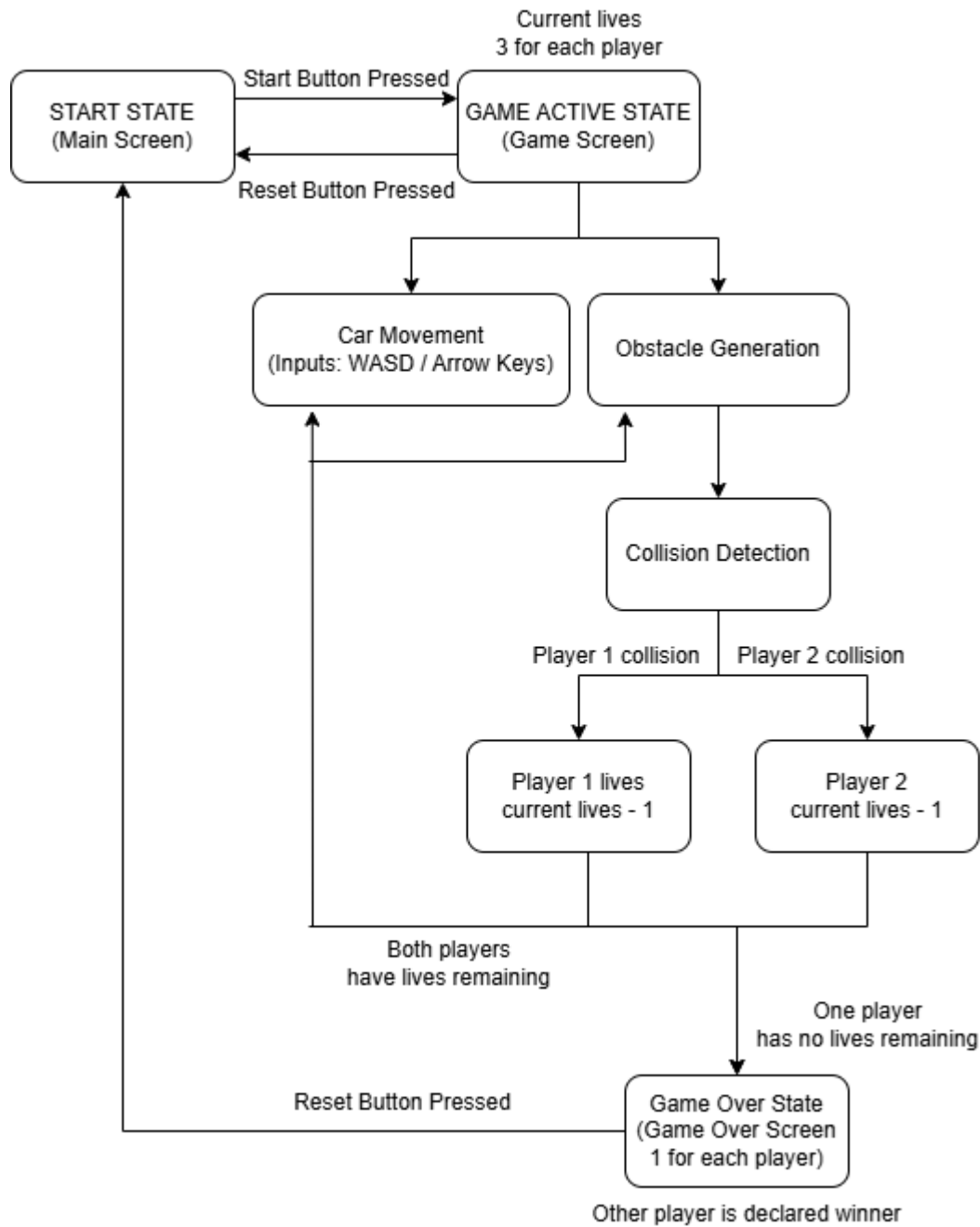


Figure 2: User Flow Diagram

The user flow diagram above provides an overview of the framework embedded within the game. After starting the game, the player will be able to view the following:

The user flow diagram above provides an overview of the framework embedded within the game. After starting the game, the player will be able to view the following:

- **Two cars** (blue and pink boxes) at the bottom of the screen, representing the players' positions. Each car moves based on the input keys.
- **Randomly generated obstacles** (red boxes) falling from the top of the screen in one of the three lanes.
- **Current lives** (displayed as 3 for each player initially).
- **Game screen** with continuous obstacle generation and car movement.

The player moves their car using the keyboard:

- Player 1 uses WASD.
- Player 2 uses arrow keys.

If a player's car collides with an obstacle, the corresponding player's lives decrease by 1. The game continues until one of the players loses all lives. At this point:

- The game transitions to the **Game Over State**, where the other player is declared the winner.

To restart the game, the reset button is pressed on the keyboard. This takes us back to the main screen, and reinitializes all variables, including the lives of both players. When the player presses the play button, the game restarts with 3 lives for each player.

## 2 Description of Modules

### 2.1 keyboard\_input

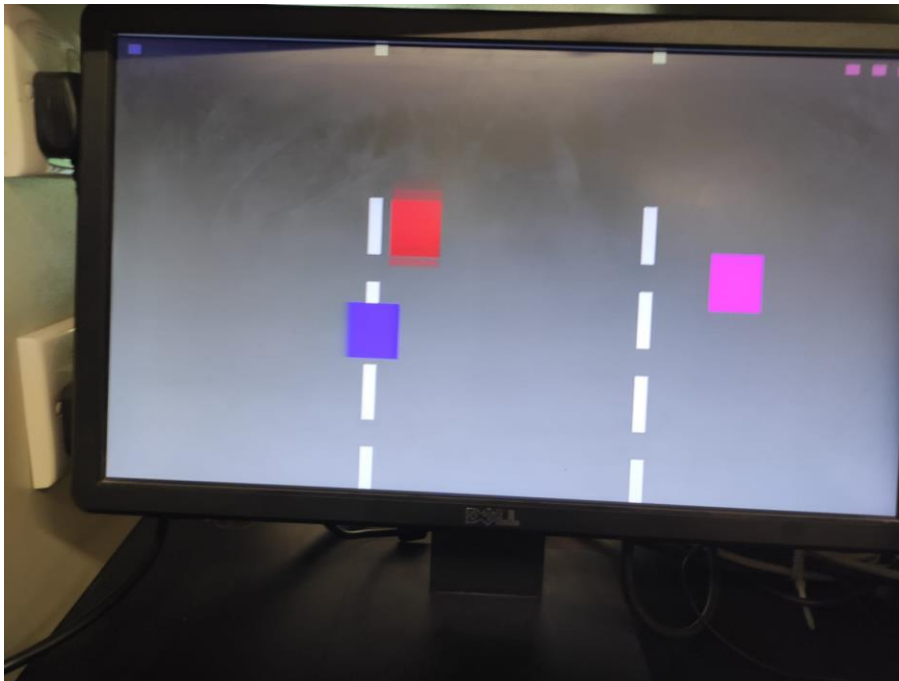


Figure 3

- The keyboard\_input module is designed to handle PS/2 keyboard inputs and translate them into actionable signals for gameplay. It processes the keyboard clock (kclk) and data (kdata) to detect key presses and releases. Key states are updated for two players, represented by keys\_1 and keys\_2, which correspond to movement directions.
- Using a state machine implemented with a counter, the module captures keyboard data byte-by-byte, synchronizing to the keyboard's clock using debounced signals. Press and release events are identified and mapped to specific actions, such as directional movement for players or control signals like start and restart.
- This implementation uses internal flags and temporary registers to manage state changes efficiently and ensure accurate processing of key events. The use of D-FlipFlops ensures reliable storage of data and state transitions within the module.

## 2.2 obstacle\_generator

- It uses a Linear Feedback Shift Register (LFSR) to create random horizontal positions (obs\_x) for obstacles.
- The vertical position (obs\_y) increments at a constant rate, simulating the downward movement of obstacles. When an obstacle reaches the bottom of the screen, the module resets the vertical position and generates a new random horizontal position.

