Project Final Report

Flappy Bird

Ali Asghar Yousuf - ay06993
Shahid Mahmood - mm06600
Azeem Haider - mh06858
Musab Sattar - ms06651

December 2021

Digital Logic and Design
Supervisors: Junaid Ahmed Memon - Hafsa Amanullah

Contents

1 Introduction

Flappy bird exploded in popularity when it was first introduced in the apple Appstore and the google play store. The game quickly rose to popularity and in the short span of time became a fan favorite game that most wanted to play in their spare time. This popularity and boost prompted a lot of market competitors to try and replicate the game and somehow mimic the popularity and monetary gain that the developers of flappy bird experienced. This could not happen and the mimickers and fake developers of the game could never copy the success. The huge success of the game meant, it was soon taken out of the play store and the app store sent behind a pay wall that no one appreciates. The game was taken away from us, the childhood game that made us want to beat the high scores of our peers. Therefore, years later, the group has decided to make our very own flappy bird and compete.

The project is simple and subtle in its delivery physically, but emotionally it has a pretty big impact. The bird on the x co-ordinate will not move, however, on the y-axis the bird will be moving up and down to navigate between the tubes. The game will be played with the help of the input which will be our keyboard, the bird will move upward in the sky when the W key is pressed, while the bird will move downward if the S key is pressed. The user has to make sure that he/she masterfully navigates the bird to glory by navigating between the tubes and creating a high score that no one else can beat. The project will be a great trip down the memory lane for the users that have played and enjoyed this game in the past but this in no way means that new users or users that haven't played this game before will not enjoy this. The target user is anyone and everyone having a liking for casual games but they still have the competitive attitude and desire to do great.

2 User-Flow Diagram and Division of tasks between blocks

2.1 User-Flow diagram

The following is the user flow diagram of our project:

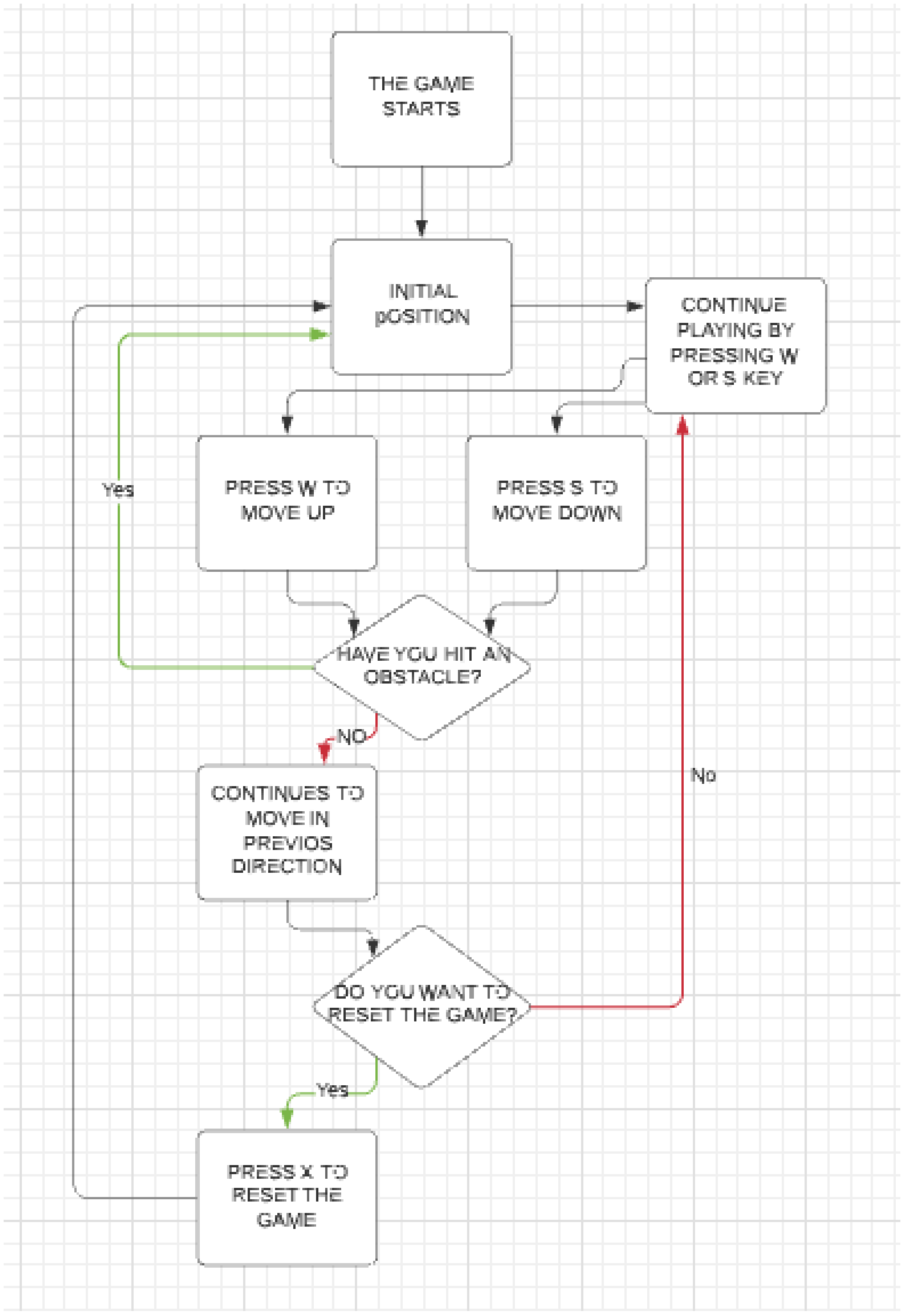

THE GAME STARTS
INITIAL pOSITION
CONTINUE PLAYING BY PRESSING W OR S KEY
PRESS W TO MOVE UP
PRESS S TO MOVE DOWN
Yes
HAVE YOU HIT AN OBSTACLE?
NO
CONTINUES TO MOVE IN PREVIOS DIRECTION
No
DO YOU WANT TO RESET THE GAME?
Yes
PRESS X TO RESET THE GAME

Figure 1: User Flow Diagram

2.2 Explanation:

The user-flow diagram is elegantly made indicating all the possibilities for a user in this game. The user is controlling three keys. By pressing the X key the user can go into reset position at any time. The W key which will make the bird move in an upward direction, the bird will continue to move in an upward direction until the X or the S key is pressed. The S key will make the bird move in a downward position and similar to the W key, the s key will make the bird go downwards until W or X key is pressed.

It is extremely essential for the success of the user to properly navigate the bird and not hit the tubes while controlling the bird. The user can either move up by pressing W or descend downwards by pressing the S key. The tubes will keep randomly generating until the game is over and the user crashes the bird with a tube. The user will be granted two points each time it accurately navigates through one set of tubes.

2.3 Division of Task:

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

1. The tasks of the input block of our project is when we press the keys X, W or S it has 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 bird 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 W key is pressed, the flappy bird will ascend(move up) until another key is pressed and descend(move down) if the S key is pressed. If the bird collides with any of the walls of the tubes or touches the upper or lower frame of the screen the game finishes and output is generated as 'GAME OVER', taking u directly to reset state.

3 Input Block

3.1 Introduction to the input peripheral: The Keyboard

The Artix 7 FPGA board is powered by connecting it to the CPU 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.
The keyboard is attached to the FPGA board at the USB connector J2. The host capability of USB HID in Basys 3 is provided by micro controller which switches to the USB host and turns on application mode

3.2 Inputs in the Project

In order to move our bird up, we press the key 'w'. Pressing 'w', moves the bird upwards and it keeps moving upwards until it crashes or another different input is received.
Similarly, to move the bird down, we press the key 's'. Once we press 's', the bird will keep moving downwards until it either collides or another different input is received. In order to exit/reset the game, we press 'x'. Pressing the reset key 'x', resets the game to its idle state regardless of the game progress.

