Habib University

Dhanani School of Science and Engineering

Digital Logic and Design

EE/CS 172/130

T2

Final Project Report

FlapGA Dodging Game

Jotesh Kumar, Khawar Mehmood Awan, Taha Hunaid, Moaz Siddiqui

Professor Farhan Khan

December 11th 2023

Table of Contents:

1. Introduction:

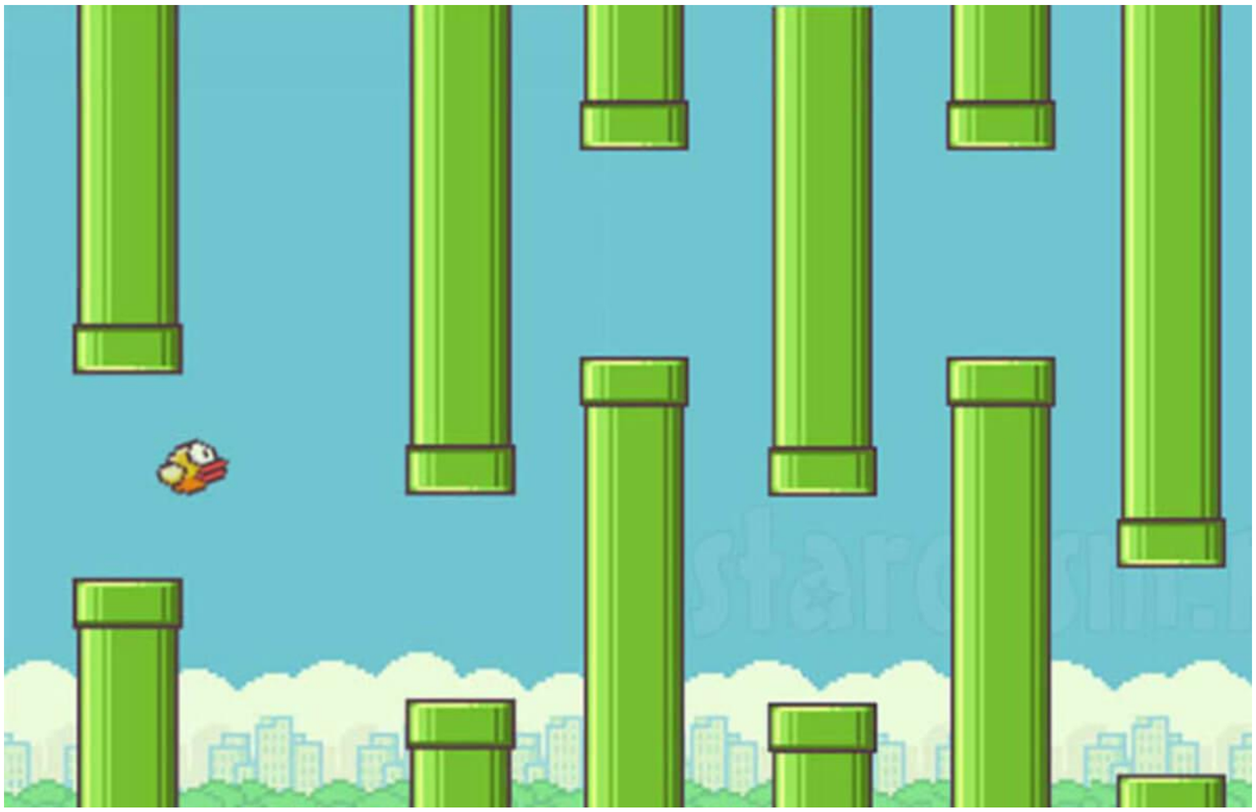
We aim to recreate the popular mobile game "Flappy Bird", called "FLAPGA DODGING GAME” In this rendition, players control a block as they navigate through a series of obstacles and challenges in a side-scrolling environment.

1.1 Game Objective and Logic:

Players guide the hero block through a horizontally scrolling world, avoiding obstacles and collecting coins. The objective is to travel and challenge yourselves to reach 30 points without colliding with obstacles.

In this game, the hero will only go up or down, so the logic is quite simple. To create the effect of gravity, several counters are used. When we press the up button, we give the hero a up-going velocity by setting the movement counter to a small value and the hero’s status to jumping up. When the counter decreases to zero, we move the hero up one pixel and assign a greater value to the movement counter until we have reached the maximum counter value. After the hero reaches its peak, we set the status to jumping down and assign a big value to the movement counter. Similarly, when the counter reaches zero, we move the hero down one pixel and assign a smaller value to the counter, until we reached the minimum value. Then the hero will keep falling down at the same speed until the up button is pressed again.

The game is lost if the hero hits one of the pipes. The game engine also needs to store how much points the player has got, whether the game has been already over. We use another module to display score on the FPGA board.



1.2 Hardware Implementation:

- We will utilize an FPGA board, specifically [Insert FPGA Model], for hardware development.
- Input will be taken through a custom-designed keypad or joystick interface.
- The output will be displayed on a VGA monitor.

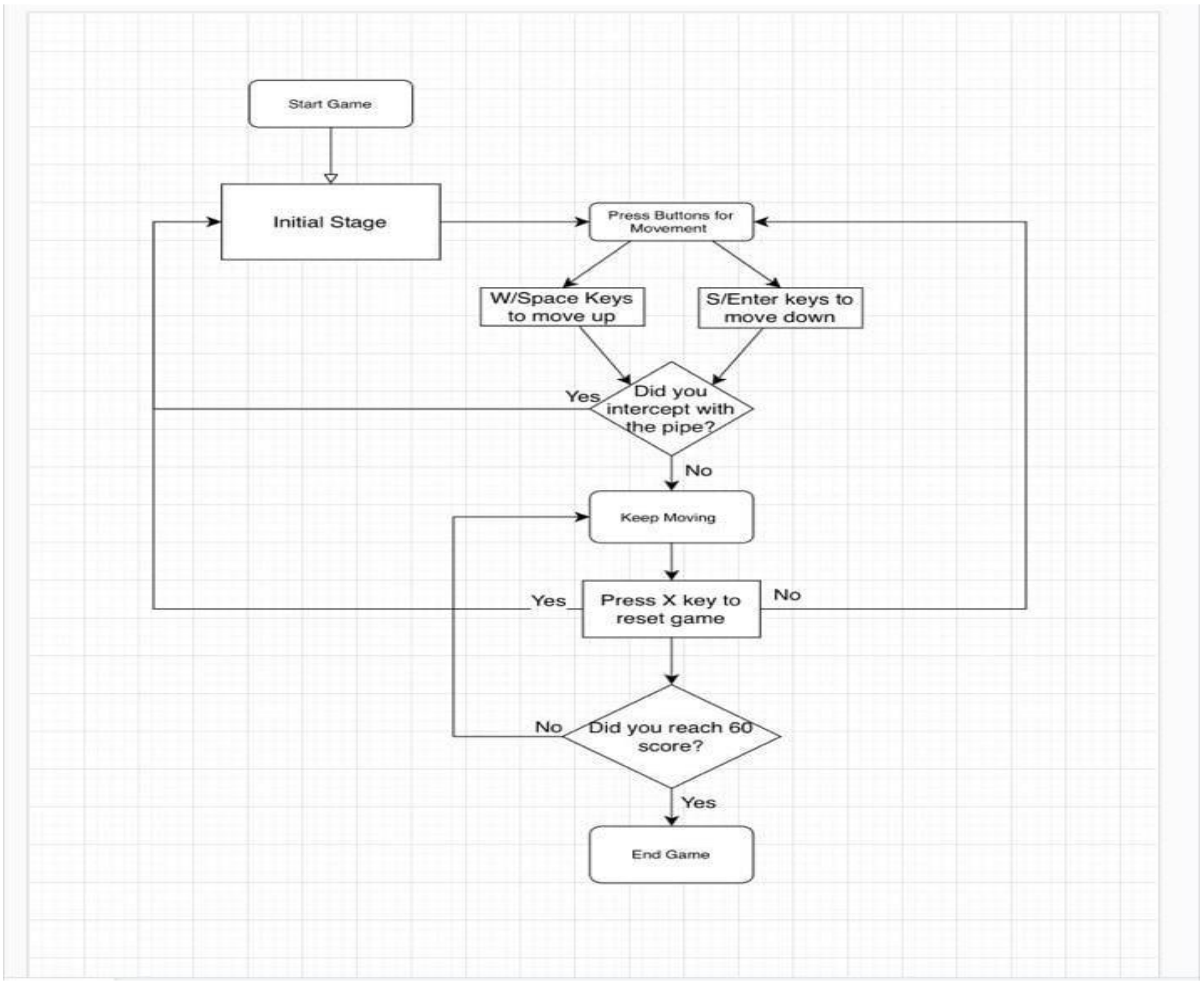
1.3 Software Implementations:

- FPGA Development Software
- Hardware Description Languages (HDLs)
- Simulation Environments
- Version Control System
- Integrated Development Environments (IDEs)