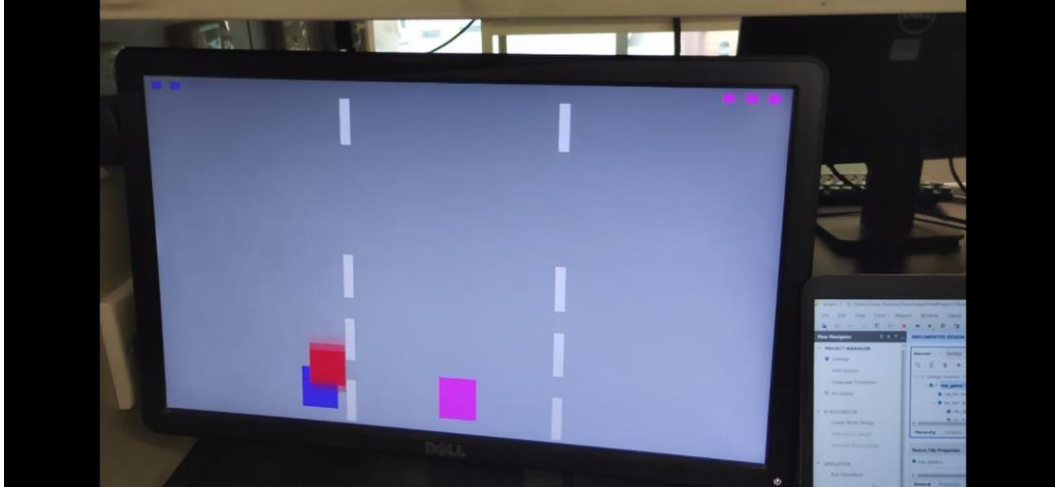
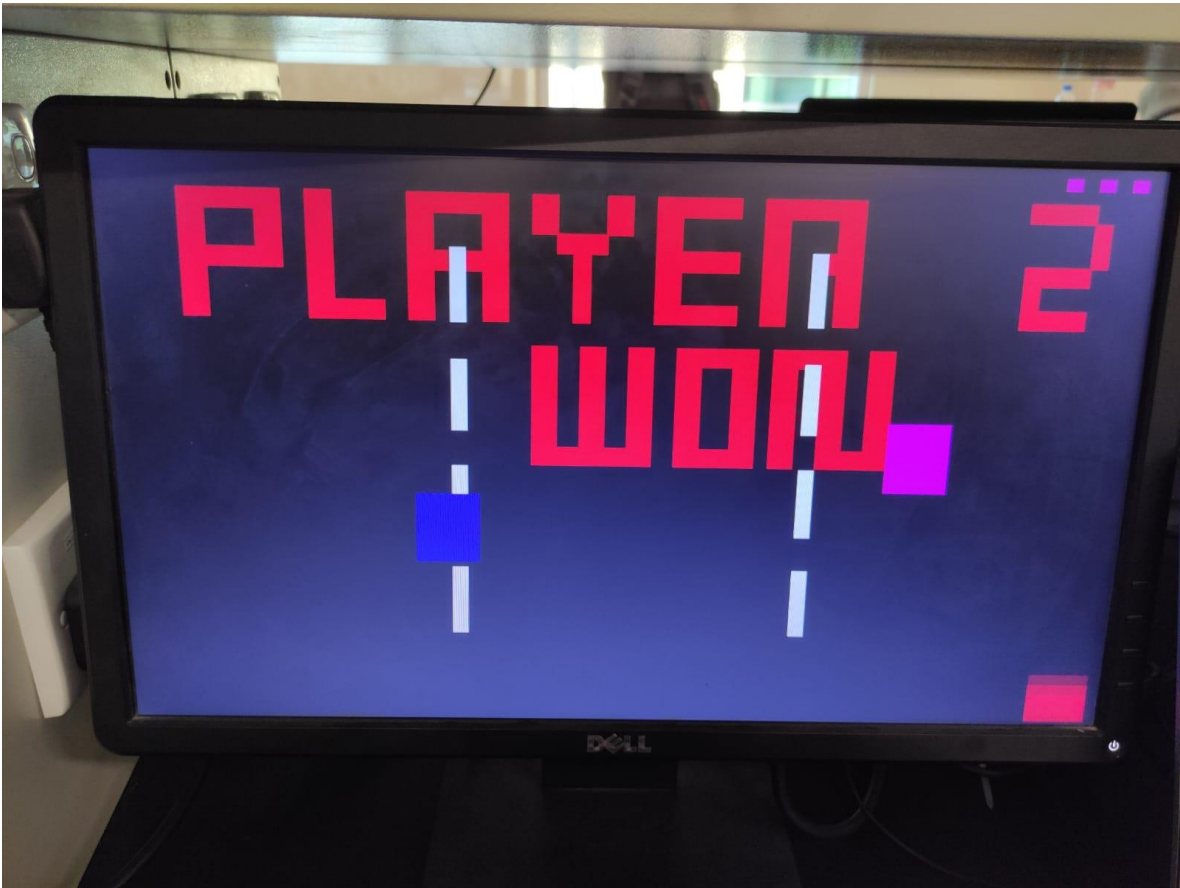


Figure 5

## 2.3 vga\_controller

- The module generates horizontal (vga\_hsync) and vertical (vga\_vsync) synchronization signals for a standard 640x480 resolution at 60 Hz refresh rate.
- Uses parameters for horizontal and vertical timing, including the active display area, front porch, sync pulse, and back porch timings.
- Implements smooth scrolling for road effects using road\_offset that updates every frame, managed by road\_clk.
- Displays required screen ("WELCOME" or "PLAYER 1/2 WON") based on game\_state input. Each letter is defined by a set of pixel conditions for its segments.
- Text is displayed in red against a black background (showing both inactive states).
- For each pixel the color is determined based on h\_count and v\_count, which give the current coordinates on the display.
- Resets internal counters (h\_count, v\_count, and road\_offset) to initial states when the reset button is pressed and rst signal is activated and checked.

## 2.4 top\_game



- The top\_game module serves as the central controller for the car racing game on the Basys 3 FPGA board. It takes inputs such as the clock signal, reset, and keyboard signals (ps2\_clk and ps2\_data) and processes them using the keyboard\_input module to generate key codes for player movement and game control. It coordinates the game logic, including player movements, obstacle generation, and collision detection.
- The game state transitions are managed using a Moore Finite State Machine, enabling dynamic gameplay with start and restart functionalities.

## 2.5 debouncer

- The debouncer module ensures that the input signal is stable and free from noise or glitches before it is processed.
- If the input signal fluctuates, the module waits until the signal remains consistent for a specified period, as defined by COUNT\_MAX, before updating the output. This guarantees that only valid, stable signals are passed on to the next module.

## 2.6 clk\_divider

- The clk\_divider module is designed to generate and manage the clock signal by dividing the input clock frequency.
- It uses a counter to track clock cycles, toggling the output clock signal (clk\_out) whenever the counter reaches a specified division value (div\_value). This ensures that the output clock operates at a frequency determined by the division factor, providing a consistent and reliable slower clock signal for other modules.

## 2.7 Output Block

### 1. Overview

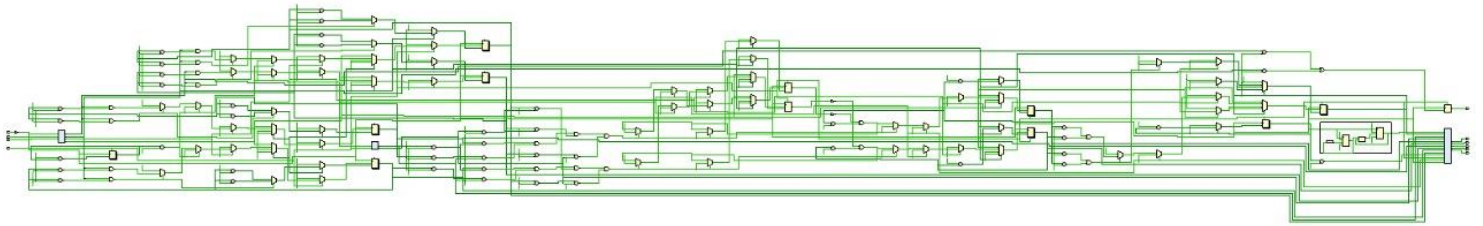


FIGURE 1: This picture shows the schematic for our top Game module. Next we will show a module view of the top game module

## 2. Modular View

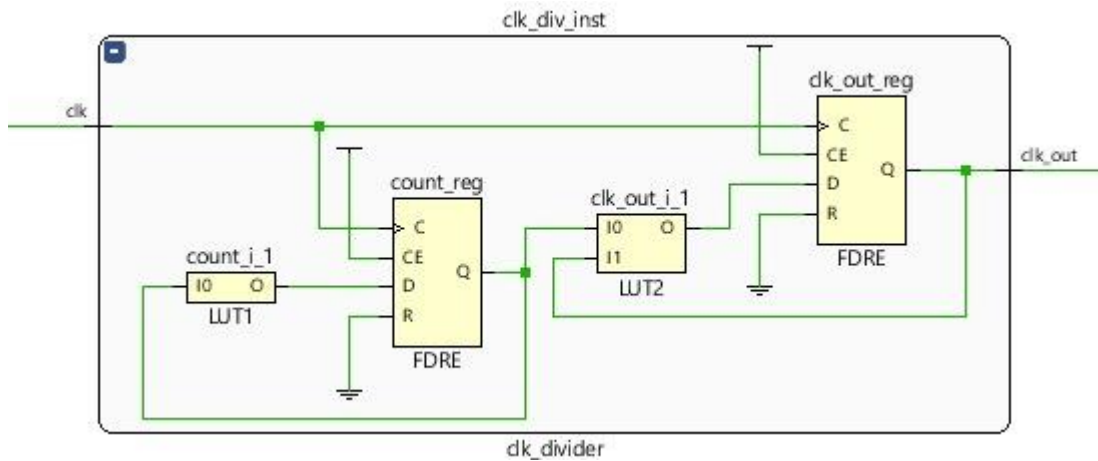


Figure 2 : Clock divider

A clock divider module that generates an output clock signal (`clk_out`) by dividing the input clock signal (`clk`) frequency by a parameterized value (`div_value`).