3.3 Configuring the keyboard

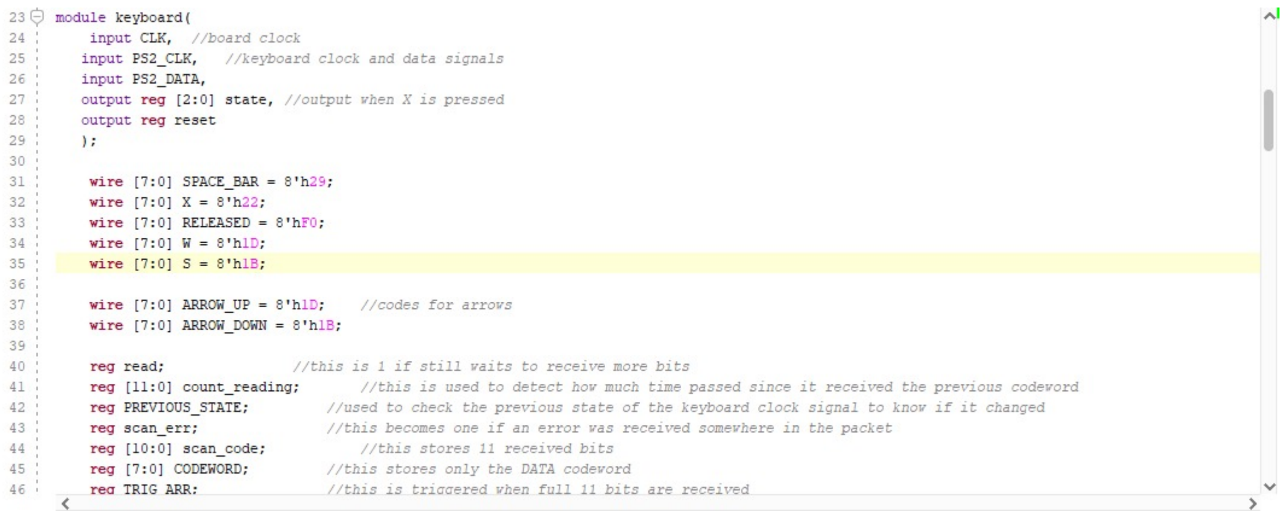
We have interfaced the USB keyboard to give scan codes of PS/2 keyboard as the FPGA only takes PS/2 keyboard inputs. When we press either 'w' or 's' on the keyboard, the unique scan code of that key is generated and the game starts or continues meaning if it's the initial press then the game starts but if it is not the first press then we are already in the game and the bird is navigated by using these presses. Similarly we have an exit key 'x'. This is a reset button which resets the game and goes back to the idle state no matter where the player currently is. Pressing x also generates a unique hexadecimal scan code that we interpret, similar to how the 'w' and 's' key's hexadecimal scan code.

Scan codes:
W = 1D
S = 1B
X= 22

3.4 Input translation to Output

The bird moves up when the key 'w' is pressed. It moves down when the key 's' is pressed. The clock of the keyboard sends the data on the rising edges. An important thing to note here is that the the keys' ('w' and 's') functionality will be designed such that once either of the keys are pressed, the bird will keep moving in that direction until it either collides or a different input is provided.

3.5 Input Code



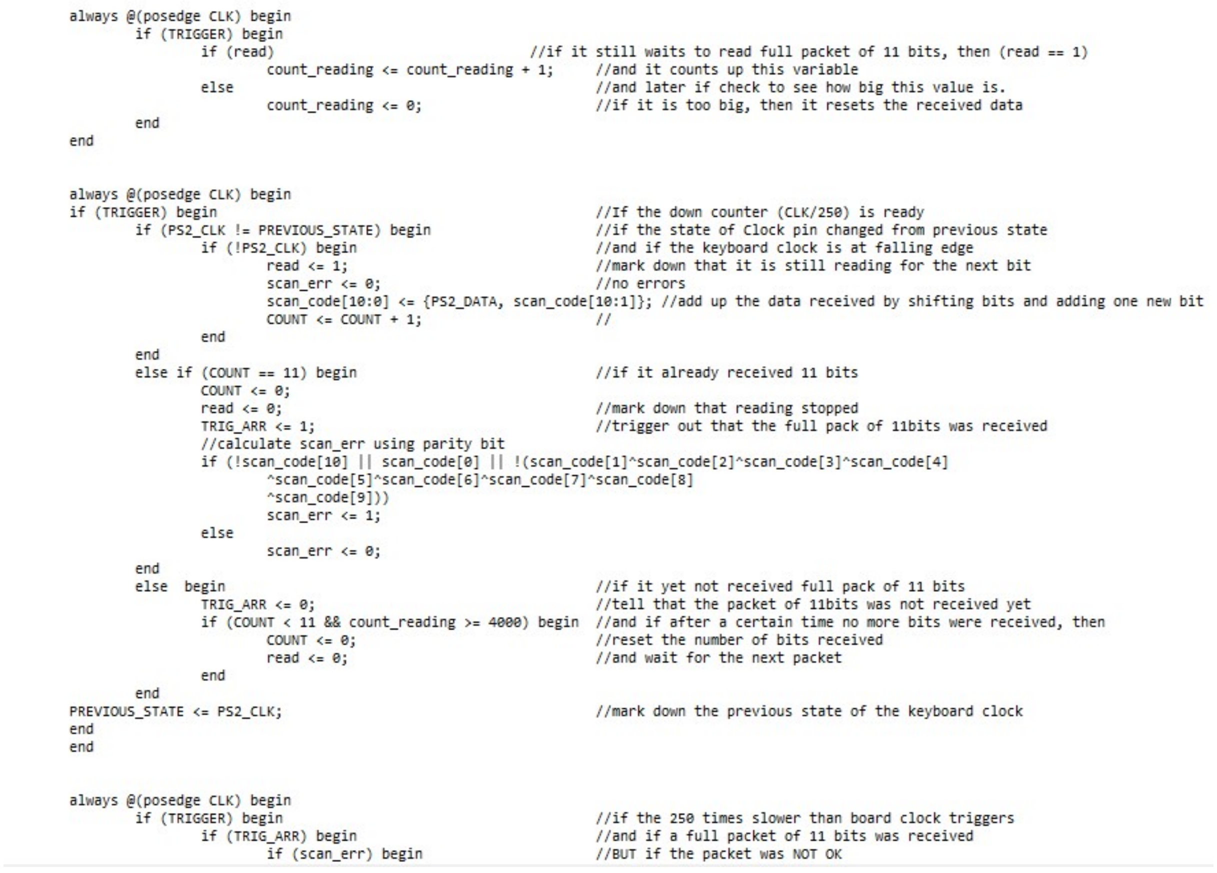
```
module keyboard(
     input CLK,  //board clock
    input PS2_CLK,   //keyboard clock and data signals
    input PS2_DATA,
    output reg [2:0] state, //output when X is pressed
    output reg reset
    );

     wire [7:0] SPACE_BAR = 8'h29;
     wire [7:0] X = 8'h22;
     wire [7:0] RELEASED = 8'hF0;
     wire [7:0] W = 8'h1D;
     wire [7:0] S = 8'h1B;

     wire [7:0] ARROW_UP = 8'h1D;    //codes for arrows
     wire [7:0] ARROW_DOWN = 8'h1B;

     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
```