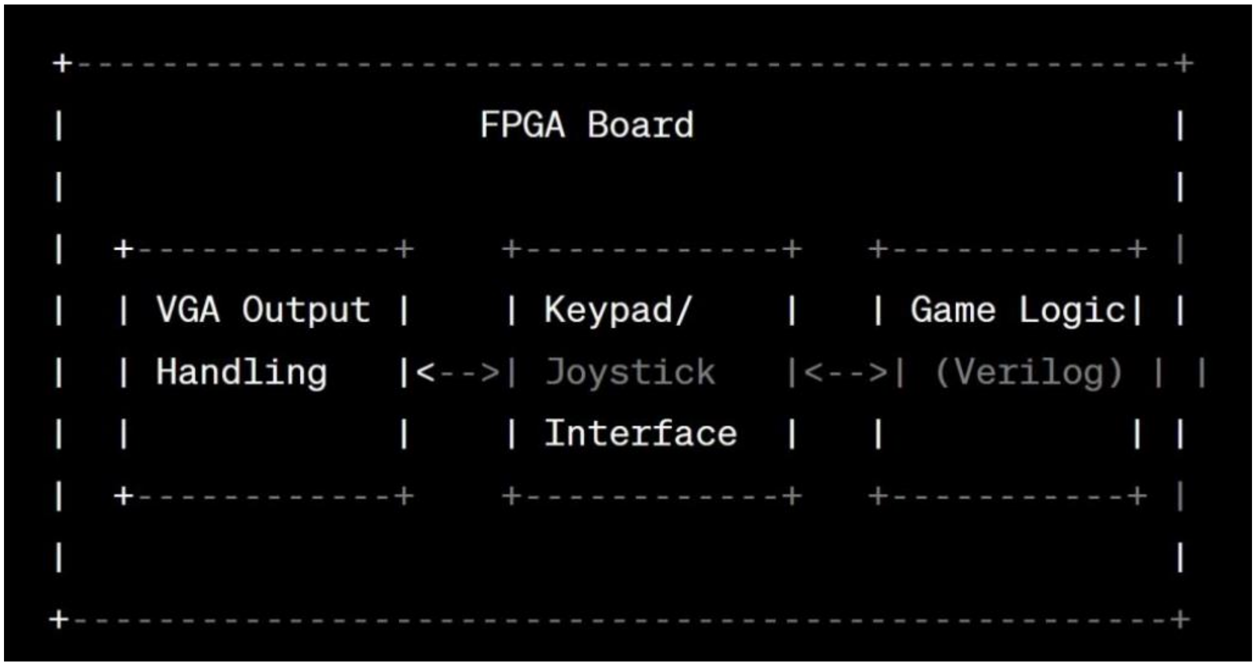
2. User Flow Diagram:

The following is the interim User Flow Diagram of our project:

The user-flow diagram is elegantly made indicating all the possibilities for a user in this game. The user is controlling any of the W or Space keys to move up and S or Enter keys to go down. Extremely essential for the success of the user to properly navigate the block and not hit the tubes while controlling the block. The user can either move up by pressing space bar or W key or descend downwards by pressing S key or Enter key. The tubes will keep randomly generating until the game is over and the user crashes the block with a tube, or when the score crosses 30, or X key is pressed, and the game resets.


Start Game
Initial Stage
Press Buttons for Movement
W/Space Keys to move up
S/Enter keys to move down
Yes
Did you intercept with the pipe?
No
Keep Moving
Yes
Press X key to reset game
No
No
Did you reach 60 score?
Yes
End Game

3. Block Diagram:



```
+------------------------------------------+
|                FPGA Board                |
|                                          |
|  +-----------+   +-----------+   +----------+ |
|  | VGA Output |   | Keypad/    |   | Game Logic| |
|  | Handling   |<-->| Joystick   |<-->| (Verilog) | |
|  |            |   | Interface  |   |          | |
|  +-----------+   +-----------+   +----------+ |
|                                          |
+------------------------------------------+
```

In this block diagram:

- "FPGA Board" represents the hardware platform used for implementing the game.
- "VGA Output Handling" is responsible for generating video signals to display the game on a VGA monitor.
- "Keypad/Joystick Interface" captures user input and controls character movements within the game.
- "Game Logic (Verilog)" encompasses the core logic and rules of the game, including character movement, obstacle behavior, scoring, and power-up implementation.

The components interact as follows:

- User input from the keypad or joystick interface is processed by the game logic module.
- The game logic module controls the movement of the main frog character, interaction with obstacles, and scoring.
- The results of the game logic, including character positions and scores, are sent to the VGA output handling module to update the game's display on the VGA monitor.

4. Division of Task:

The task division between the three blocks of our project is as followed:

1. The tasks of the input block of our project is when we press any of the keys to provide the control block with an input/task for it to perform.

2. The task of the control block of our project is to check the entry of the input and make sure the output is being performed. In simple words, the hero moving upward and downward is a control block but since it also a function of the input we are providing, it can also be understood to be an output block.

3. The task of the output block of our project is that when the space bar is pressed, the flappy bird will ascend (move up) for a unit time and then descend (move down) when descend keys are pressed. If the hero collides with any of the walls of the tubes or touches the upper or lower frame of the screen the game finishes and goes back to the initial stage.

5. Implementation

5.1. Input Block:

5.1.1 Introduction to the input peripheral: The Keyboard:

The Artix 7 FPGA board is powered by connecting it to the pc via micro USB. After turning the switch on we are required to configure the board. Bit streams generated (in .bit files) for the configuration using Vivado software. We program them using the hardware language, Verilog.

In order to move our bird, we press 'spacebar'. Pressing the space bar moves the bird upwards for an instance after which it continues its downwards trajectory. In order to exit/reset the game, we press 'x'. Pressing the reset button (x) resets the game to its idle state regardless of the game progress.

5.1.2 Input translation to Output:

The bird moves up when the space bar is pressed. The clock of the keyboard sends the data on the rising edges. An important thing to note here is that the space bar's functionality will be designed such that at every press the bird jumps up rather than just escalating up with a constant direction and angle.

5.2 Input Code:

5.2.1 Module: Keyboard

This module is designed to handle PS2 keyboard input and translate it into a codeword. The codeword is then used to determine the game state.

Inputs:

CLK: Clock signal.

PS2_CLK, PS2_DATA: Signals from the PS2 keyboard.

reset: Reset signal.

Outputs:

state: Represents the game state based on keyboard input.

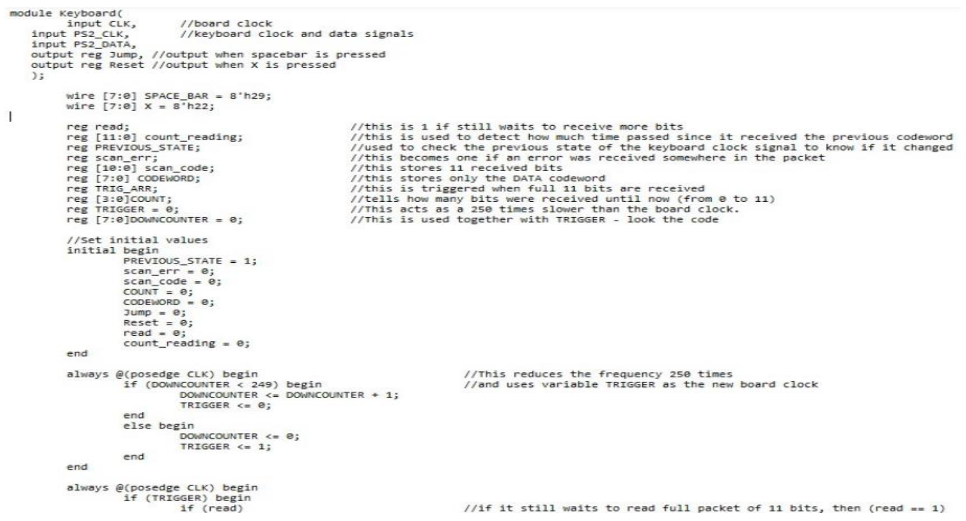
reset: Signal to reset the game.

Functionality:

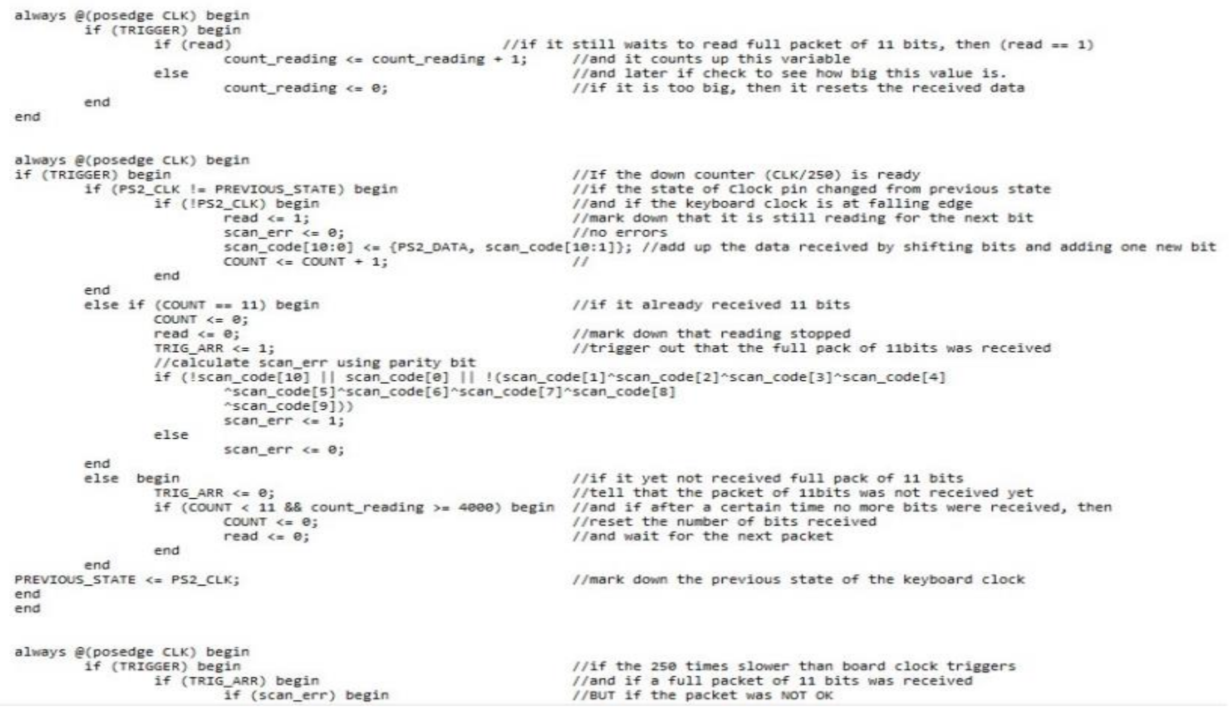
The module detects keyboard inputs and translates them into codewords.