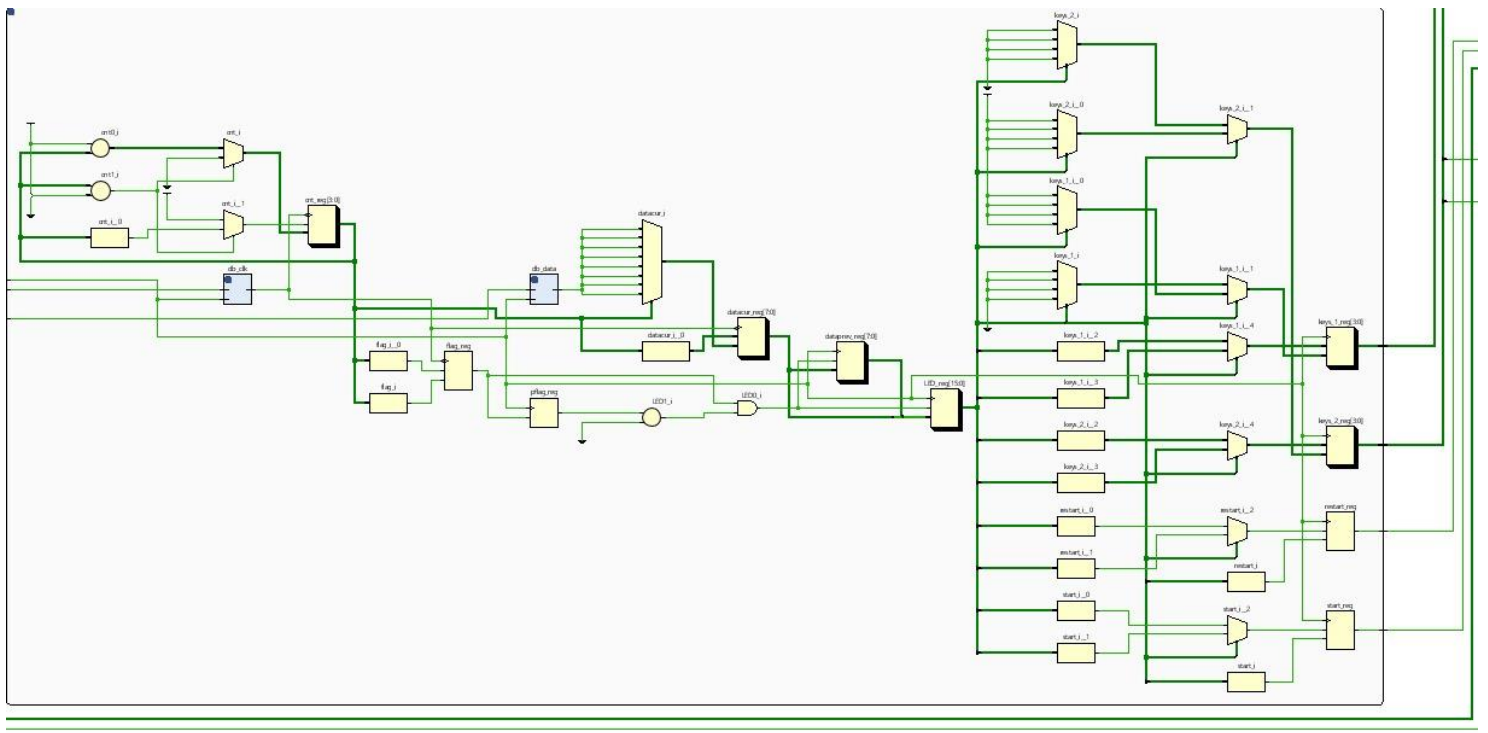


Figure 3: A PS/2 keyboard input module that decodes key presses and releases for two players' directional controls (Player 1: Arrow keys, Player 2: W, S, A, D), as well as Start (Enter) and Restart (Space) signals.



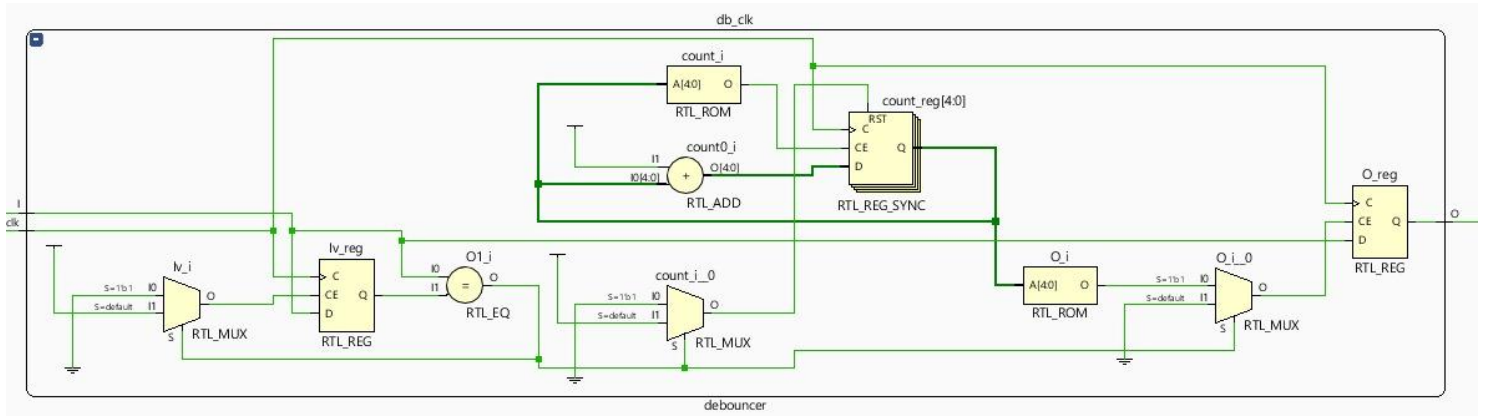


Figure 4: A debouncer module that stabilizes an input signal (I) by eliminating glitches using a counter-based delay, producing a clean output signal (O).

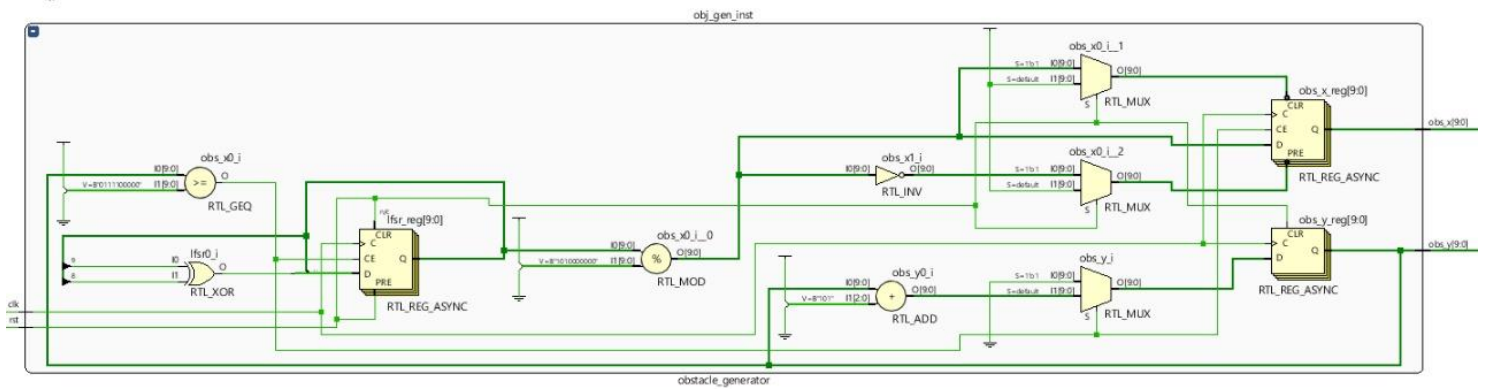


Figure 5: An obstacle generator module that uses an LFSR-based randomizer to generate horizontal positions (obs\_x) and vertically moves obstacles (obs\_y) down the screen, resetting upon reaching the bottom

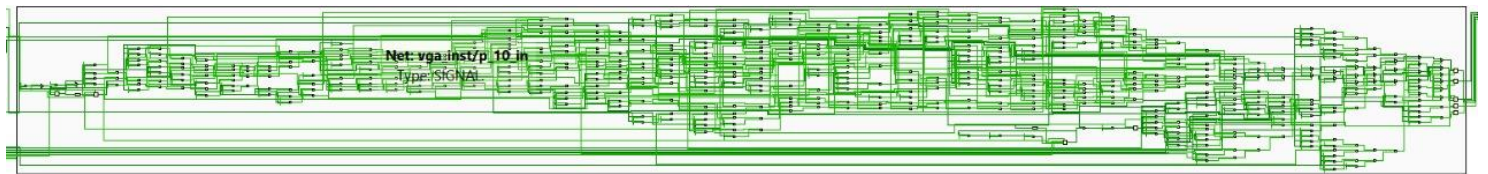


Figure 6: the vga\_controller module is an extensive implementation of a VGA display logic system, including specific designs for game states with visual text representation ("WELCOME," "PLAYER 1," "PLAYER 2," etc.) on the screen.