Figure 2: input code1

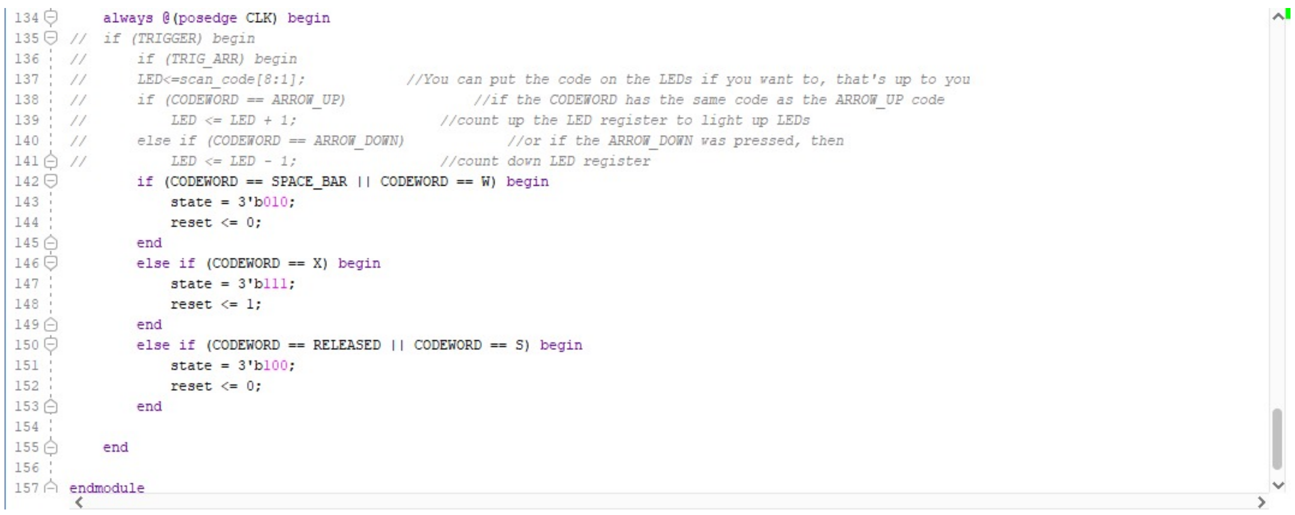


```
always @(posedge CLK) begin
        if (TRIGGER) begin
                if (read)                               //if it still waits to read full packet of 11 bits, then (read == 1)
                        count_reading <= count_reading + 1;     //and it counts up this variable
                else                                            //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
```

Figure 3: input code2



```
    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, that'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 || CODEWORD == W) begin
            state = 3'b010;
            reset <= 0;
        end
        else if (CODEWORD == X) begin
            state = 3'b111;
            reset <= 1;
        end
        else if (CODEWORD == RELEASED || CODEWORD == S) begin
            state = 3'b100;
            reset <= 0;
        end

    end

endmodule
```

Figure 4: input code3

3.6 Input Elaborated Diagram

Here is the elaborated diagram of the input:

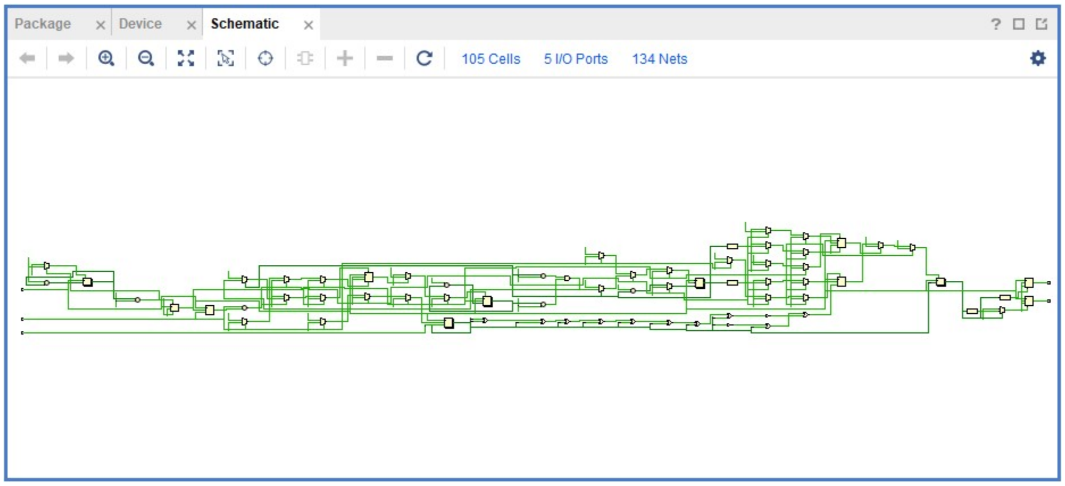
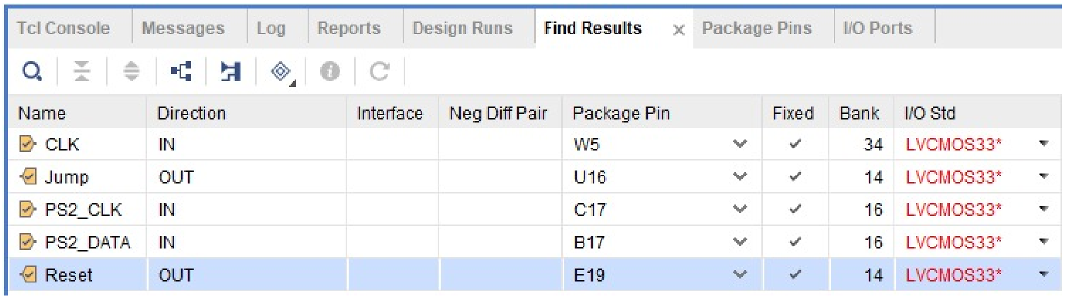


Figure 5: Elaborated Design of Input block

3.7 Input I/O Ports

Port configuration:
$PS2\_CLK : C17$
$PS2\_DATA : B17$
$CLK : W5$


Tcl Console | Messages | Log | Reports | Design Runs | Find Results | Package Pins | I/O Ports

Name	Direction	Interface	Neg Diff Pair	Package Pin	Fixed	Bank	I/O Std
CLK	IN			W5	✓	34	LVCMOS33*
Jump	OUT			U16	✓	14	LVCMOS33*
PS2_CLK	IN			C17	✓	16	LVCMOS33*
PS2_DATA	IN			B17	✓	16	LVCMOS33*
Reset	OUT			E19	✓	14	LVCMOS33*

Figure 6: IO Ports used

4 Output Block

4.1 Introduction to the Output Block

The output block of our project is the VGA display. The most important part of our project is the output and the VGA display. What appears on the screen makes the user excited for the game that they are about to play. The following is how the group decided to map the pixels in the output display of our project.

4.2 Pixel Mapping

The gap between the tubes in our output display will be 80 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 40 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 30 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. Where as during the downfall the bird will have transitions of 5 pixels each (downwards) for as long as its in the downfall state.