It keeps track of the state of the keyboard and generates appropriate signals based on key presses (e.g., SPACE_BAR, X, RELEASED, W, S).

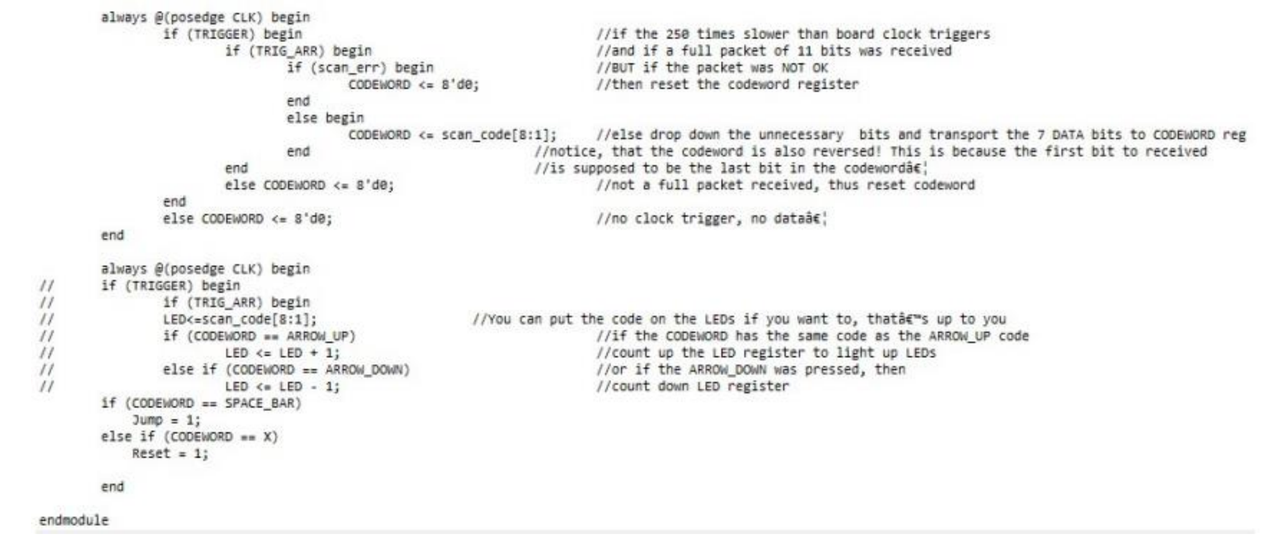
The state and reset signals are then used in the game logic.



```verilog
module Keyboard(
        input CLK,      //board clock
   input PS2_CLK,       //keyboard clock and data signals
   input PS2_DATA,
   output reg Jump, //output when spacebar is pressed
   output reg Reset //output when X is pressed
   );

        wire [7:0] SPACE_BAR = 8'h29;
        wire [7:0] X = 8'h22;

        reg read;                               //this is 1 if still waits to receive more bits
        reg [11:0] count_reading;               //this is used to detect how much time passed since it received the previous codeword
        reg PREVIOUS_STATE;                     //used to check the previous state of the keyboard clock signal to know if it changed
        reg scan_err;                           //this becomes one if an error was received somewhere in the packet
        reg [10:0] scan_code;                   //this stores 11 received bits
        reg [7:0] CODEWORD;                     //this stores only the DATA codeword
        reg TRIG_ARR;                           //this is triggered when full 11 bits are received
        reg [3:0]COUNT;                         //tells how many bits were received until now (from 0 to 11)
        reg TRIGGER = 0;                        //This acts as a 250 times slower than the board clock.
        reg [7:0]DOWNCOUNTER = 0;               //This is used together with TRIGGER - look the code

        //Set initial values
        initial begin
                PREVIOUS_STATE = 1;
                scan_err = 0;
                scan_code = 0;
                COUNT = 0;
                CODEWORD = 0;
                Jump = 0;
                Reset = 0;
                read = 0;
                count_reading = 0;
        end

        always @(posedge CLK) begin                             //This reduces the frequency 250 times
                if (DOWNCOUNTER < 249) begin                    //and uses variable TRIGGER as the new board clock
                        DOWNCOUNTER <= DOWNCOUNTER + 1;
                        TRIGGER <= 0;
                end
                else begin
                        DOWNCOUNTER <= 0;
                        TRIGGER <= 1;
                end
        end

        always @(posedge CLK) begin
                if (TRIGGER) begin
                        if (read)                               //if it still waits to read full packet of 11 bits, then (read == 1)
```



```
always @(posedge CLK) begin
        if (TRIGGER) begin
                if (read)                                        //if it still waits to read full packet of 11 bits, then (read == 1)
                        count_reading <= count_reading + 1;      //and it counts up this variable
                else                                             //and later if check to see how big this value is.
                        count_reading <= 0;                      //if it is too big, then it resets the received data
        end
end

always @(posedge CLK) begin
if (TRIGGER) begin                                                //If the down counter (CLK/250) is ready
        if (PS2_CLK != PREVIOUS_STATE) begin                      //if the state of Clock pin changed from previous state
                if (!PS2_CLK) begin                               //and if the keyboard clock is at falling edge
                        read <= 1;                                //mark down that it is still reading for the next bit
                        scan_err <= 0;                            //no errors
                        scan_code[10:0] <= {PS2_DATA, scan_code[10:1]}; //add up the data received by shifting bits and adding one new bit
                        COUNT <= COUNT + 1;                       //
                end
        end
        else if (COUNT == 11) begin                               //if it already received 11 bits
                COUNT <= 0;
                read <= 0;                                        //mark down that reading stopped
                TRIG_ARR <= 1;                                    //trigger out that the full pack of 11bits was received
                //calculate scan_err using parity bit
                if (!scan_code[10] || scan_code[0] || !(scan_code[1]^scan_code[2]^scan_code[3]^scan_code[4]
                    ^scan_code[5]^scan_code[6]^scan_code[7]^scan_code[8]
                        ^scan_code[9]))
                        scan_err <= 1;
                else
                        scan_err <= 0;
        end
        else  begin                                               //if it yet not received full pack of 11 bits
                TRIG_ARR <= 0;                                    //tell that the packet of 11bits was not received yet
                if (COUNT < 11 && count_reading >= 4000) begin    //and if after a certain time no more bits were received, then
                        COUNT <= 0;                               //reset the number of bits received
                        read <= 0;                                //and wait for the next packet
                end
        end
PREVIOUS_STATE <= PS2_CLK;                                        //mark down the previous state of the keyboard clock
end
end

always @(posedge CLK) begin
        if (TRIGGER) begin                                        //if the 250 times slower than board clock triggers
                if (TRIG_ARR) begin                               //and if a full packet of 11 bits was received
                        if (scan_err) begin                       //BUT if the packet was NOT OK
```



```
        always @(posedge CLK) begin
                if (TRIGGER) begin                                //if the 250 times slower than board clock triggers
                        if (TRIG_ARR) begin                       //and if a full packet of 11 bits was received
                                if (scan_err) begin               //BUT if the packet was NOT OK
                                        CODEWORD <= 8'd0;         //then reset the codeword register
                                end
                                else begin
                                        CODEWORD <= scan_code[8:1];    //else drop down the unnecessary  bits and transport the 7 DATA bits to CODEWORD reg
                                end                               //notice, that the codeword is also reversed! This is because the first bit to received
                        end                                       //is supposed to be the last bit in the codewordâ€¦
                        else CODEWORD <= 8'd0;                    //not a full packet received, thus reset codeword
                end
                else CODEWORD <= 8'd0;                            //no clock trigger, no dataâ€¦
        end

        always @(posedge CLK) begin
//      if (TRIGGER) begin
//              if (TRIG_ARR) begin
//              LED<=scan_code[8:1];                      //You can put the code on the LEDs if you want to, thatâ€™s up to you
//              if (CODEWORD == ARROW_UP)                         //if the CODEWORD has the same code as the ARROW_UP code
//                      LED <= LED + 1;                           //count up the LED register to light up LEDs
//              else if (CODEWORD == ARROW_DOWN)                  //or if the ARROW_DOWN was pressed, then
//                      LED <= LED - 1;                           //count down LED register
        if (CODEWORD == SPACE_BAR)
            Jump = 1;
        else if (CODEWORD == X)
            Reset = 1;

        end

endmodule
```

5.2.2 Module: pixel_gen

This module generates pixel information for a simple game display. It takes various inputs such as bird and pipe coordinates, clock signals, and control signals. The output of this module includes RGB values (red, blue, and green) for each pixel.

Inputs:

pixel_x, pixel_y: Coordinates of the pixel.

clk_div: Clock signal divided by a factor.

start: Signal to initiate the pixel generation.

video_on: Signal to enable video generation.

counter1: Unclear purpose without more context.

bird_x, bird_y: Coordinates of the bird in the game.

pipe1_x, pipe1y_up, pipe2_x, pipe2y_up, pipe3_x, pipe3y_up: Coordinates of pipes in the game.