## 2.8 Input-Output-Control Block

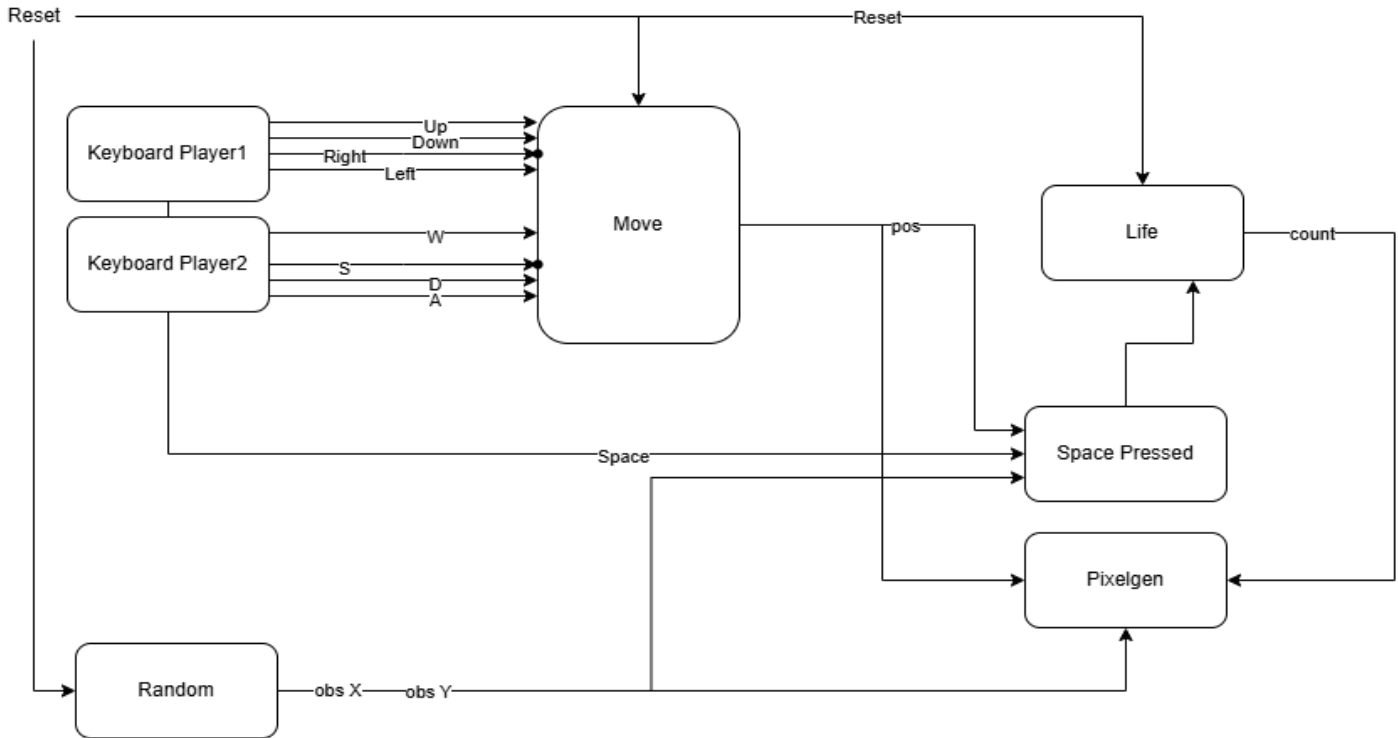


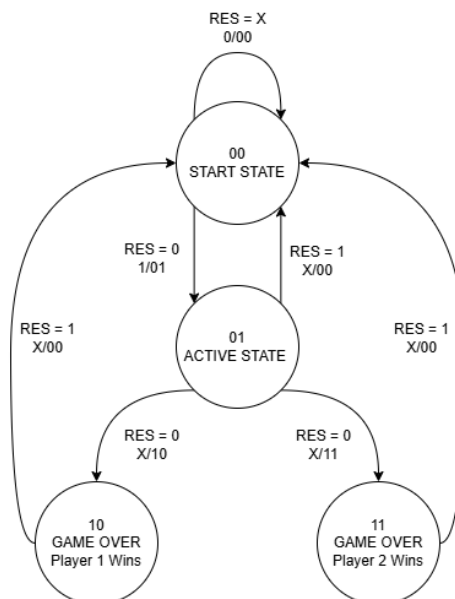
Figure 18

The Input-Output-Control block above shows how inputs transform to outputs in a racing game for two participants. Keyboard Player1 and Player2 to control movement through arrow keys and WASD displayed in Move module which shows the position of the players “pos”. The life Module counts lives (count) and alters them after a collision. For certain game events, the Space Pressed module is used for identifying space bar operations. The Pixelgen module generates visual outputs based on player positions and obstacle coordinates (obs X, obs Y) from the Random module. Combined, these modules help in the processing of game logic, visuals and interactions.

## • FSM Design Procedure

### 1. top\_game Module:

#### • State Assignment and Diagram





- **State Transition Table**

Present States		Input	RES	Next States	
A(t)	B(t)	X	Y	A(t+1)	B(t+1)
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	1
0	0	1	1	0	0
0	1	0	0	1	X
0	1	0	1	0	0
0	1	1	0	1	X
0	1	1	1	0	0
1	0	0	0	1	0
1	0	0	1	0	0
1	0	1	0	1	0
1	0	1	1	0	0
1	1	0	0	1	1
1	1	0	1	0	0
1	1	1	0	1	1
1	1	1	1	0	0

- **State Equations**

$$A(t+1) = \text{RES}' \cdot (A(t) \cdot B(t)' + X \cdot B(t))$$

$$B(t+1) = \text{RES}' \cdot (Y + A(t)' \cdot B(t) \cdot X)$$

(The vga\_controller module relies on the game\_state input, which is generated and controlled by the top\_game module)

- **Detailed Input Module Overview**

- Input Device: Keyboard

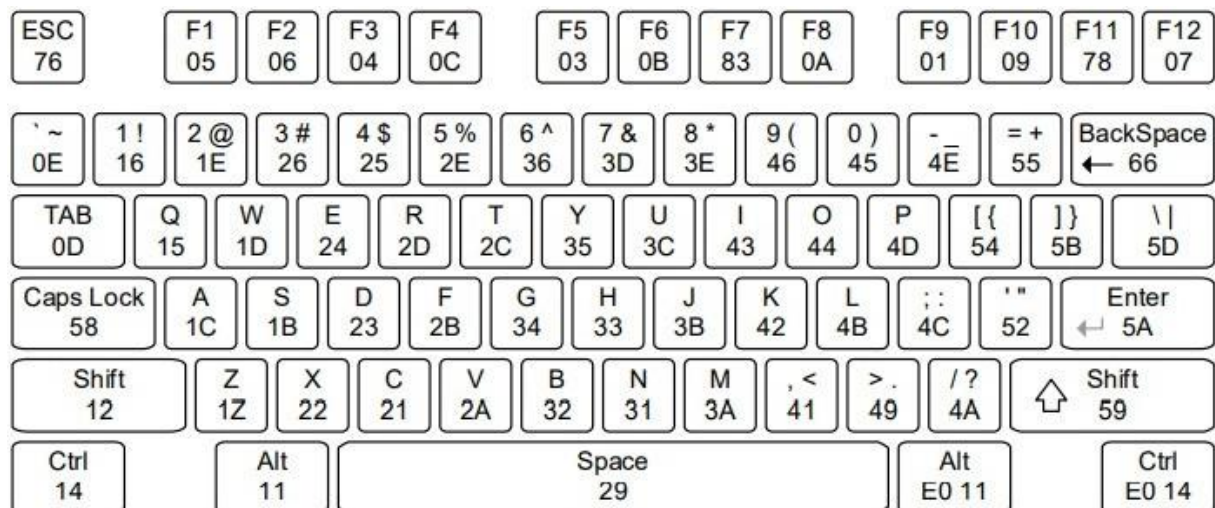


Figure 19

- Input Description:

PS2 Receiver module is used to shift-in keycodes from a keyboard plugged into the PS2 port 2 wires are used to transmit data, which are called PS2DATA and PS2CLOCK. When you press the button on the keyboard, the micro-controller inside the keyboard sends bits of information on DATA wire. Each bit is accompanied by a falling CLOCK edge signal, meaning that you must only read the data when CLOCK signal falls down and not the other time.