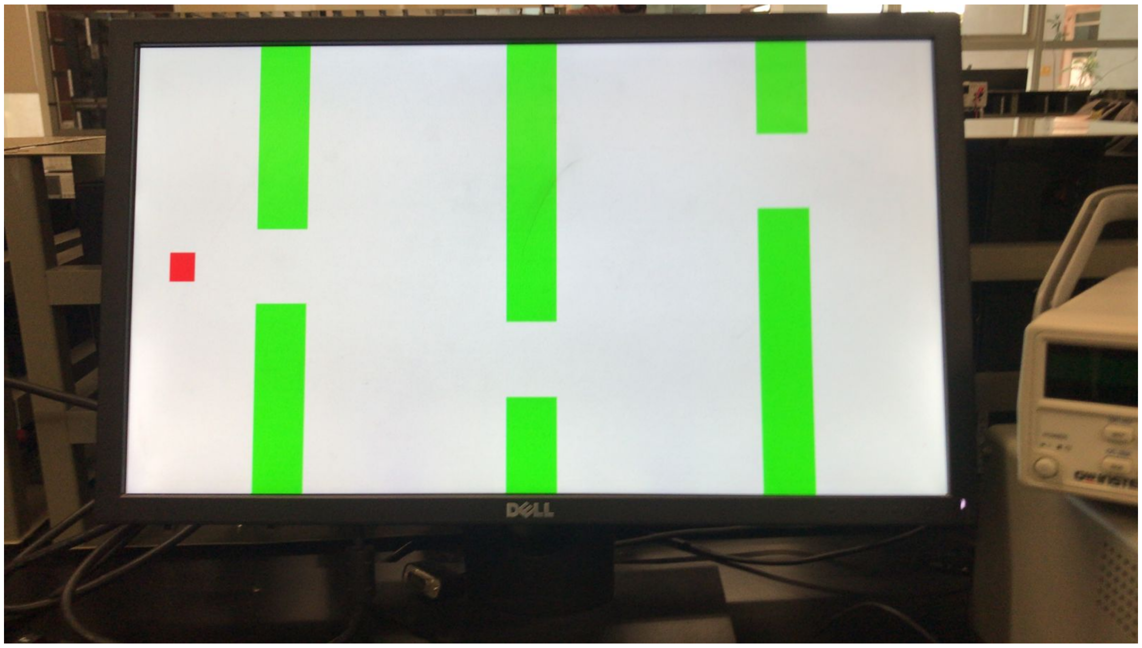
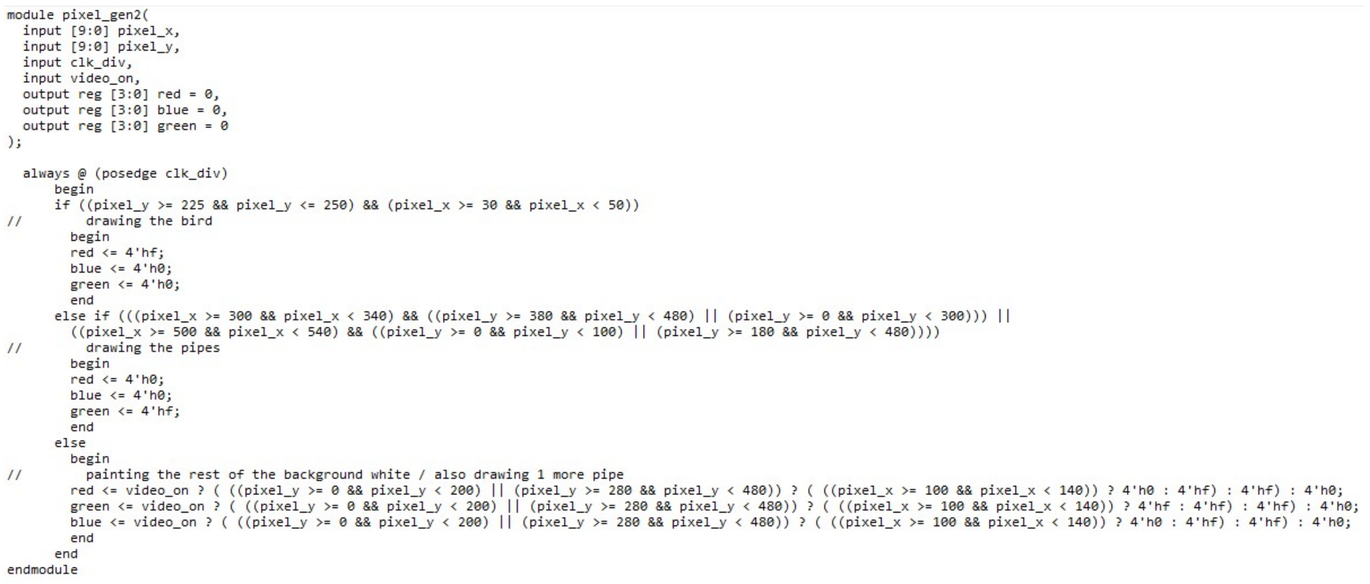


Figure 7: Display screen of the game

4.3 Output Demo Code

Following is the display output and the output code for the demo display:

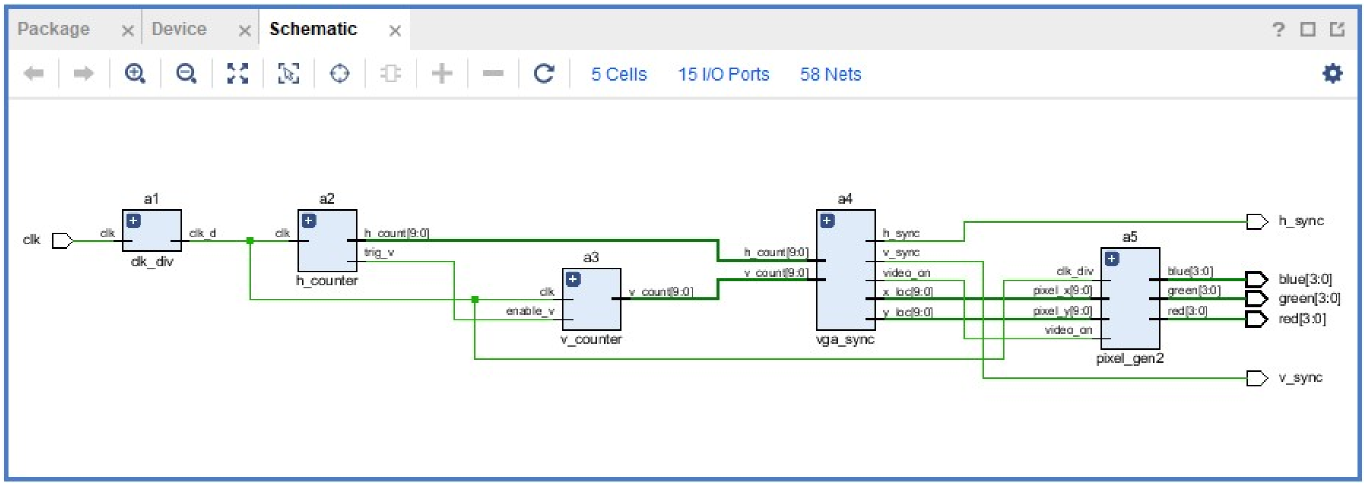


```
module pixel_gen2(
  input [9:0] pixel_x,
  input [9:0] pixel_y,
  input clk_div,
  input video_on,
  output reg [3:0] red = 0,
  output reg [3:0] blue = 0,
  output reg [3:0] green = 0
);

  always @ (posedge clk_div)
      begin
      if ((pixel_y >= 225 && pixel_y <= 250) && (pixel_x >= 30 && pixel_x < 50))
//          drawing the bird
        begin
        red <= 4'hf;
        blue <= 4'h0;
        green <= 4'h0;
        end
      else if (((pixel_x >= 300 && pixel_x < 340) && ((pixel_y >= 380 && pixel_y < 480) || (pixel_y >= 0 && pixel_y < 300))) ||
        ((pixel_x >= 500 && pixel_x < 540) && ((pixel_y >= 0 && pixel_y < 100) || (pixel_y >= 180 && pixel_y < 480))))
//          drawing the pipes
        begin
        red <= 4'h0;
        blue <= 4'h0;
        green <= 4'hf;
        end
      else
        begin
//          painting the rest of the background white / also drawing 1 more pipe
        red <= video_on ? ( ((pixel_y >= 0 && pixel_y < 200) || (pixel_y >= 280 && pixel_y < 480)) ? ( ((pixel_x >= 100 && pixel_x < 140)) ? 4'h0 : 4'hf) : 4'hf) : 4'h0;
        green <= video_on ? ( ((pixel_y >= 0 && pixel_y < 200) || (pixel_y >= 280 && pixel_y < 480)) ? ( ((pixel_x >= 100 && pixel_x < 140)) ? 4'hf : 4'hf) : 4'hf) : 4'h0;
        blue <= video_on ? ( ((pixel_y >= 0 && pixel_y < 200) || (pixel_y >= 280 && pixel_y < 480)) ? ( ((pixel_x >= 100 && pixel_x < 140)) ? 4'h0 : 4'hf) : 4'hf) : 4'h0;
        end
      end
endmodule
```

4.4 Output Elaborated Diagram

Here is the elaborated diagram of the sample output:



4.5 Output I/O Ports

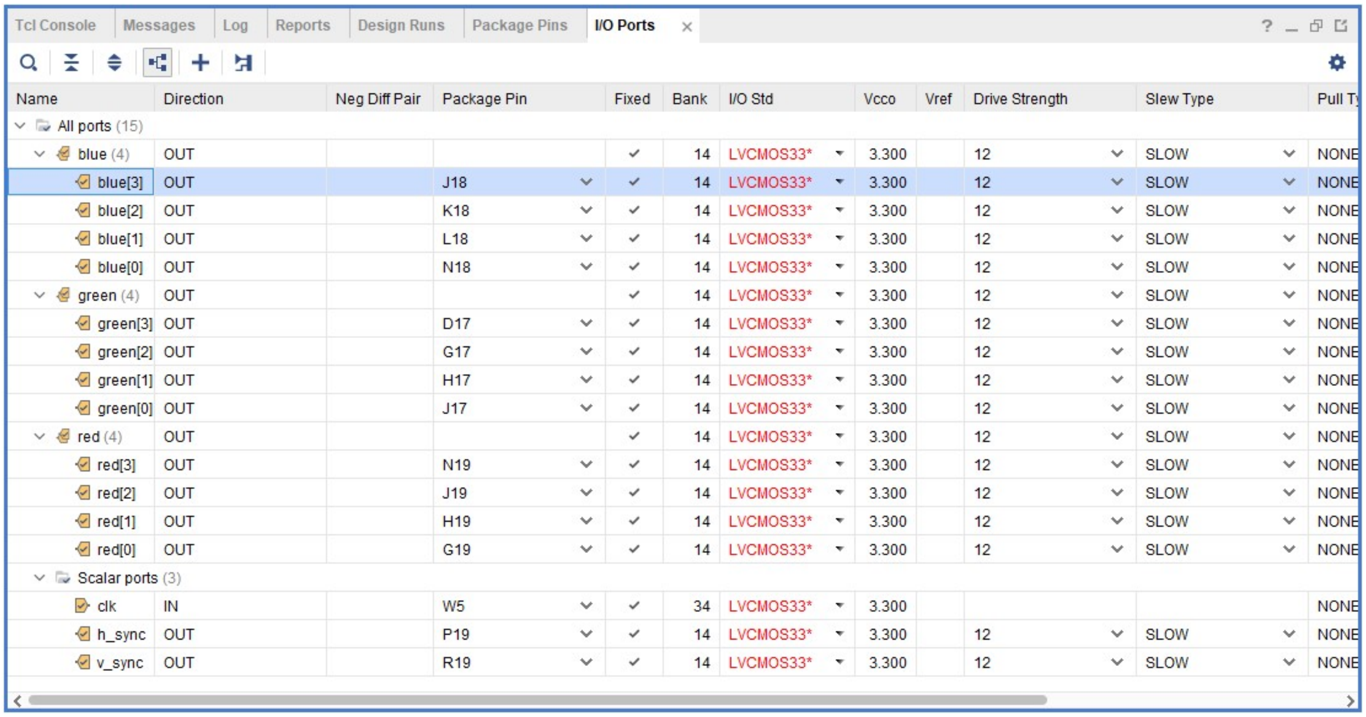
Port configuration:

$blue[3] : J18$
$blue[2] : K18$
$blue[1] : L18$
$blue[0] : N18$

$green[3] : D17$
$green[2] : G17$
$green[1] : H17$
$green[0] : J17$

$red[3] : N19$
$red[2] : J19$
$red[1] : H19$
$red[0] : G19$

$clk : W5$
$h\_sync : P19$
$v\_sync : R19$



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

5 Control Block

The description of the states of our project are as followed. In addition, the state transition diagram of the project is also attached below.

5.1 States of our project

1. Reset: Before starting the game, the bird is in the reset state and so is the game. If the X key is pressed during any phase of the game, it returns to its reset state. If the X key is not pressed, the game remains in the state it currently is unless the game ends due to collision with a tube.

2. Up: The bird will move in an upward direction if the W key is pressed. The bird will continue moving up until and unless the S key is pressed to change its direction downward, X key is pressed for the game to return in the initial stage or a tube is hit by the bird which automatically brings the game in initial position.

3. Down: The bird will move in a downward direction if the S key is pressed. The bird will continue moving in the downward direction if the S key is pressed. The bird will only change the direction if W or X key is pressed. Consequently, the game will also return to intial state if the bird hits a tube.

5.2 State Transition Diagram

Here is the state transition diagram:

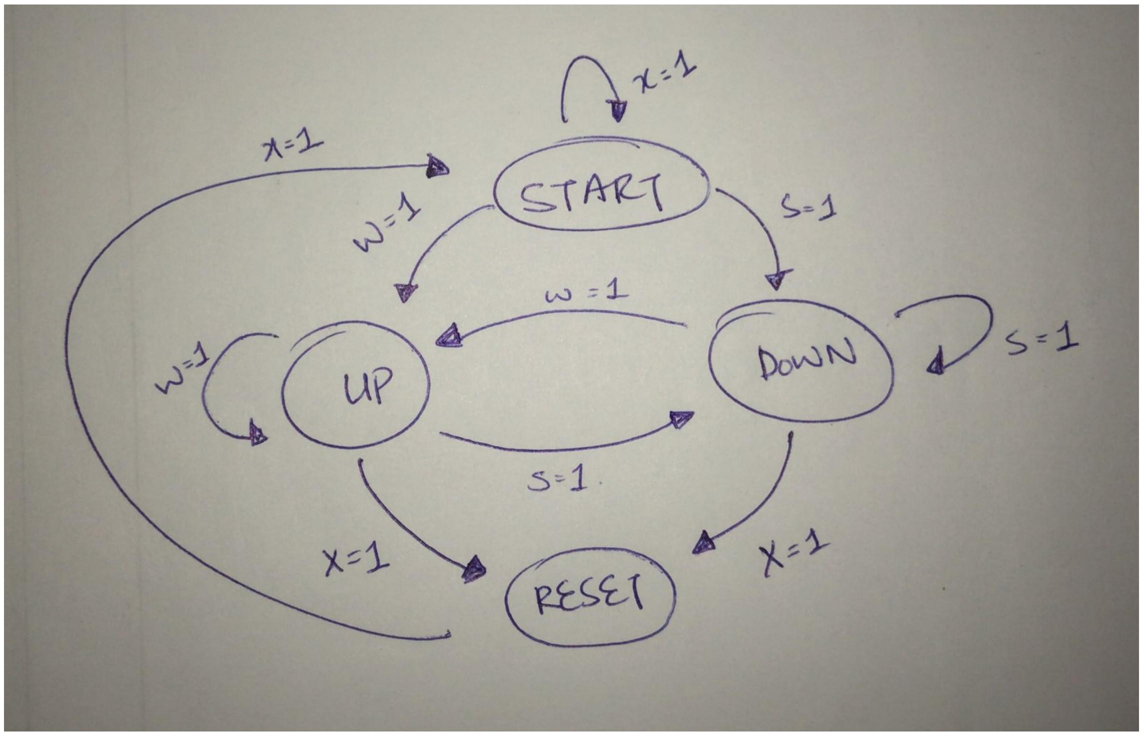


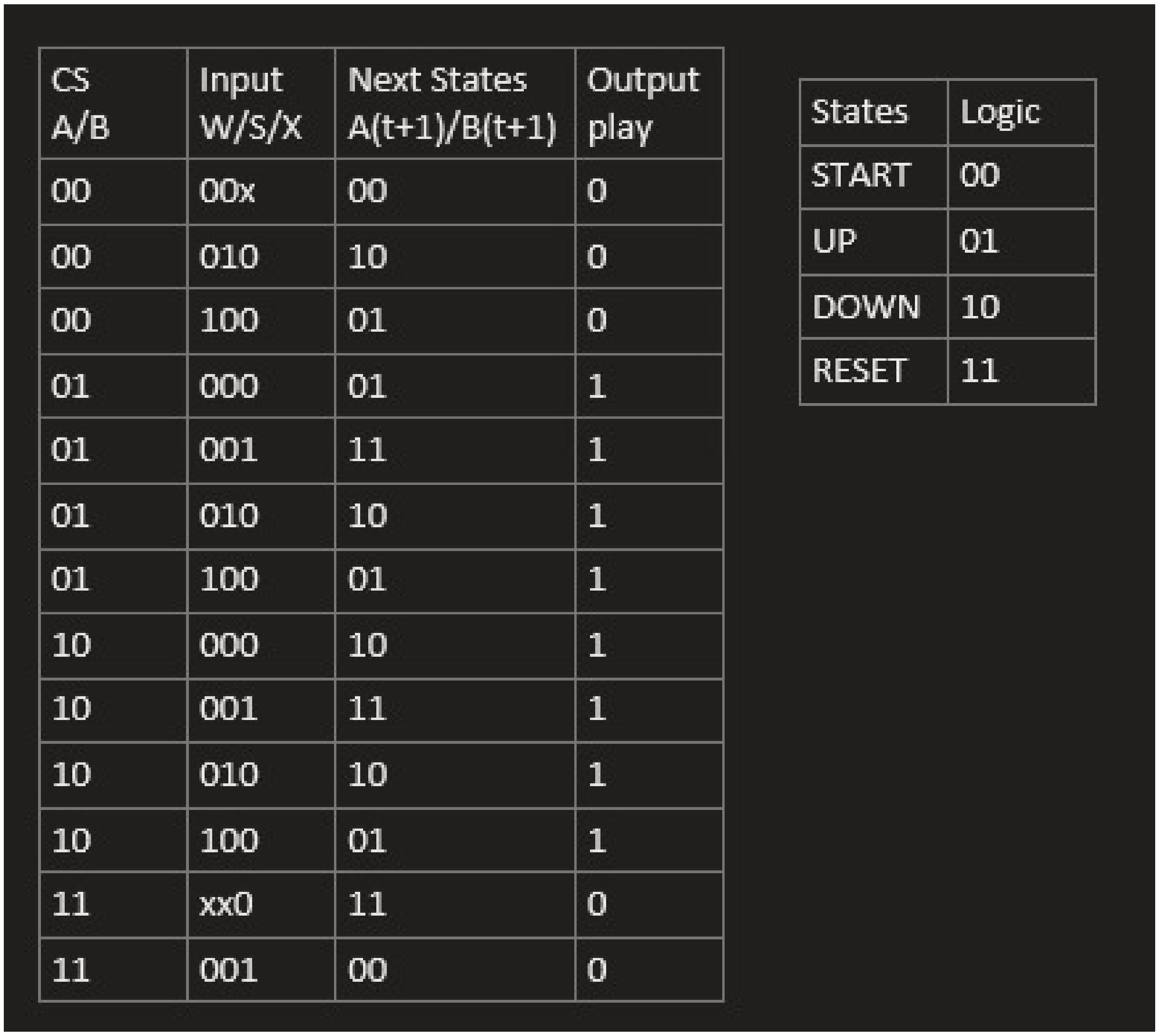
Figure 8: X = reset, W = up, S = down

6 FSM Design Procedures

6.1 Assumptions

1. We can have only one input at a given time.
2. The previous input will keep in motion until a new input is put in place through a new input by pressing any key.
3. The game ends if the bird keeps going up or down and eventually hits the top or the bottom of the screen.

6.2 State Table

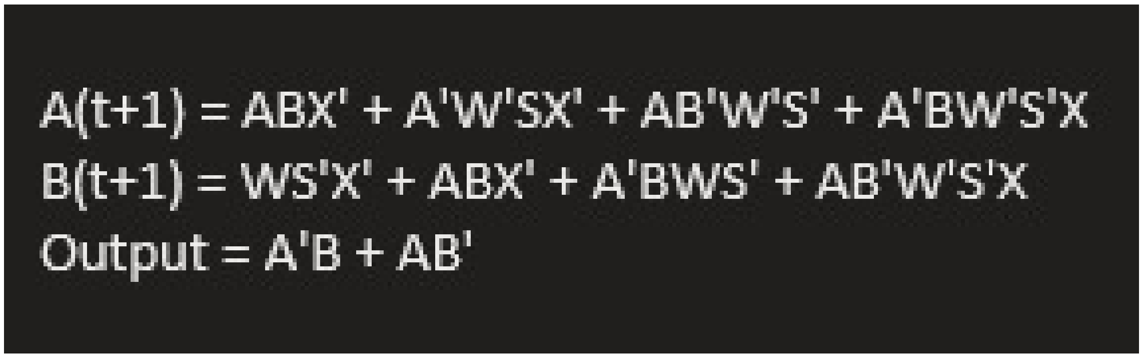


CS A/B	Input W/S/X	Next States A(t+1)/B(t+1)	Output play
00	00x	00	0
00	010	10	0
00	100	01	0
01	000	01	1
01	001	11	1
01	010	10	1
01	100	01	1
10	000	10	1
10	001	11	1
10	010	10	1
10	100	01	1
11	xx0	11	0
11	001	00	0

States	Logic
START	00
UP	01
DOWN	10
RESET	11

Figure 9: State table

6.3 State Equations



A(t+1) = ABX' + A'W'SX' + AB'W'S' + A'BW'S'X
B(t+1) = WS'X' + ABX' + A'BWS' + AB'W'S'X
Output = A'B + AB'