Outputs:

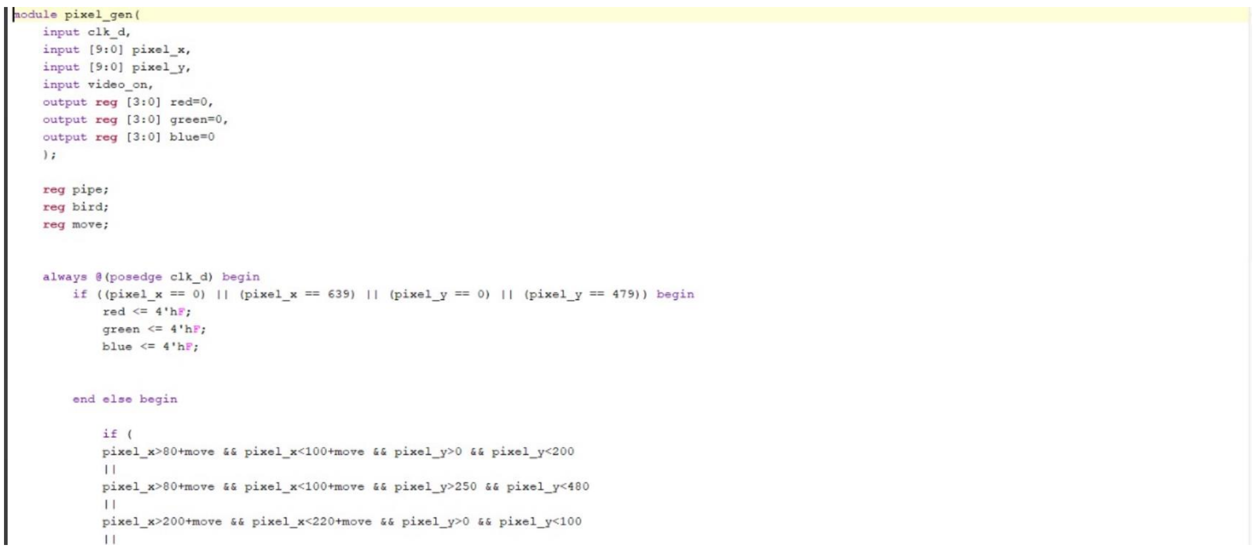
red, blue, green: RGB values for the pixel.

Functionality:

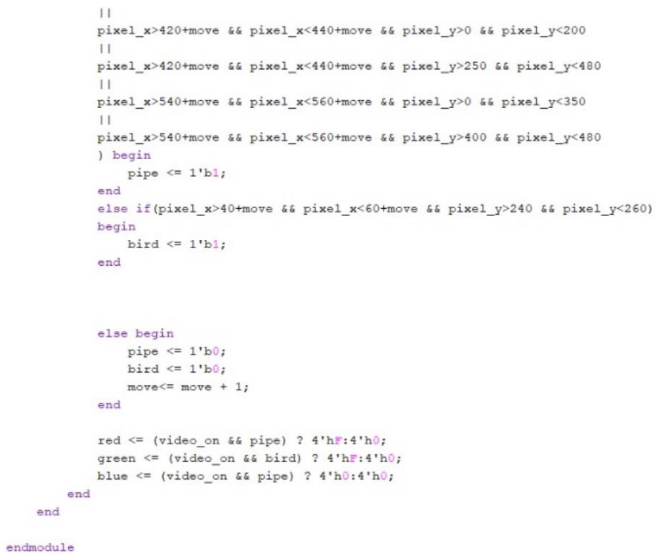
The module initializes some parameters if start signal is asserted.

It checks the pixel coordinates against bird and pipe coordinates to determine the color of the pixel.

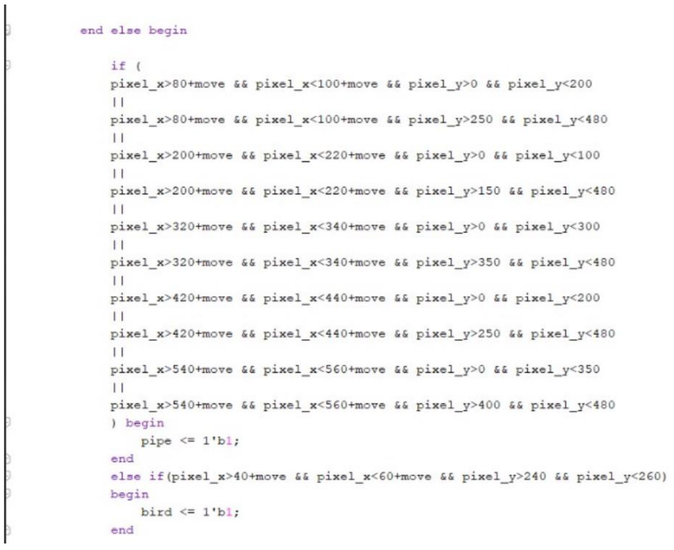
If the pixel is within the bird's coordinates, it sets the color to red. If it's within the pipe's coordinates, it sets the color to green. Otherwise, the color is set to black.



```
module pixel_gen(
    input clk_d,
    input [9:0] pixel_x,
    input [9:0] pixel_y,
    input video_on,
    output reg [3:0] red=0,
    output reg [3:0] green=0,
    output reg [3:0] blue=0
    );

    reg pipe;
    reg bird;
    reg move;


    always @(posedge clk_d) begin
        if ((pixel_x == 0) || (pixel_x == 639) || (pixel_y == 0) || (pixel_y == 479)) begin
            red <= 4'hF;
            green <= 4'hF;
            blue <= 4'hF;


        end else begin

            if (
            pixel_x>80+move && pixel_x<100+move && pixel_y>0 && pixel_y<200
            ||
            pixel_x>80+move && pixel_x<100+move && pixel_y>250 && pixel_y<480
            ||
            pixel_x>200+move && pixel_x<220+move && pixel_y>0 && pixel_y<100
            ||
```

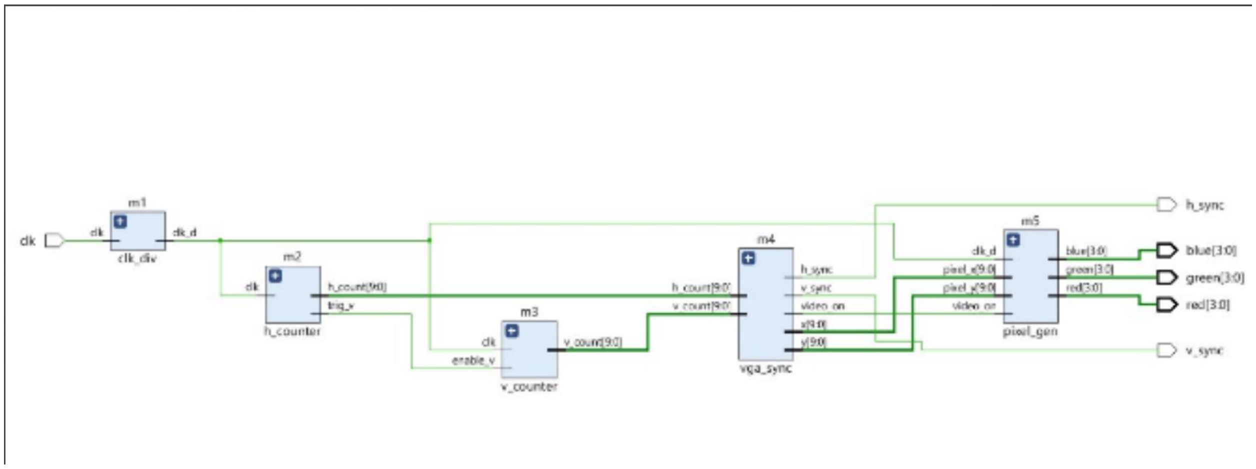


```
            ||
            pixel_x>420+move && pixel_x<440+move && pixel_y>0 && pixel_y<200
            ||
            pixel_x>420+move && pixel_x<440+move && pixel_y>250 && pixel_y<480
            ||
            pixel_x>540+move && pixel_x<560+move && pixel_y>0 && pixel_y<350
            ||
            pixel_x>540+move && pixel_x<560+move && pixel_y>400 && pixel_y<480
            ) begin
                pipe <= 1'b1;
            end
            else if(pixel_x>40+move && pixel_x<60+move && pixel_y>240 && pixel_y<260)
            begin
                bird <= 1'b1;
            end



            else begin
                pipe <= 1'b0;
                bird <= 1'b0;
                move<= move + 1;
            end

            red <= (video_on && pipe) ? 4'hF:4'h0;
            green <= (video_on && bird) ? 4'hF:4'h0;
            blue <= (video_on && pipe) ? 4'h0:4'h0;
        end
    end

endmodule
```