The first bit is START bit, then there are 8 bits for DATA0-7, then a PARITY bit and finally STOP bit Now looking broader, when a button is pressed, the keyboard sends one packet of 11 bits to tell which button was pressed. you only need to keep DATA0-7 bits and get rid of the rest. If the key is pressed and held then it starts repeatedly send the same packet. Finally if you release the button it sends another 2 packets, telling that the button was released and which button was released.

Whenever the user presses a key on a keyboard connected to the USB HID port (J2, labeled "USB"), a scan code is sent to the Basys3 through a PS/2 interface. This scan code is read and transmitted to the computer via the USB-UART bridge. When the key is released, a scan code of 0xF0XX is transmitted, indicating that the key with PS/2 code "XX" has been released.

## ▪ **Known Issues & Drawbacks**

- Lane Movement speed not aligned with speed of obstacles
- When we go back to main screen after resetting from active state, lives do not reset until game starts again

## ▪ **References**

[https://digilent.com/reference/\\_media/basys3:basys3\\_rm.pdf](https://digilent.com/reference/_media/basys3:basys3_rm.pdf)

## ▪ **GitHub Link**

Source code available at: <https://github.com/hamzaraza123/DLD-PROJECT---CAR-SMASH.git>

## • Configurations

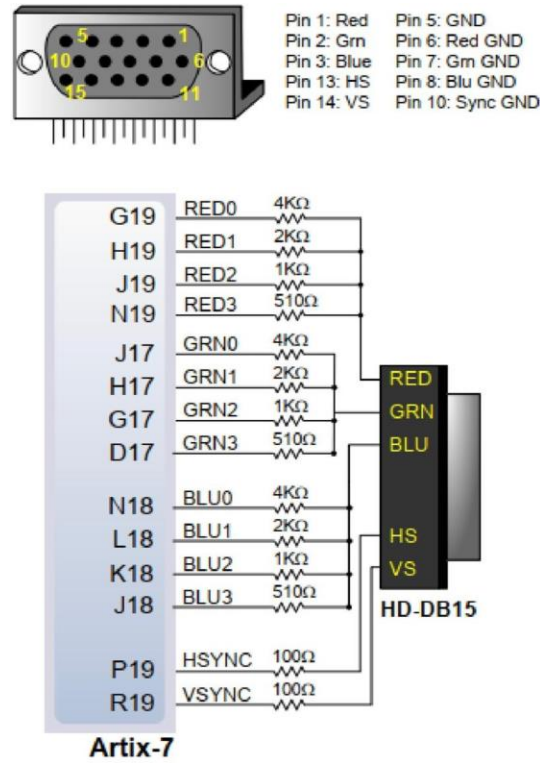


Figure 20

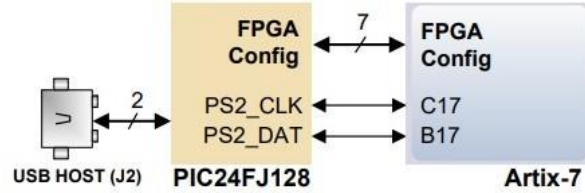


Figure 21

The connections between the FT2232HQ and the Artix-7 are shown in Fig. 6.

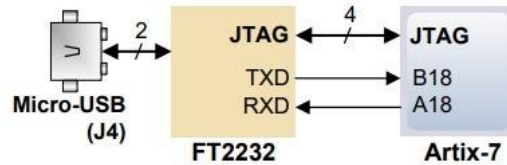


Figure 22

## ▪ Addendum (Main Module Codes for Reference):

### 1. Top Game

```
module top_game (input clk, input rst, input ps2_clk, input ps2_data, output [3:0] vga_r, vga_g, vga_b, output vga_hsync, vga_vsync, output led);
```

```
localparam STEP_SIZE = 3;
```

```
localparam SCREEN_WIDTH = 640;
```

```
localparam SCREEN_HEIGHT = 480;
```

```
wire clk25MHz;
```

```

wire slow_clk;
wire road_clk;
wire [3:0] keys_1, keys_2;
wire start, restart;
reg [9:0] player1_x, player1_y;
reg [9:0] player2_x, player2_y;
wire [9:0] obstacle_x, obstacle_y;
reg [1:0] game_state; // 0: Idle, 1: Playing, 2: Game Over
reg [1:0] player1_lives, player2_lives;
// Collision flags to ensure lives decrement only once per collision
reg player1_collision_flag;
reg player2_collision_flag;
wire player1_collision = (player1_x < obstacle_x + 0) && (player1_x + 40 > obstacle_x) && (player1_y < obstacle_y + 60) &&
    (player1_y + 60 > obstacle_y);
wire player2_collision = (player2_x < obstacle_x + 40) && (player2_x + 40 > obstacle_x) && (player2_y < obstacle_y + 60) &&
    (player2_y + 60 > obstacle_y);
clk_divider clk_div_inst (.clk(clk), .clk_out(clk25MHz));
reg [20:0] slow_clk_counter;
assign slow_clk = (slow_clk_counter == 0);
always @(posedge clk or posedge rst) begin
    if (rst)
        slow_clk_counter <= 0;
    else if (slow_clk_counter == 3333333)
        slow_clk_counter <= 0;
    else
        slow_clk_counter <= slow_clk_counter + 1;
end
reg [20:0] road_clk_counter;
parameter ROAD_CLK_DIV = 52_083; // Adjust for desired frequency
assign road_clk = (road_clk_counter == 0);
always @(posedge clk or posedge rst) begin
    if (rst)
        road_clk_counter <= 0;
    else if (road_clk_counter == ROAD_CLK_DIV)
        road_clk_counter <= 0;
end

```

```

else
    road_clk_counter <= road_clk_counter + 1;
end

keyboard_input kb_inst (.clk(clk), .kclk(ps2_clk), .kdata(ps2_data), .keys_1(keys_1), .keys_2(keys_2), .start(start), .restart(restart));

reg led_reg;

always @(posedge clk or posedge rst) begin
    if (rst)
        led_reg <= 0;
    else
        led_reg <= (keys_1 != 4'b0000 || keys_2 != 4'b0000);
end

assign led = led_reg;

always @(posedge clk or posedge rst) begin
    if (rst) begin
        game_state <= 0;
        player1_lives <= 2'd3;
        player2_lives <= 2'd3;
        player1_collision_flag <= 0;
        player2_collision_flag <= 0;
    end else begin
        case (game_state)
            0: if (start) begin
                game_state <= 1;
                player1_lives <= 2'd3;
                player2_lives <= 2'd3;
                player1_x <= 160;
                player1_y <= 400;
                player2_x <= 320;
                player2_y <= 400;
                player1_collision_flag <= 0;
                player2_collision_flag <= 0;
            end
            1: begin
                if (restart)

```

```

game_state <= 0;
else if (slow_clk) begin
    // Player 1 movement
    if (keys_1[0] && player1_y > 60)           // Prevent moving above top
        player1_y <= player1_y - STEP_SIZE;
    if (keys_1[1] && player1_y < SCREEN_HEIGHT - 60) // Prevent moving below bottom
        player1_y <= player1_y + STEP_SIZE;
    if (keys_1[2] && player1_x > 1)           // Prevent moving left of left edge
        player1_x <= player1_x - STEP_SIZE;
    if (keys_1[3] && player1_x < SCREEN_WIDTH - 40) // Prevent moving past right edge
        player1_x <= player1_x + STEP_SIZE;
    // Player 2 movement
    if (keys_2[0] && player2_y > 60)           // Prevent moving above top
        player2_y <= player2_y - STEP_SIZE;
    if (keys_2[1] && player2_y < SCREEN_HEIGHT - 60) // Prevent moving below bottom
        player2_y <= player2_y + STEP_SIZE;
    if (keys_2[2] && player2_x > 1)           // Prevent moving left of left edge
        player2_x <= player2_x - STEP_SIZE;
    if (keys_2[3] && player2_x < SCREEN_WIDTH - 40) // Prevent moving past right edge
        player2_x <= player2_x + STEP_SIZE;
    // Collision handling for Player 1
    if (player1_collision) begin
        if (!player1_collision_flag && player1_lives > 2'd0) begin
            player1_lives <= player1_lives - 2'd1;
            player1_collision_flag <= 1; // Set flag to avoid repeated decrement
        end
    end else begin
        player1_collision_flag <= 0; // Reset flag when no collision
    end
    // Collision handling for Player 2
    if (player2_collision) begin
        if (!player2_collision_flag && player2_lives > 2'd0) begin
            player2_lives <= player2_lives - 2'd1;
            player2_collision_flag <= 1; // Set flag to avoid repeated decrement
        end
    end
end