Figure 10: State Equations

6.4 Circuit Diagram

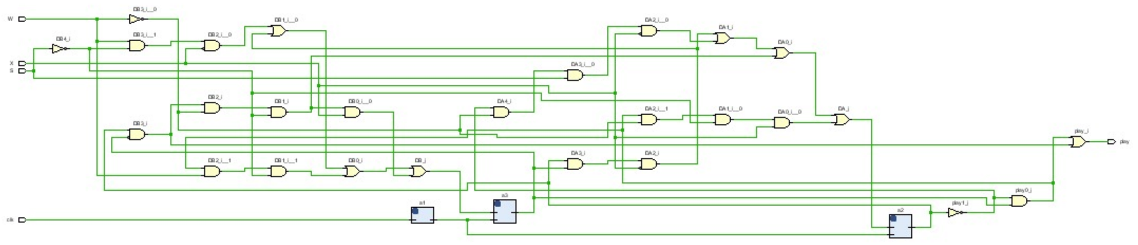


Figure 11: Circuit Diagram

6.5 Gate level Implementation of FSM

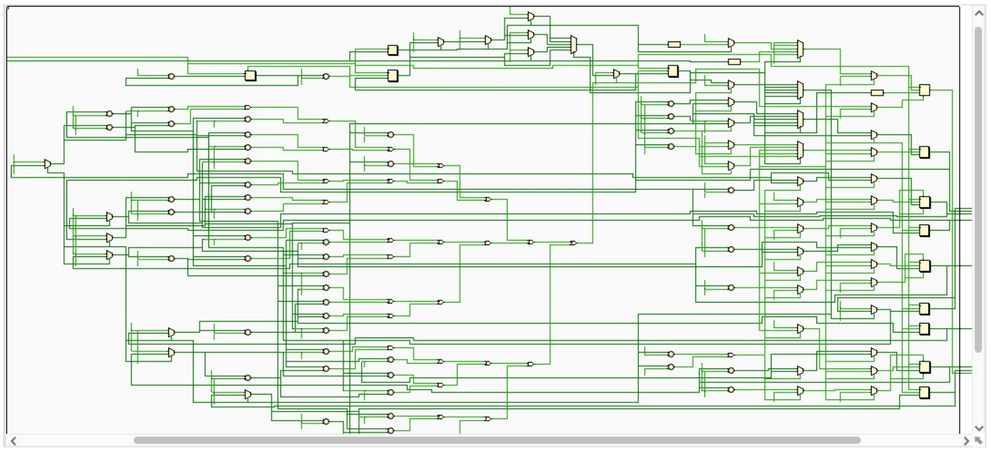


Figure 12: FSM on a gate level

7 Final Game

7.1 Elaborated Design

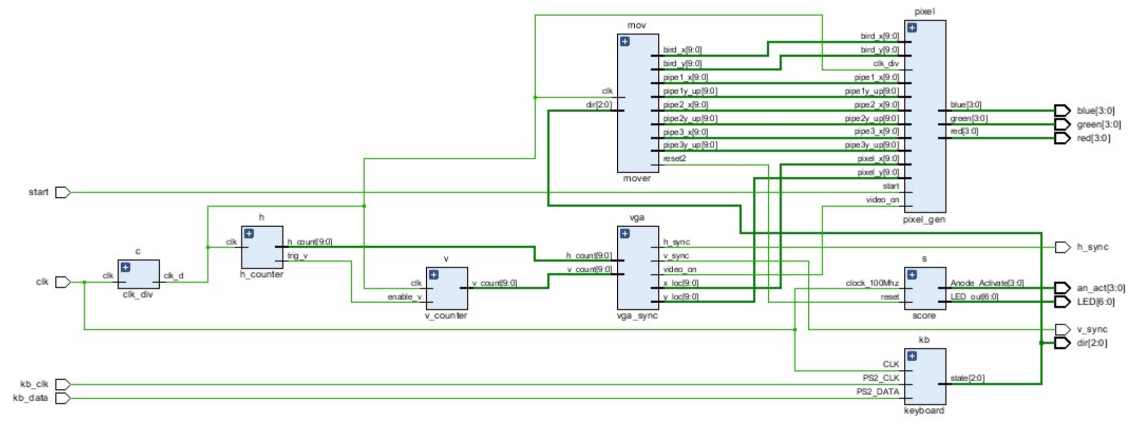


Figure 13: elaborated design for the game

7.2 Mover

Our entire game's function including the FSM is integrated in this module. This module updates the location of the bird and the pipe at each instance according to the state of the game which is then outputted to pixel gen which maps the pixels accordingly. Following is the code for the module:

```
 module mover(clk, dir, reset2, bird_x, bird_y, pipe1_x, pipe1y_up, pipe2_x, pipe2y_up, pipe3_x, pipe3y_
input clk;
//reset;
input [2:0] dir;
reg [2:0] c_state;
reg [2:0] state;
reg [2:0] next;
parameter UP = 3'b010;
parameter DOWN = 3'b100;
parameter START = 3'b000;
parameter RESET = 3'b111;
output reg reset2;
output reg [9:0] bird_x = 30;
output reg [9:0] bird_y = 230;

output reg [9:0] pipe1_x = 300;
output reg [9:0] pipe1y_up = 300;
// output reg [9:0] pipe1y_down = 380;

output reg [9:0] pipe2_x = 500;
output reg [9:0] pipe2y_up = 100;
// output reg [9:0] pipe2_down = 180;

output reg [9:0] pipe3_x = 100;
output reg [9:0] pipe3y_up = 200;
// output reg [9:0] pipe3y_down = 280;

reg [22:0] counter;
reg collision;

initial begin
  state = 000;
  collision = 0;
  counter = 0;
  reset2 <= 1;
end
always @ (posedge clk)
  begin
  c_state <= dir;
  counter <= counter + 1;
  if (counter >= 2499999)
    begin
      case (state)
      UP:
      begin
        if (bird_y > 0) begin
          bird_y <= bird_y - 10;
        end
        else begin
          state = RESET;
        end
        if (c_state == UP) begin
          next = UP;
        end
        if (c_state == DOWN) begin
          next = DOWN;
        end
        if (c_state == RESET) begin
          next = RESET;
        end
      end
      DOWN:
      begin
        if (bird_y < 460) begin
          bird_y <= bird_y + 10;
        end
        if (c_state == UP) begin
          next = UP;
        end
        if (c_state == DOWN) begin
          next = DOWN;
        end
        if (c_state == RESET) begin
          next = RESET;
        end
      end
      START:
      begin
        if (c_state == UP) begin
          reset2 <= 0;
          next = UP;
        end
        if (c_state == DOWN) begin
          reset2 <= 0;
          next = DOWN;
        end
        if (c_state == RESET) begin
          next = START;
        end
      end
      RESET:
      begin
        if (dir == RESET) begin
          next = START;
          reset2 <= 1;
        end
        end
      endcase
      if (state != RESET && state != START) begin
        if (pipe1_x < 1) begin
          pipe1_x <= 640;
        end
        else begin
          pipe1_x <= pipe1_x - 5;
        end
        if (pipe2_x < 1) begin
          pipe2_x <= 640;
        end
        else begin
          pipe2_x <= pipe2_x - 5;
```

```
          end
          if (pipe3_x < 1) begin
            pipe3_x <= 640;
          end
          else begin
            pipe3_x <= pipe3_x - 5;
          end
        end
        if (state == START) begin
          bird_x = 30;
          bird_y = 230;

          pipe1_x = 300;
          pipe1y_up = 300;

          pipe2_x = 500;
          pipe2y_up = 100;

          pipe3_x = 100;
          pipe3y_up = 200;
        end

        if (bird_y < 1 || bird_y > 459 ||
        ((((bird_x + 20) > pipe1_x && (bird_x + 20) < pipe1_x + 40) || ((bird_x + 20) == pipe1_x) || (bird
        ((((bird_x + 20) > pipe2_x && (bird_x + 20) < pipe2_x + 40) || ((bird_x + 20) == pipe2_x) || (bird
        ((((bird_x + 20) > pipe3_x && (bird_x + 20) < pipe3_x + 40) || ((bird_x + 20) == pipe3_x) || (bird
          next = START;
          reset2 <= 1;
        end
        // if (bird_y < 1 || bird_y > 479 ||
        // ((bird_y < pipe1y_up || bird_y > pipe1y_up+80) && (bird_x+20 > pipe1_x || bird_x+20 < pipe1_x+5
        // ((bird_y < pipe2y_up || bird_y > pipe2y_up+80) && (bird_x+20 > pipe2_x || bird_x+20 < pipe2_x+5
        // ((bird_y < pipe3y_up || bird_y > pipe3y_up+80) && (bird_x+20 > pipe3_x || bird_x+20 < pipe3_x+5
        //   next = RESET;
        //   reset2 <= 1;
        // end
        state = next;
        counter <= 0;
      end

    end
endmodule
```

7.3 Pixel Gen

We are defining the pixels for the bird and the pipes at each instance. Following is the code for the module:

```
   // Code your design here
module pixel_gen(

  input [9:0] pixel_x,
  input [9:0] pixel_y,

  input clk_div,
  input start,
  input video_on,
  input [9:0] bird_x,
  input [9:0] bird_y,

  input [9:0] pipe1_x,
  input [9:0] pipe1y_up,
  input [9:0] pipe2_x,
  input [9:0] pipe2y_up,
  input [9:0] pipe3_x,
  input [9:0] pipe3y_up,

  output reg [3:0] red = 0,
  output reg [3:0] blue = 0,
  output reg [3:0] green = 0
);

  reg w = 1'b1;
  reg [22:0] counter = 0;
  parameter UP = 3'b010;
  parameter DOWN = 3'b100;
  parameter START = 3'b000;
  parameter RESET = 3'b111;
  always @ (posedge clk_div && start)
    begin
      if ((pixel_y >= bird_y && pixel_y <= (bird_y + 20)) && (pixel_x >= bird_x && pixel_x < (bird_x + 20)
//        drawing the bird
        begin
          red <= 4'hf;
          blue <= 4'h0;
          green <= 4'h0;
        end
      else if (((pixel_x >= pipe1_x && pixel_x < (pipe1_x + 40)) && ((pixel_y >= (pipe1y_up + 80) && pixel
      ((pixel_x >= pipe2_x && pixel_x < (pipe2_x + 40)) && ((pixel_y >= 0 && pixel_y < (pipe2y_up)) || (pi
      ((pixel_x >= pipe3_x && pixel_x < (pipe3_x + 40)) && ((pixel_y >= (pipe3y_up + 80) && pixel_y < 480)
//        drawing the pipes
        begin
          red <= 4'h0;
          blue <= 4'h0;
          green <= 4'hf;
        end
       else
 //        painting the rest of the background white
         begin
           red <= video_on ? 4'h0 : 4'h0;
           green <= video_on ? 4'h0 : 4'h0;
           blue <= video_on ? 4'h0 : 4'h0;
         end
    end

endmodule
```