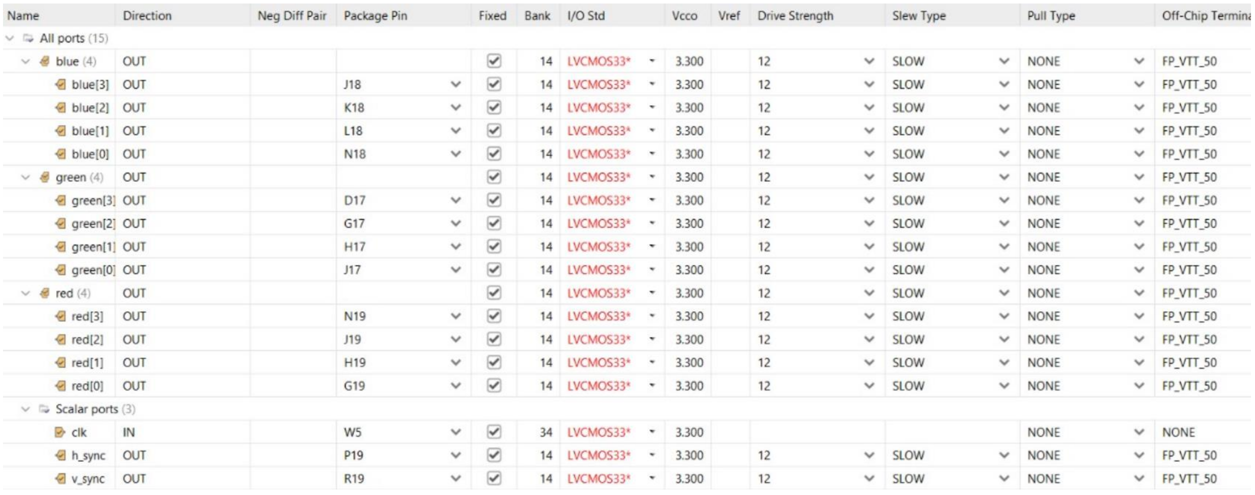


```
        end else begin

            if (
            pixel_x>80+move && pixel_x<100+move && pixel_y>0 && pixel_y<200
            ||
            pixel_x>80+move && pixel_x<100+move && pixel_y>250 && pixel_y<480
            ||
            pixel_x>200+move && pixel_x<220+move && pixel_y>0 && pixel_y<100
            ||
            pixel_x>200+move && pixel_x<220+move && pixel_y>150 && pixel_y<480
            ||
            pixel_x>320+move && pixel_x<340+move && pixel_y>0 && pixel_y<300
            ||
            pixel_x>320+move && pixel_x<340+move && pixel_y>350 && pixel_y<480
            ||
            pixel_x>420+move && pixel_x<440+move && pixel_y>0 && pixel_y<200
            ||
            pixel_x>420+move && pixel_x<440+move && pixel_y>250 && pixel_y<480
            ||
            pixel_x>540+move && pixel_x<560+move && pixel_y>0 && pixel_y<350
            ||
            pixel_x>540+move && pixel_x<560+move && pixel_y>400 && pixel_y<480
            ) begin
                pipe <= 1'b1;
            end
            else if(pixel_x>40+move && pixel_x<60+move && pixel_y>240 && pixel_y<260)
            begin
                bird <= 1'b1;
            end
```

PIXEL_GEN Elaborated Design:



Pixel_Gen I/O Ports:



Name	Direction	Neg Diff Pair	Package Pin	Fixed	Bank	I/O Std	Vcco	Vref	Drive Strength	Slew Type	Pull Type	Off-Chip Termina
All ports (15)												
blue (4)	OUT			✓	14	LVCMOS33*	3.300		12	SLOW	NONE	FP_VTT_50
blue[3]	OUT		J18	✓	14	LVCMOS33*	3.300		12	SLOW	NONE	FP_VTT_50
blue[2]	OUT		K18	✓	14	LVCMOS33*	3.300		12	SLOW	NONE	FP_VTT_50
blue[1]	OUT		L18	✓	14	LVCMOS33*	3.300		12	SLOW	NONE	FP_VTT_50
blue[0]	OUT		N18	✓	14	LVCMOS33*	3.300		12	SLOW	NONE	FP_VTT_50
green (4)	OUT			✓	14	LVCMOS33*	3.300		12	SLOW	NONE	FP_VTT_50
green[3]	OUT		D17	✓	14	LVCMOS33*	3.300		12	SLOW	NONE	FP_VTT_50
green[2]	OUT		G17	✓	14	LVCMOS33*	3.300		12	SLOW	NONE	FP_VTT_50
green[1]	OUT		H17	✓	14	LVCMOS33*	3.300		12	SLOW	NONE	FP_VTT_50
green[0]	OUT		J17	✓	14	LVCMOS33*	3.300		12	SLOW	NONE	FP_VTT_50
red (4)	OUT			✓	14	LVCMOS33*	3.300		12	SLOW	NONE	FP_VTT_50
red[3]	OUT		N19	✓	14	LVCMOS33*	3.300		12	SLOW	NONE	FP_VTT_50
red[2]	OUT		J19	✓	14	LVCMOS33*	3.300		12	SLOW	NONE	FP_VTT_50
red[1]	OUT		H19	✓	14	LVCMOS33*	3.300		12	SLOW	NONE	FP_VTT_50
red[0]	OUT		G19	✓	14	LVCMOS33*	3.300		12	SLOW	NONE	FP_VTT_50
Scalar ports (3)												
clk	IN		W5	✓	34	LVCMOS33*	3.300				NONE	NONE
h_sync	OUT		P19	✓	14	LVCMOS33*	3.300		12	SLOW	NONE	FP_VTT_50
v_sync	OUT		R19	✓	14	LVCMOS33*	3.300		12	SLOW	NONE	FP_VTT_50

5.3 Control Block

The game consists of three main states which are represented by three different screen displays:

1. Start State Screen
2. Game Screen
3. Restart State Screen

5.3.1. Start State Screen

Our Start State Screen would show the beginning state of our game with the pipes and our hero block in a static state.

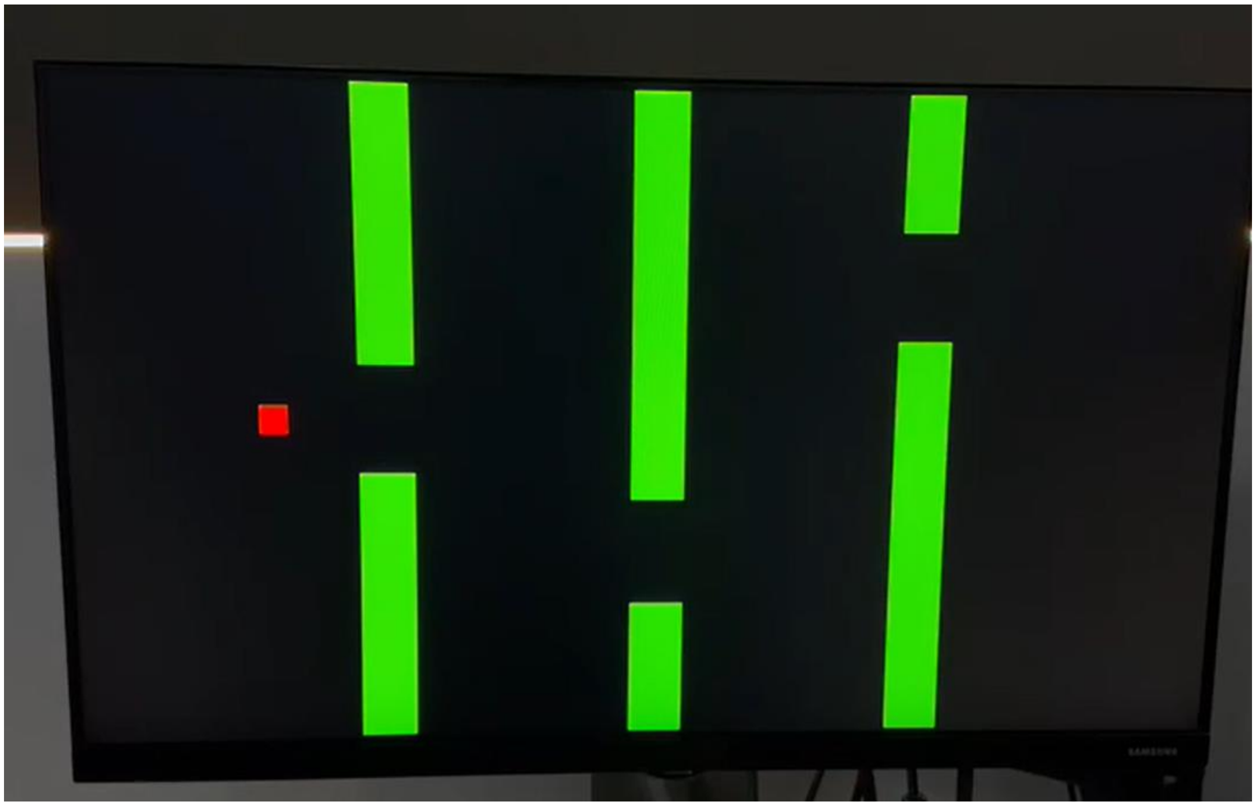


Fig 1: Start Screen Static State

5.3.2. Game Screen

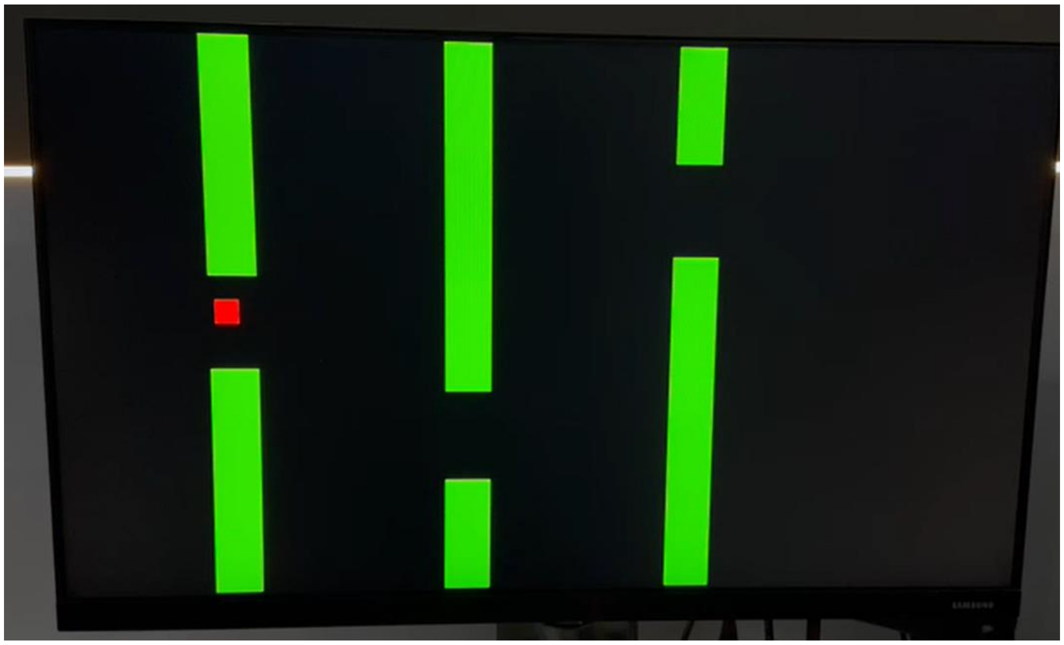


Fig 2: Game Screen

5.3.2.1 MODULES:

Module: mover

This module manages the movement of game elements (bird and pipes) based on input directions and resets.