```

```

        end
    end else begin
        player2_collision_flag <= 0; // Reset flag when no collision
    end
    if(player2_lives==0 && player1_lives !=0)
        game_state<=3;
    else if(player1_lives==0 && player2_lives!=0)
        game_state<=2;
    end
end
2: begin
if(restart)
    game_state<=0;
    if(start)
        game_state<=1;
end

3: begin
if(restart)
    game_state<=0;
    if(start)
        game_state<=1;
end
endcase
end
end

obstacle_generator obj_gen_inst (.clk(slow_clk), .rst(rst), .obs_x(obstacle_x), .obs_y(obstacle_y));

vga_controller vga_inst (.clk25MHz(clk25MHz), .slow_clk(slow_clk), .sys_clk(clk), .road_clk(road_clk), .rst(rst),
    .game_state(game_state), .p1_x(player1_x), .p1_y(player1_y), .p2_x(player2_x), .p2_y(player2_y), .obs_x(obstacle_x),
    .obs_y(obstacle_y), .player1_lives(player1_lives), .player2_lives(player2_lives), .vga_r(vga_r), .vga_g(vga_g), .vga_b(vga_b),
    .vga_hsync(vga_hsync), .vga_vsync(vga_vsync));

endmodule

```

## 2. Clock Divider

```

module clk_divider (clk, clk_out);
    parameter div_value = 1;

```



```

input clk;

output clk_out;

reg clk_out; reg count;

initial

begin

clk_out = 0; count = 0;

end

always @(posedge clk)

begin

if (count == div_value)

count <= 0; // reset count

else

count <= count + 1; // count up

end

always @(posedge clk)

begin

if (count == div_value)

clk_out <= ~clk_out; //toggle

end

endmodule

```

### 3. Keyboard Input

```

module keyboard_input(input clk, input kclk, input kdata, output reg [3:0] keys_1 = 0, output reg [3:0] keys_2 = 0, output reg start = 0,
output reg restart = 0);

reg [7:0] datacur = 0;    // Current data byte
reg [7:0] dataprev = 0;  // Previous data byte
reg [15:0] LED = 0;      // Temporary storage for key states
reg [3:0] cnt = 0;       // State machine counter
reg flag = 0;            // Data ready flag
reg oflag = 0;           // Output flag for state change
reg pflag = 0;           // Previous flag state

// Debounced clock and data signals
wire kclkf, kdataf;

// Debouncer instances for clock and data

```

```

debouncer #(.COUNT_MAX(19), .COUNT_WIDTH(5)) db_clk (.clk(clk), .I(kclk), .O(kclkf));
debouncer #(.COUNT_MAX(19), .COUNT_WIDTH(5)) db_data (.clk(clk), .I(kdata), .O(kdataf));

// State machine for receiving PS/2 data
always @(negedge(kclkf)) begin
    case (cnt)
        0: ; // Start bit
        1: datacur[0] <= kdataf;
        2: datacur[1] <= kdataf;
        3: datacur[2] <= kdataf;
        4: datacur[3] <= kdataf;
        5: datacur[4] <= kdataf;
        6: datacur[5] <= kdataf;
        7: datacur[6] <= kdataf;
        8: datacur[7] <= kdataf;
        9: flag <= 1'b1; // Data byte ready
        10: flag <= 1'b0; // Reset flag
    endcase
    if (cnt <= 9)
        cnt <= cnt + 1;
    else if (cnt == 10)
        cnt <= 0;
end

// Process received data and update key states
always @(posedge clk) begin
    if (flag == 1'b1 && pflag == 1'b0) begin
        LED <= {dataprev, datacur}; // Store current and previous data
        oflag <= 1'b1;
        dataprev <= datacur;
    end else
        oflag <= 1'b0;
    pflag <= flag;
    // Handle key release (break code 0xF0)
    if (LED[15:8] == 8'hF0) begin
        // Player 1 key release

```

```

case (LED[7:0])
    8'h75: keys_1[0] <= 1'b0; // Arrow Up
    8'h72: keys_1[1] <= 1'b0; // Arrow Down
    8'h6B: keys_1[2] <= 1'b0; // Arrow Left
    8'h74: keys_1[3] <= 1'b0; // Arrow Right
    // Player 2 key release
    8'h1D: keys_2[0] <= 1'b0; // W
    8'h1B: keys_2[1] <= 1'b0; // S
    8'h1C: keys_2[2] <= 1'b0; // A
    8'h23: keys_2[3] <= 1'b0; // D
    // Start and Restart key release
    8'h5A: start <= 1'b0; // Enter
    8'h29: restart <= 1'b0; // Space
endcase
end else begin
    // Player 1 key press
    case (LED[7:0])
        8'h75: keys_1[0] <= 1'b1; // Arrow Up
        8'h72: keys_1[1] <= 1'b1; // Arrow Down
        8'h6B: keys_1[2] <= 1'b1; // Arrow Left
        8'h74: keys_1[3] <= 1'b1; // Arrow Right
    endcase
    // Player 2 key press
    case (LED[7:0])
        8'h1D: keys_2[0] <= 1'b1; // W
        8'h1B: keys_2[1] <= 1'b1; // S
        8'h1C: keys_2[2] <= 1'b1; // A
        8'h23: keys_2[3] <= 1'b1; // D
    endcase
    // Start and Restart key press
    case (LED[7:0])
        8'h5A: start <= 1'b1; // Enter
        8'h29: restart <= 1'b1; // Space
    endcase

```

```

    end
end
endmodule

```

#### 4. Debouncer

```

module debouncer(input clk, input I, output reg O);
    parameter COUNT_MAX=255, COUNT_WIDTH=8;
    reg [COUNT_WIDTH-1:0] count;
    reg Iv=0;
    always@(posedge clk)
        if (I == Iv) begin
            if (count == COUNT_MAX)
                O <= I;
            else
                count <= count + 1'b1;
        end else begin
            count <= 'b0;
            Iv <= I;
        end
endmodule

```

#### 5. obstacle\_generator

```

module obstacle_generator (input clk, rst, output reg [9:0] obs_x, obs_y);
    reg [9:0] lfsr = 10'b1010101010; // Initial LFSR value
    parameter V_PIXELS = 480;
    parameter H_PIXELS = 640;
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            obs_x <= lfsr % H_PIXELS;
            obs_y <= 0;
            lfsr <= 10'b1010101010; // Reset LFSR
        end else begin
            if (obs_y >= V_PIXELS) begin
                obs_y <= 0;
                lfsr <= {lfsr[8:0], lfsr[9] ^ lfsr[8]}; // Updated tap for 10-bit LFSR
                obs_x <= lfsr % H_PIXELS; // Random x-coordinate
            end
        end
    end
endmodule

```

```

    end else begin
        obs_y <= obs_y + 5; // Move down
    end
end
end
endmodule

```

## 6. VGA Controller

```

module vga_controller (input clk25MHz, input slow_clk, input sys_clk, input road_clk, input rst, input [9:0] p1_x, p1_y, input [9:0]
    p2_x, p2_y, input [9:0] obs_x, obs_y, input [1:0] game_state, input [1:0] player1_lives, input [1:0] player2_lives, output reg
    [3:0] vga_r, vga_g, vga_b, output reg vga_hsync, vga_vsync);

    // VGA parameters (640x480 resolution, 60Hz refresh rate)
    parameter H_PIXELS = 640;
    parameter V_PIXELS = 480;
    parameter H_FRONT_PORCH = 16;
    parameter H_SYNC_PULSE = 96;
    parameter H_BACK_PORCH = 48;
    parameter V_FRONT_PORCH = 10;
    parameter V_SYNC_PULSE = 2;
    parameter V_BACK_PORCH = 33;
    localparam SCREEN_WIDTH = 640;
    localparam SCREEN_HEIGHT = 480;

    // Counters for horizontal and vertical pixel positions
    reg [9:0] h_count = 0;
    reg [9:0] v_count = 0;

    // Road marking variables
    reg [9:0] road_offset = 0; // Vertical scrolling effect
    reg [9:0] road_marking_y;

    // Horizontal and vertical sync generation
    always @(posedge clk25MHz or posedge rst) begin
        if (rst) begin
            h_count <= 0;
            v_count <= 0;
        end else begin
            if (h_count == (H_PIXELS + H_FRONT_PORCH + H_SYNC_PULSE + H_BACK_PORCH - 1)) begin
                h_count <= 0;
            end
        end
    end
endmodule

```

```

        if (v_count == (V_PIXELS + V_FRONT_PORCH + V_SYNC_PULSE + V_BACK_PORCH - 1))
            v_count <= 0;
        else
            v_count <= v_count + 1;
        end else
            h_count <= h_count + 1;
        end
    end
end

// Sync pulse logic
always @(posedge clk25MHz) begin
    vga_hsync <= (h_count < (H_PIXELS + H_FRONT_PORCH)) ||
        (h_count >= (H_PIXELS + H_FRONT_PORCH + H_SYNC_PULSE));
    vga_vsync <= (v_count < (V_PIXELS + V_FRONT_PORCH)) ||
        (v_count >= (V_PIXELS + V_FRONT_PORCH + V_SYNC_PULSE));
end

// Road scrolling logic (smooth increment)
always @(posedge road_clk or posedge rst) begin
    if (rst)
        road_offset <= 0;
    else if (v_count == V_PIXELS - 1) // Update every frame
        road_offset <= (road_offset - 4 + V_PIXELS) % V_PIXELS; // Decrement to scroll down
    end

    always @(posedge clk25MHz) begin
        road_marking_y <= (v_count + road_offset) % V_PIXELS;
    end

    // Pixel color assignment
    always @(posedge clk25MHz) begin
        // Default background: grey
        vga_r <= 4'h8;
        vga_g <= 4'h8;
        vga_b <= 4'h8;

        if ((h_count < H_PIXELS) && (v_count < V_PIXELS)) begin
            // Road Markings (White stripes)
            if (game_state==0) begin