7.4 Score

We have added a score module which calculates the score of the user according to the time the user is successful in playing the game without losing. Following is the code for the module:

```
module score(
input clock_100Mhz, // 100 Mhz clock source on Basys 3 FPGA
 input reset, // reset
 output reg [3:0] Anode_Activate, // anode signals of the 7-segment LED display
 output reg [6:0] LED_out// cathode patterns of the 7-segment LED display
 );
 reg [26:0] one_second_counter; // counter for generating 1 second clock enable
 wire one_second_enable;// one second enable for counting numbers
 reg [15:0] displayed_number; // counting number to be displayed
 reg [3:0] LED_BCD;
 reg [19:0] refresh_counter; // 20-bit for creating 10.5ms refresh period or 380Hz refresh rate
          // the first 2 MSB bits for creating 4 LED-activating signals with 2.6ms digit period
 wire [1:0] LED_activating_counter;
              // count     0    ->  1  ->  2  ->  3
           // activates    LED1    LED2   LED3   LED4
          // and repeat
 always @(posedge clock_100Mhz or posedge reset)
 begin
     if(reset==1)
         one_second_counter <= 0;
     else begin
         if(one_second_counter>=99999999)
              one_second_counter <= 0;
```

```
        else
            one_second_counter <= one_second_counter + 1;
    end
end
assign one_second_enable = (one_second_counter==99999999)?1:0;
always @(posedge clock_100Mhz or posedge reset)
begin
    if(reset==1)
        displayed_number <= 0;
    else if(one_second_enable==1)
        displayed_number <= displayed_number + 2;
end
always @(posedge clock_100Mhz or posedge reset)
begin
    if(reset==1)
        refresh_counter <= 0;
    else
        refresh_counter <= refresh_counter + 1;
end
assign LED_activating_counter = refresh_counter[19:18];
// anode activating signals for 4 LEDs, digit period of 2.6ms
// decoder to generate anode signals
always @(*)
begin
    case(LED_activating_counter)
    2'b00: begin
        Anode_Activate = 4'b0111;
        // activate LED1 and Deactivate LED2, LED3, LED4
        LED_BCD = displayed_number/1000;
        // the first digit of the 16-bit number
          end
    2'b01: begin
        Anode_Activate = 4'b1011;
        // activate LED2 and Deactivate LED1, LED3, LED4
        LED_BCD = (displayed_number % 1000)/100;
        // the second digit of the 16-bit number
          end
    2'b10: begin
        Anode_Activate = 4'b1101;
        // activate LED3 and Deactivate LED2, LED1, LED4
        LED_BCD = ((displayed_number % 1000)%100)/10;
        // the third digit of the 16-bit number
            end
    2'b11: begin
        Anode_Activate = 4'b1110;
        // activate LED4 and Deactivate LED2, LED3, LED1
        LED_BCD = ((displayed_number % 1000)%100)%10;
        // the fourth digit of the 16-bit number
           end
    endcase
end
// Cathode patterns of the 7-segment LED display
always @(*)
begin
    case(LED_BCD)
    4'b0000: LED_out = 7'b0000001; // "0"
    4'b0001: LED_out = 7'b1001111; // "1"
    4'b0010: LED_out = 7'b0010010; // "2"
    4'b0011: LED_out = 7'b0000110; // "3"
    4'b0100: LED_out = 7'b1001100; // "4"
    4'b0101: LED_out = 7'b0100100; // "5"
    4'b0110: LED_out = 7'b0100000; // "6"
    4'b0111: LED_out = 7'b0001111; // "7"
    4'b1000: LED_out = 7'b0000000; // "8"
    4'b1001: LED_out = 7'b0000100; // "9"
    default: LED_out = 7'b0000001; // "0"
    endcase
end
endmodule
```

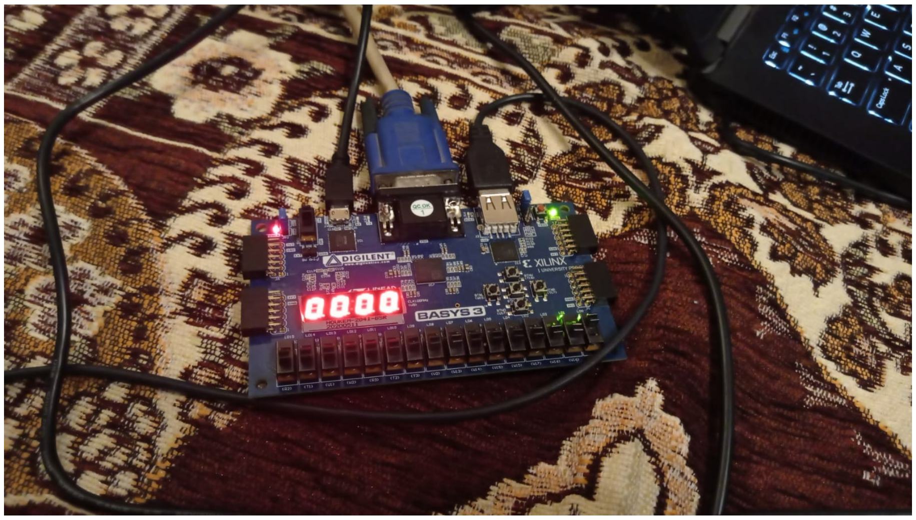


Figure 14: 7 Segment Display

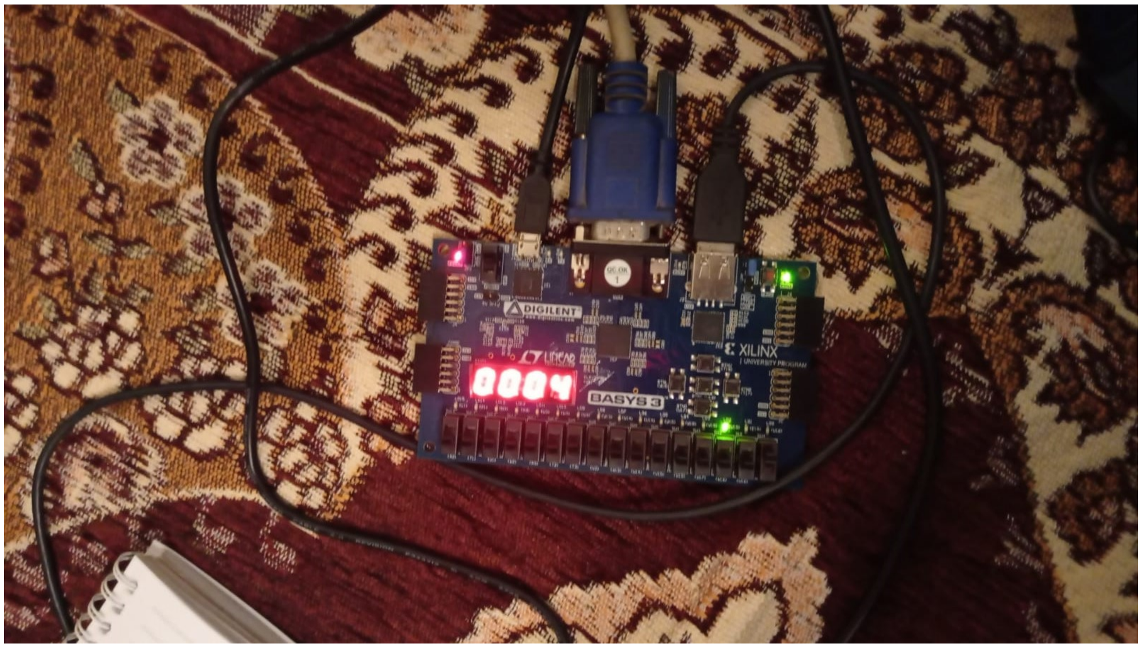


Figure 15: 7 Segment Display

8 Conclusion

The report presented is a detailed explanation to our very project, Flappy Bird, which is an FPGA emulation of the actual "Flappy Bird" game. Our report brings forward the ideas involved in making of the game.

The report introduces the audience of the user-flow diagram along with its division of blocks. Moreover, it gives a complete insight to the input, control and output blocks of the game made. With that said, lets briefly include what they comprise of as well.

The focal point of the input block makes us understand the Input Output (IO) pins involved, which in our case are keys from the basic keyboard. The bird moves with respect to the order assigned to it. Pressing 'W' would make the bird go up and 'S' to make it come back down.

The basis of the control block is to check the validity of the inputs provided and make sure the output is generated. Besides its usual function, it explains the states, in company with the state transition diagram.

The output block, is the main block considering that's what the user interacts with, explains the pixel division of the components stated in our project which are the bird and the tubes it has pass through. If at any point, the user is unable to get through the desired tube, the game resets and points are set back to zero.

9 References

Keyboard Input: `https://www.instructables.com/PS2-Keyboard-for-FPGA/`
Key scan codes and IO pins: `https://digilent.com/reference/programmable-logic/basys-3/start`
Reference paper: Gladyshev, P., Patel, A. (2004). Finite state machine approach to digital event reconstruction. Digital Investigation, 1(2), 130-149.

10 Github Repository

`https://github.com/princebiscuits/DLD-Project.git`