Inputs:

clk: Clock signal.

dir: Input direction (UP, DOWN, RESET).

reset2: Reset signal.

bird_x, bird_y: Coordinates of the bird.

pipe1_x, pipe1y_up, pipe2_x, pipe2y_up, pipe3_x, pipe3y_up: Coordinates of pipes.

Outputs:

reset2: Output reset signal.

bird_x, bird_y: Updated coordinates of the bird.

pipe1_x, pipe1y_up, pipe2_x, pipe2y_up, pipe3_x, pipe3y_up: Updated coordinates of pipes.

Functionality:

The module updates the positions of the bird and pipes based on the input direction and resets.

It handles boundary conditions and resets the game if the bird collides with pipes or goes out of bounds.

Module: score

This module handles the scoring and display on a 7-segment LED.

Inputs:

clock_100Mhz: 100 MHz clock source.

reset: Reset signal.

Outputs:

Anode_Activate: Anode signals for the 7-segment display.

LED_out: Cathode patterns for the 7-segment display.

Functionality:

The module keeps track of time using a counter.

It updates the displayed number every second.

The displayed number is then converted to 7-segment LED patterns.

Module: vga_sync

This module generates synchronization signals for VGA output.

Inputs:

h_count, v_count: Horizontal and vertical pixel counters.

Outputs:

h_sync, v_sync: Horizontal and vertical synchronization signals.

video_on: Signal indicating whether to display video.

x_loc, y_loc: X and Y coordinates of the pixel.

Functionality:

The module generates synchronization signals based on pixel counters and determines whether to display video or not.

Module: v_counter

This module generates a vertical counter.

Inputs:

clk: Clock signal.

enable_v: Enable signal.

Outputs:

v_count: Vertical pixel counter.

Functionality:

The module increments the vertical pixel counter when enabled.

Module: h_counter

This module generates a horizontal counter and triggers a vertical counter.

Inputs:

clk: Clock signal.

Outputs:

h_count: Horizontal pixel counter.

trig_v: Trigger signal for the vertical counter.

Functionality:

The module increments the horizontal pixel counter and triggers the vertical counter at the end of each line.

Module: clk_div

This module divides the input clock signal.

Inputs:

clk: Input clock signal.

Outputs:

clk_d: Divided clock signal.

Functionality:

The module generates a divided clock signal.

Module: TopLevelModule

This is the top-level module that instantiates and connects all the sub-modules. It defines the overall structure of the game logic, including input handling, pixel generation, and scoring. The *Top* module takes in a 100 MHz onboard clock and a reset signal. It outputs VGA horizontal sync and vertical sync signals, 4-bit VGA Red, Green, and Blue color values, and receives input signals for left and right for the hero's control.

Inputs:

clk: Clock signal.

kb_clk, kb_data: PS2 keyboard clock and data signals.

start: Start signal.

Outputs:

h_sync, v_sync: VGA synchronization signals.

red, blue, green: RGB values for pixel generation.

dir: Direction signal for game movement.

an_act: Anode signals for 7-segment display.

LED: Cathode patterns for 7-segment display.

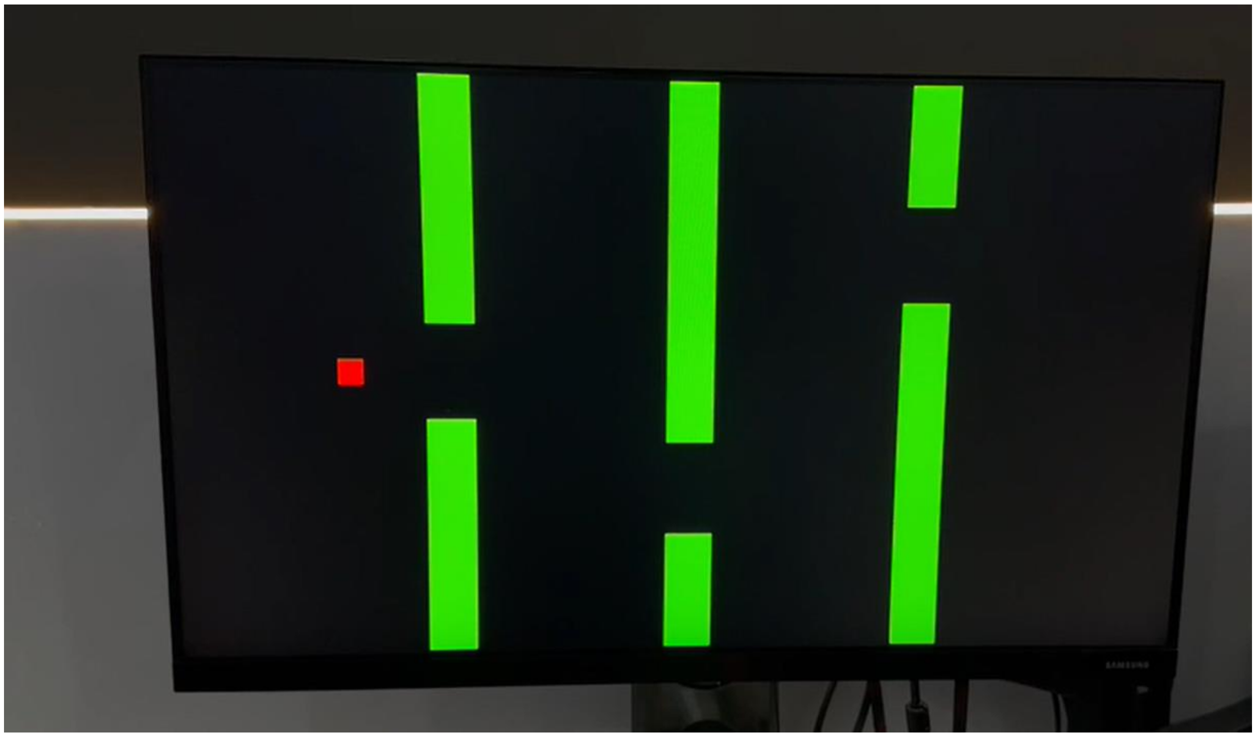
Functionality:

The module instantiates and connects all the sub-modules to create the overall game logic.

Updates VGA monitor display based on game logic output.

5.3.3. Reset State Screen

Upon pressing the X key or reaching the score of 30, we reach the Restart State screen. Our Restart State Screen would show the beginning state of our game with the pipes and our hero block in a static state.



5.4 Display Screen (Output Block)

The display screen is generated using the VGA (video graphic array) connector. The display which we are using is standard 640 x 480 pixels.
For imaging on the horizontal axis, we turn the video on from 0 till 639 pixels and for the vertical axis the video is on from 0 till 479 pixels as that is the main display, after that the video is turned off for borders and retracing.
To make the final screen we first used a clock divider to reduce the frequency from 100 MHz to 25 MHz for FPGA to work, then we used h counter to count the pixels of horizontal axis and counter for vertical axis. Then VGA sync was used for turning the video on on the required part of the screen and finally pixel gen is used to create our desired screen.

Following pin configuration is used to connect the VGA connector to the FPGA board.

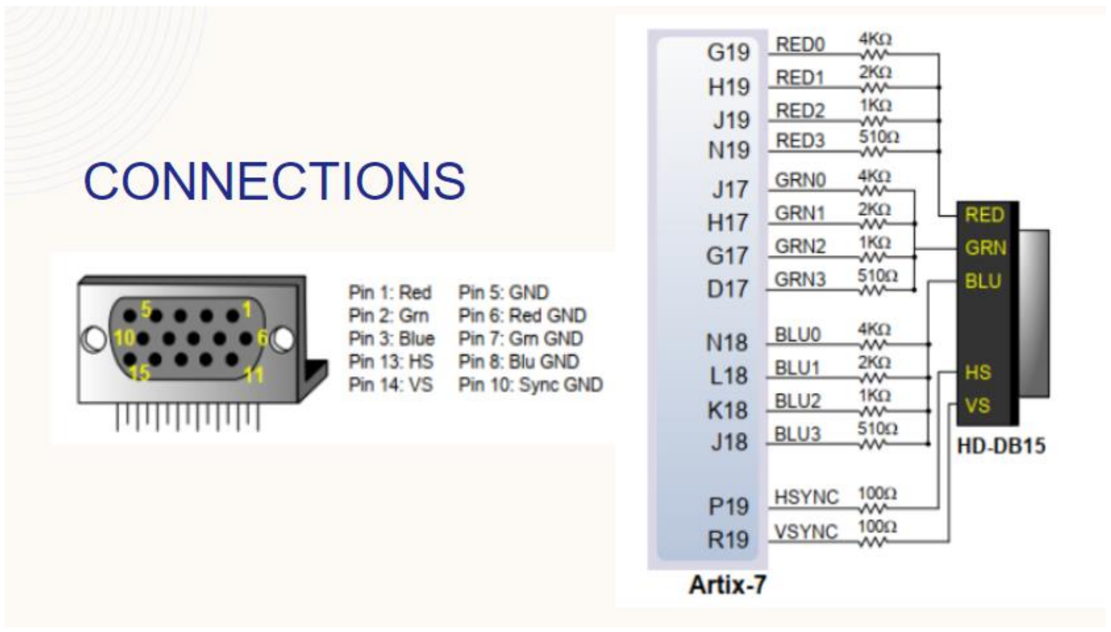
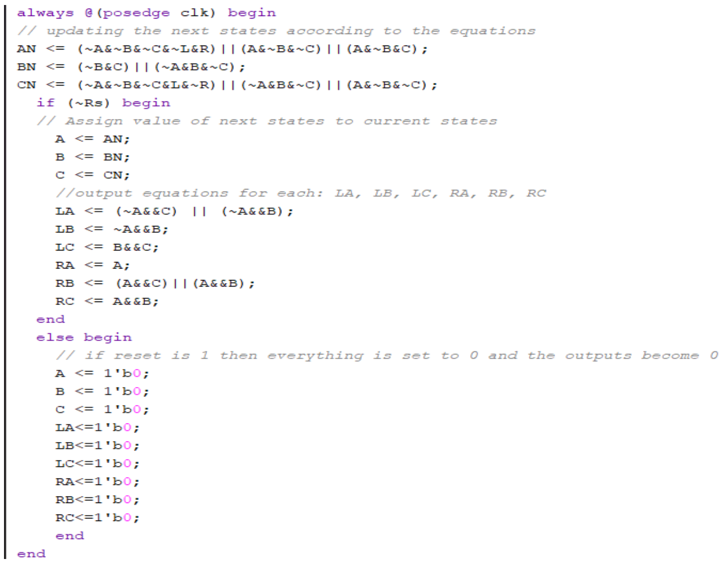