```

```

if (// W conditions
(h_count >= 50 && h_count <= 65 && v_count >= 30 && v_count <= 130) || // Left bar of W
(h_count >= 65 && h_count <= 85 && v_count >= 110 && v_count <= 130) || // bottom left bar of W
(h_count >= 85 && h_count <= 100 && v_count >= 60 && v_count <= 130) || // Middle bar of W
(h_count >= 100 && h_count <= 120 && v_count >= 110 && v_count <= 130) || // bottom right bar of W
(h_count >= 120 && h_count <= 135 && v_count >= 30 && v_count <= 130) || // right bar of W
// E conditions
(h_count >= 150 && h_count <= 165 && v_count >= 30 && v_count <= 130) || // Left vertical bar of E
(h_count >= 165 && h_count <= 195 && v_count >= 30 && v_count <= 45) || // Top bar of E
(h_count >= 165 && h_count <= 195 && v_count >= 72 && v_count <= 88) || // Middle bar of E
(h_count >= 165 && h_count <= 195 && v_count >= 115 && v_count <= 130) || // Bottom bar of E
// L conditions
(h_count >= 210 && h_count <= 225 && v_count >= 30 && v_count <= 130) || // Left vertical bar of L
(h_count >= 225 && h_count <= 260 && v_count >= 115 && v_count <= 130) || // Bottom bar of L
// C conditions
(h_count >= 275 && h_count <= 290 && v_count >= 30 && v_count <= 130) || // Left Bar of C
(h_count >= 290 && h_count <= 325 && v_count >= 30 && v_count <= 45) || // Top Bar of C
(h_count >= 290 && h_count <= 325 && v_count >= 115 && v_count <= 130) || // Bottom Bar of C
// O conditions
(h_count >= 340 && h_count <= 355 && v_count >= 30 && v_count <= 130) || // Left Bar of O
(h_count >= 355 && h_count <= 390 && v_count >= 30 && v_count <= 45) || // Top Bar of O
(h_count >= 355 && h_count <= 390 && v_count >= 115 && v_count <= 130) || // Bottom Bar of O
(h_count >= 390 && h_count <= 405 && v_count >= 30 && v_count <= 130) || // Right Bar of O
// M conditions
(h_count >= 420 && h_count <= 435 && v_count >= 30 && v_count <= 130) || // Left Bar of M
(h_count >= 435 && h_count <= 455 && v_count >= 30 && v_count <= 45) || // Top Left Bar of M
(h_count >= 455 && h_count <= 470 && v_count >= 30 && v_count <= 100) || // Middle Bar of M
(h_count >= 470 && h_count <= 490 && v_count >= 30 && v_count <= 45) || // Top Right Bar of M
(h_count >= 490 && h_count <= 505 && v_count >= 30 && v_count <= 130) || // Right Bar of M
// E conditions (again for second E)
(h_count >= 520 && h_count <= 535 && v_count >= 30 && v_count <= 130) || // Left vertical bar of E
(h_count >= 535 && h_count <= 560 && v_count >= 30 && v_count <= 45) || // Top bar of E
(h_count >= 535 && h_count <= 560 && v_count >= 72 && v_count <= 88) || // Middle bar of E
(h_count >= 535 && h_count <= 560 && v_count >= 115 && v_count <= 130) // Bottom bar of E

```



```

) begin

    // Text color (light pink)

    vga_r <= 4'hF; // High vga_r intensity
    vga_g <= 4'h0; // High vga_g intensity
    vga_b <= 4'h0; // High vga_b intensity (light pink)
end else begin

    // Background color (black)

    vga_r <= 4'h0; // No vga_r intensity
    vga_g <= 4'h0; // No vga_g intensity
    vga_b <= 4'h0; // No vga_b intensity
end

end else if(game_state==2) begin

    if ( // P conditions

        (h_count >= 50 && h_count <= 65 && v_count >= 30 && v_count <= 130) || // Left bar of P
        (h_count >= 65 && h_count <= 95 && v_count >= 30 && v_count <= 45) || // Top bar of P
        (h_count >= 65 && h_count <= 95 && v_count >= 75 && v_count <= 90) || // Middle bar of P
        (h_count >= 95 && h_count <= 110 && v_count >= 30 && v_count <= 90) || // Right bar of P

        // L conditions

        (h_count >= 125 && h_count <= 140 && v_count >= 30 && v_count <= 130) || // Left vertical bar of L
        (h_count >= 140 && h_count <= 170 && v_count >= 115 && v_count <= 130) || // Bottom bar of L

        // A Conditions

        (h_count >= 185 && h_count <= 200 && v_count >= 30 && v_count <= 130) || // Left Bar of A
        (h_count >= 200 && h_count <= 230 && v_count >= 30 && v_count <= 45) || // Top Bar of A
        (h_count >= 200 && h_count <= 230 && v_count >= 75 && v_count <= 90) || // Middle Bar of A
        (h_count >= 230 && h_count <= 245 && v_count >= 30 && v_count <= 130) || // Right Bar of A

        // Y Conditions

        (h_count >= 260 && h_count <= 275 && v_count >= 30 && v_count <= 60) || // Left Bar of Y
        (h_count >= 275 && h_count <= 305 && v_count >= 60 && v_count <= 75) || // Middle Horizontal Bar of Y
        (h_count >= 305 && h_count <= 320 && v_count >= 30 && v_count <= 60) || // Right Bar of A
        (h_count >= 285 && h_count <= 295 && v_count >= 75 && v_count <= 130) || // Middle Vertical Bar of A

        // E conditions

        (h_count >= 335 && h_count <= 350 && v_count >= 30 && v_count <= 130) || // Left vertical bar of E
        (h_count >= 350 && h_count <= 385 && v_count >= 30 && v_count <= 45) || // Top bar of E
        (h_count >= 350 && h_count <= 385 && v_count >= 72 && v_count <= 88) || // Middle bar of E

```

```

(h_count >= 350 && h_count <= 385 && v_count >= 115 && v_count <= 130) || // Bottom bar of E

// R Condition
(h_count >= 400 && h_count <= 415 && v_count >= 30 && v_count <= 130) || // Left Bar of R
(h_count >= 415 && h_count <= 445 && v_count >= 30 && v_count <= 45) || // Top Bar of R
(h_count >= 430 && h_count <= 445 && v_count >= 72 && v_count <= 88) || // Middle Bar of R
(h_count >= 445 && h_count <= 460 && v_count >= 30 && v_count <= 130) || // Right Bar of R

// Conditions for drawing the number 2
(h_count >= 560 && h_count <= 600 && v_count >= 30 && v_count <= 45) || // Top horizontal bar of 2
(h_count >= 600 && h_count <= 610 && v_count >= 45 && v_count <= 80) || // Top-right vertical bar of 2
(h_count >= 560 && h_count <= 600 && v_count >= 80 && v_count <= 95) || // Middle horizontal bar of 2
(h_count >= 560 && h_count <= 570 && v_count >= 95 && v_count <= 130) || // Bottom-left vertical bar of 2
(h_count >= 560 && h_count <= 600 && v_count >= 115 && v_count <= 130) || // Bottom horizontal bar of 2

// W conditions
(h_count >= 260 && h_count <= 275 && v_count >= 150 && v_count <= 250) || // Left bar of W
(h_count >= 275 && h_count <= 290 && v_count >= 235 && v_count <= 250) || // bottom left bar of W
(h_count >= 290 && h_count <= 305 && v_count >= 150 && v_count <= 250) || // Middle bar of W
(h_count >= 305 && h_count <= 320 && v_count >= 235 && v_count <= 250) || // bottom right bar of W
(h_count >= 320 && h_count <= 335 && v_count >= 150 && v_count <= 250) || // right bar of W

// O conditions
(h_count >= 350 && h_count <= 365 && v_count >= 150 && v_count <= 250) || // Left Bar of O
(h_count >= 365 && h_count <= 385 && v_count >= 150 && v_count <= 165) || // Top Bar of O
(h_count >= 365 && h_count <= 385 && v_count >= 235 && v_count <= 250) || // Bottom Bar of O
(h_count >= 385 && h_count <= 400 && v_count >= 150 && v_count <= 250) || // Right Bar of O

// N Conditions
(h_count >= 410 && h_count <= 425 && v_count >= 150 && v_count <= 250) || // Left Bar of N
(h_count >= 425 && h_count <= 440 && v_count >= 150 && v_count <= 165) || // Top-left Bar of N
(h_count >= 440 && h_count <= 455 && v_count >= 150 && v_count <= 250) || // Middle line-Bar of N
(h_count >= 455 && h_count <= 470 && v_count >= 235 && v_count <= 250) || // Bottom-right Bar of N
(h_count >= 470 && h_count <= 485 && v_count >= 150 && v_count <= 250) // Right Bar of N

)

begin

// Text color (light pink)

vga_r <= 4'hF; // High vga_r intensity

vga_g <= 4'h0; // High vga_g intensity