Fig 11: VGA Connections and RGB Ports*

6. FSM:

Although the start and end states of the game, dictated by the start game and end game screens, take the switch and reset button as input for state transitions, our main game itself does not make any state changes based on any kind of input from the user and instead terminates automatically when the reset button is pressed or a score of 30 is crossed. This transition to the end state can be seen as solely dependent on the gameplay state i.e. the next state depends on the initial state. So we can conclude that the FSM implemented in our game is a **Moore Machine** as our next state of our game is not dependent on the input block.

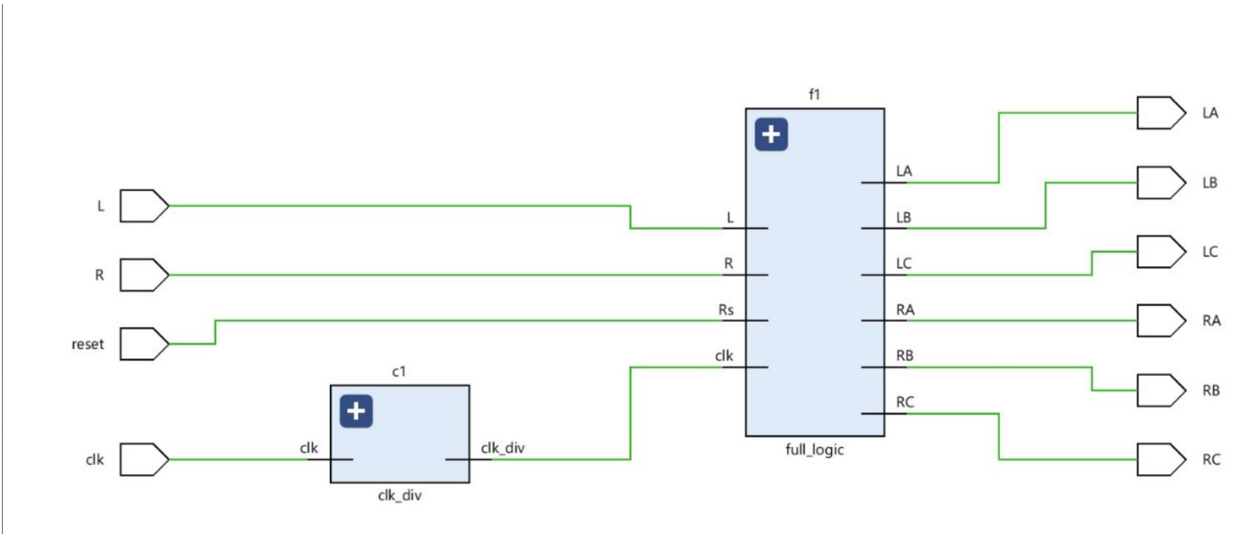
We were also able to generate the FSM for our project which is as follows:



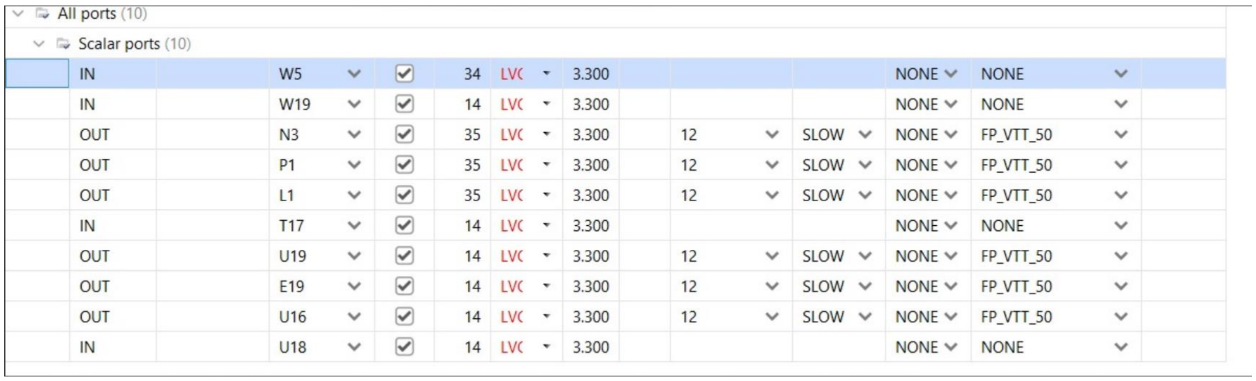
```
always @(posedge clk) begin
// updating the next states according to the equations
AN <= (~A&~B&~C&~L&R)||(A&~B&~C)||(A&~B&C);
BN <= (~B&C)||(~A&B&~C);
CN <= (~A&~B&~C&L&~R)||(~A&B&~C)||(A&~B&~C);
  if (~Rs) begin
  // Assign value of next states to current states
    A <= AN;
    B <= BN;
    C <= CN;
    //output equations for each: LA, LB, LC, RA, RB, RC
    LA <= (~A&&C) || (~A&&B);
    LB <= ~A&&B;
    LC <= B&&C;
    RA <= A;
    RB <= (A&&C)||(A&&B);
    RC <= A&&B;
  end
  else begin
    // if reset is 1 then everything is set to 0 and the outputs become 0
    A <= 1'b0;
    B <= 1'b0;
    C <= 1'b0;
    LA<=1'b0;
    LB<=1'b0;
    LC<=1'b0;
    RA<=1'b0;
    RB<=1'b0;
    RC<=1'b0;
    end
end
```

```
always @(posedge clk) begin
// updating the next states according to the equations
AN <= (~A&~B&~C&~L&R)||(A&~B&~C)||(A&~B&C);
BN <= (~B&C)||(~A&B&~C);
CN <= (~A&~B&~C&L&~R)||(~A&B&~C)||(A&~B&~C);
  if (~Rs) begin
  // Assign value of next states to current states
    A <= AN;
    B <= BN;
    C <= CN;
    //output equations for each: LA, LB, LC, RA, RB, RC
    LA <= (~A&&C) || (~A&&B);
    LB <= ~A&&B;
    LC <= B&&C;
    RA <= A;
    RB <= (A&&C)||(A&&B);
    RC <= A&&B;
  end
  else begin
    // if reset is 1 then everything is set to 0 and the outputs become 0
    A <= 1'b0;
    B <= 1'b0;
    C <= 1'b0;
    LA<=1'b0;
    LB<=1'b0;
    LC<=1'b0;
    RA<=1'b0;
    RB<=1'b0;
    RC<=1'b0;
    end
end
```