```

```

    vga_b <= 4'h0; // High vga_b intensity (light pink)
end else begin

    // Background color (black)

    vga_r <= 4'h0; // No vga_r intensity

    vga_g <= 4'h0; // No vga_g intensity

    vga_b <= 4'h0; // No vga_b intensity

end

end

else if(game_state==3) begin

    if (

        // P conditions

        (h_count >= 50 && h_count <= 65 && v_count >= 30 && v_count <= 130) || // Left bar of P

        (h_count >= 65 && h_count <= 95 && v_count >= 30 && v_count <= 45) || // Top bar of P

        (h_count >= 65 && h_count <= 95 && v_count >= 75 && v_count <= 90) || // Middle bar of P

        (h_count >= 95 && h_count <= 110 && v_count >= 30 && v_count <= 90) || // Right bar of P

        // L conditions

        (h_count >= 125 && h_count <= 140 && v_count >= 30 && v_count <= 130) || // Left vertical bar of L

        (h_count >= 140 && h_count <= 170 && v_count >= 115 && v_count <= 130) || // Bottom bar of L

        // A Conditions

        (h_count >= 185 && h_count <= 200 && v_count >= 30 && v_count <= 130) || // Left Bar of A

        (h_count >= 200 && h_count <= 230 && v_count >= 30 && v_count <= 45) || // Top Bar of A

        (h_count >= 200 && h_count <= 230 && v_count >= 75 && v_count <= 90) || // Middle Bar of A

        (h_count >= 230 && h_count <= 245 && v_count >= 30 && v_count <= 130) || // Right Bar of A

        // Y Conditions

        (h_count >= 260 && h_count <= 275 && v_count >= 30 && v_count <= 60) || // Left Bar of Y

        (h_count >= 275 && h_count <= 305 && v_count >= 60 && v_count <= 75) || // Middle Horizontal Bar of Y

        (h_count >= 305 && h_count <= 320 && v_count >= 30 && v_count <= 60) || // Right Bar of A

        (h_count >= 285 && h_count <= 295 && v_count >= 75 && v_count <= 130) || // Middle Vertical Bar of A

        // E conditions

        (h_count >= 335 && h_count <= 350 && v_count >= 30 && v_count <= 130) || // Left vertical bar of E

        (h_count >= 350 && h_count <= 385 && v_count >= 30 && v_count <= 45) || // Top bar of E

        (h_count >= 350 && h_count <= 385 && v_count >= 72 && v_count <= 88) || // Middle bar of E

        (h_count >= 350 && h_count <= 385 && v_count >= 115 && v_count <= 130) || // Bottom bar of E

        // R Condition

```

```

(h_count >= 400 && h_count <= 415 && v_count >= 30 && v_count <= 130) || // Left Bar of R
(h_count >= 415 && h_count <= 445 && v_count >= 30 && v_count <= 45) || // Top Bar of R
(h_count >= 430 && h_count <= 445 && v_count >= 72 && v_count <= 88) || // Middle Bar of R
(h_count >= 445 && h_count <= 460 && v_count >= 30 && v_count <= 130) || // Right Bar of R

// I Conditions
(h_count >= 510 && h_count <= 550 && v_count >= 115 && v_count <= 130) || // Bottom Bar of I
(h_count >= 525 && h_count <= 535 && v_count >= 30 && v_count <= 115) || // Middle Bar of I
(h_count >= 520 && h_count <= 525 && v_count >= 30 && v_count <= 45) || // Top small Bar of I

// W conditions
(h_count >= 260 && h_count <= 275 && v_count >= 150 && v_count <= 250) || // Left bar of W
(h_count >= 275 && h_count <= 290 && v_count >= 235 && v_count <= 250) || // bottom left bar of W
(h_count >= 290 && h_count <= 305 && v_count >= 150 && v_count <= 250) || // Middle bar of W
(h_count >= 305 && h_count <= 320 && v_count >= 235 && v_count <= 250) || // bottom right bar of W
(h_count >= 320 && h_count <= 335 && v_count >= 150 && v_count <= 250) || // right bar of W

// O conditions
(h_count >= 350 && h_count <= 365 && v_count >= 150 && v_count <= 250) || // Left Bar of O
(h_count >= 365 && h_count <= 385 && v_count >= 150 && v_count <= 165) || // Top Bar of O
(h_count >= 365 && h_count <= 385 && v_count >= 235 && v_count <= 250) || // Bottom Bar of O
(h_count >= 385 && h_count <= 400 && v_count >= 150 && v_count <= 250) || // Right Bar of O

// N Conditions
(h_count >= 410 && h_count <= 425 && v_count >= 150 && v_count <= 250) || // Left Bar of N
(h_count >= 425 && h_count <= 440 && v_count >= 150 && v_count <= 165) || // Top-left Bar of N
(h_count >= 440 && h_count <= 455 && v_count >= 150 && v_count <= 250) || // Middle line-Bar of N
(h_count >= 455 && h_count <= 470 && v_count >= 235 && v_count <= 250) || // Bottom-right Bar of N
(h_count >= 470 && h_count <= 485 && v_count >= 150 && v_count <= 250) // Right Bar of N
)

begin
    // Text color (light pink)
    vga_r <= 4'hF; // High vga_r intensity
    vga_g <= 4'h0; // High vga_g intensity
    vga_b <= 4'h0; // High vga_b intensity (light pink)
end else begin
    // Background color (black)
    vga_r <= 4'h0; // No vga_r intensity

```

```

    vga_g <= 4'h0; // No vga_g intensity
    vga_b <= 4'h0; // No vga_b intensity
end
end

if (((road_marking_y >= 20 && road_marking_y < 80) || (road_marking_y >= 110 && road_marking_y < 170) ||
    (road_marking_y >= 200 && road_marking_y < 260) || (road_marking_y >= 290 && road_marking_y < 350)) &&
    ((h_count >= 210 && h_count < 220) || (h_count >= 430 && h_count < 440))) begin
    vga_r <= 4'hF;
    vga_g <= 4'hF;
    vga_b <= 4'hF;
end

// Player 1 Car (Blue)
if ((h_count >= p1_x) && (h_count < (p1_x + 40)) &&
    (v_count >= p1_y) && (v_count < (p1_y + 60)) &&
    (p1_x < SCREEN_WIDTH) && (p1_y < SCREEN_HEIGHT)) begin
    vga_r <= 4'h0; // Blue
    vga_g <= 4'h0;
    vga_b <= 4'hF;
end

// Player 2 Car (Purple)
if ((h_count >= p2_x) && (h_count < (p2_x + 40)) &&
    (v_count >= p2_y) && (v_count < (p2_y + 60)) &&
    (p2_x < SCREEN_WIDTH) && (p2_y < SCREEN_HEIGHT)) begin
    vga_r <= 4'hF; // Purple
    vga_g <= 4'h0;
    vga_b <= 4'hF;
end

// Obstacle (Dark Red)
if ((h_count >= obs_x) && (h_count < (obs_x + 40)) &&
    (v_count >= obs_y) && (v_count < (obs_y + 60))) begin
    vga_r <= 4'hF;
    vga_g <= 4'h0;
    vga_b <= 4'h0;
end
end

```

```

// Player 1 Lives: Red rectangles at the top-left
if ((h_count >= 10 && h_count < 20 && player1_lives >= 1) ||
    (h_count >= 30 && h_count < 40 && player1_lives >= 2) ||
    (h_count >= 50 && h_count < 60 && player1_lives == 3)) begin
    if (v_count >= 10 && v_count < 20) begin
        vga_r <= 4'h0; // Red
        vga_g <= 4'h0;
        vga_b <= 4'hF;
    end
end

// Player 2 Lives: Blue rectangles at the top-right
if ((h_count >= 580 && h_count < 590 && player2_lives == 3) ||
    (h_count >= 600 && h_count < 610 && player2_lives >= 2) ||
    (h_count >= 620 && h_count < 630 && player2_lives >= 1)) begin
    if (v_count >= 10 && v_count < 20) begin
        vga_r <= 4'hF; // Blue
        vga_g <= 4'h0;
        vga_b <= 4'hF;
    end
end

end else begin
    // Outside active display area
    vga_r <= 4'h0;
    vga_g <= 4'h0;
    vga_b <= 4'h0;
end

end

endmodule

```