6.1 FSM Elaborated Design:



6.2 I/O PORTS:



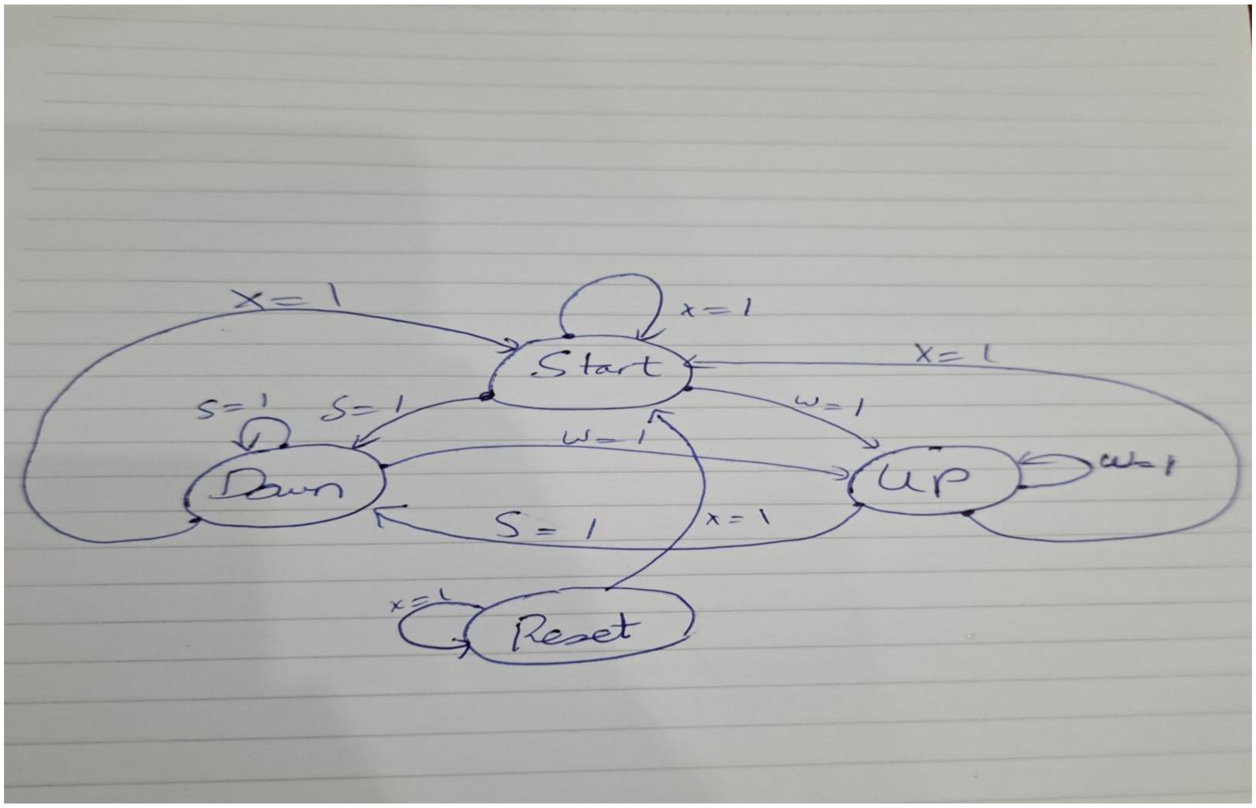
All ports (10)

Scalar ports (10)

	IN		W5	✓	34	LVC	3.300			NONE	NONE
	IN		W19	✓	14	LVC	3.300			NONE	NONE
	OUT		N3	✓	35	LVC	3.300	12	SLOW	NONE	FP_VTT_50
	OUT		P1	✓	35	LVC	3.300	12	SLOW	NONE	FP_VTT_50
	OUT		L1	✓	35	LVC	3.300	12	SLOW	NONE	FP_VTT_50
	IN		T17	✓	14	LVC	3.300			NONE	NONE
	OUT		U19	✓	14	LVC	3.300	12	SLOW	NONE	FP_VTT_50
	OUT		E19	✓	14	LVC	3.300	12	SLOW	NONE	FP_VTT_50
	OUT		U16	✓	14	LVC	3.300	12	SLOW	NONE	FP_VTT_50
	IN		U18	✓	14	LVC	3.300			NONE	NONE

6.3 FSM State Diagram

So the initial state of our FSM would be the start state in which we will se a static screen with our hero block positioned in the middle of the screen and our pipes drawn out. On pressing the W or Spacebar key on our PS2 keyboard the game will move to the UP state in which the hero block will move up until the button for the down state S or ENTER is pressed or our hero block collides with the pipes, and in that case the game would begin from the last state it was in either UP or DOWN. On pressing the X button the game will go to RESET State and then immediately go back to our initial state which was START.



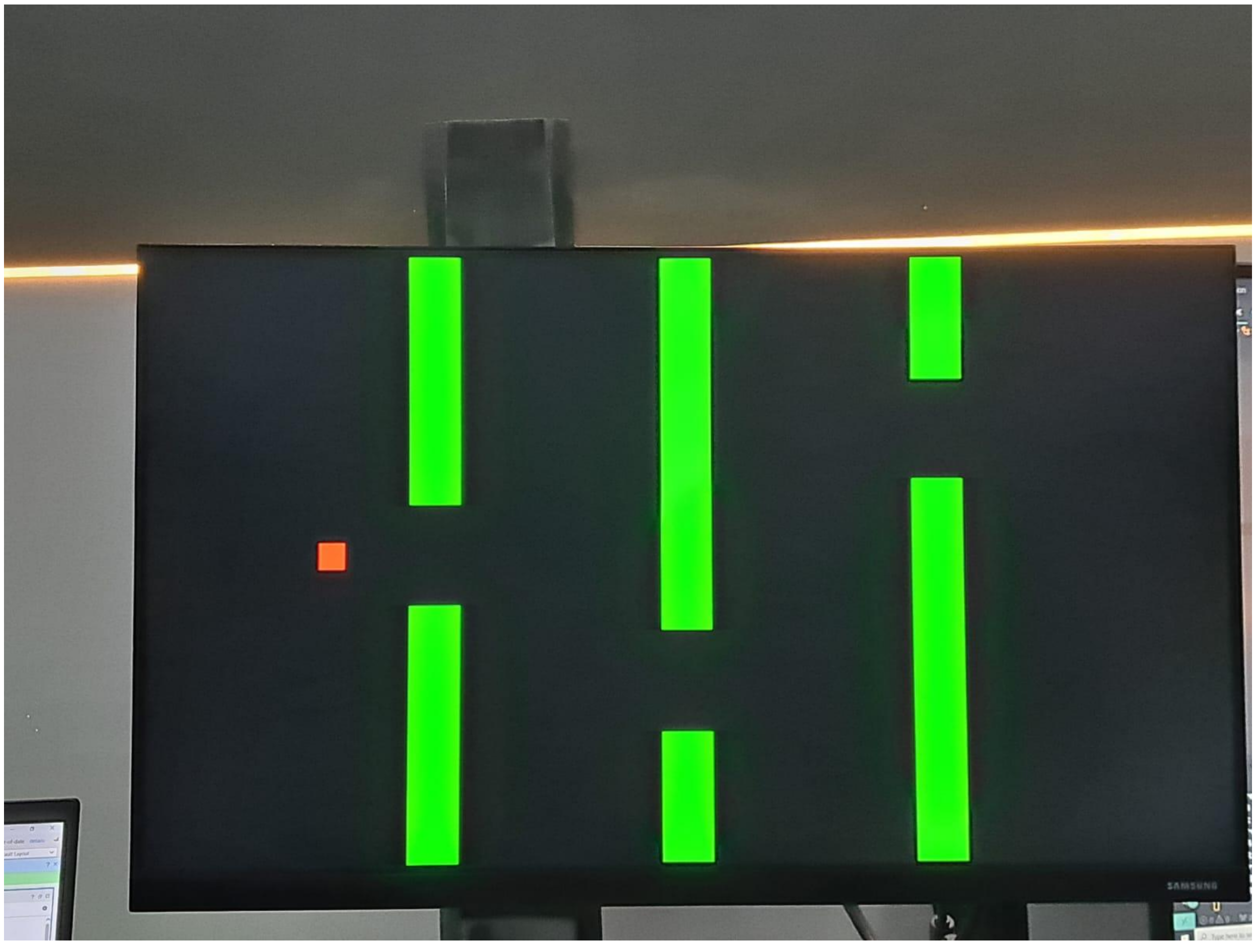
7. Pixel Mapping:

The gap between the tubes in our output display will be 100 pixels. This will remain constant throughout the game. The tube height of both the sides will be varying because the obstacles have to be made in a challenging manner so the user has to use the key to navigate the flappy bird up and down. The width of the tube height is 10 pixels of the screen. The dimensions of the flappy bird are 20x25. This means that the x component of the bird is 20 pixels wide and the y component is 25 pixels long. Each input press will jump the bird 30 pixels upward in the y direction. Every time the user provides an input for a jump, the bird will have 6 transitions of 5 pixels each in order to complete the 20 pixel jump, and the transitions will complete regardless of the input from the user at those instances. The game will read the input after all 6 transitions have been completed. During the downfall the bird will have transitions of 5 pixels each (downwards) for as long as it’s in the downfall state.

8. Major Challenges Faced

The major challenges we had faced in this project was limiting the game to 30 seconds as we had to implement another RESET condition in our score and mover module and it kept interfering with our current reset condition and caused us many problems.

Another problem we faced was that sometimes our score module wouldn't work properly and the 7 segment display on our FPGA board wouldn't record the score Sometimes our Pixel Gen would glitch out and we would have no display on our screen



8. Conclusion:

"Mario's Flappy Adventure" promises to offer an engaging fusion of the Flappy Bird gameplay mechanics with the beloved Mario universe. This project aims to showcase our prowess in digital logic design and deliver an entertaining gaming experience. The following report introduces the readers to the game flappy bird that is made by the group. In the report, the readers are being introduced to the user-flow diagram of the project and the division of blocks in the user-flow diagram. In addition, the report introduces the users to the detailed input block, Pixel_gen module and the FSM, and the understanding of their I/O ports. Along with the VGA picture we are able to generate currently. We are working on the keyboard input module currently as it is incomplete so it is not in the report.

9.References:

- "FPGA Prototyping by Verilog Examples" by Pong P. Chu: This book provides practical examples and step-by-step guidance for implementing FPGA-based designs using Verilog.
- "Digital Design and Computer Architecture" by David Money Harris and Sarah L. Harris: This book can serve as a comprehensive reference for digital logic design principles and concepts.
- Online FPGA Documentation: Refer to the official documentation provided by the FPGA manufacturer for detailed information on hardware implementation, pin configurations, and interfacing with input/output devices.

- Online Game Development Communities: Engage with online communities and forums dedicated to game development and FPGA-based projects. Platforms like GitHub, Stack Overflow, and specialized FPGA forums can provide valuable insights, code samples, and troubleshooting assistance.
- Audio output: http://www.instructables.com/id/ Basys3-FPGA-Digital-Audio-Synthesizer/
- Key scan codes and IO pins: https://digilent.com/reference/programmable-logic/basys-3/start