Indian Institute of Space Science and Technology

Thiruvananthapuram

B Tech Digital Electronics & VLSI Lab Project

A

Project Report on

Designing Tic-Tac-Toe(X-O) game using Verilog

Submitted by

B Tech Electronics and Communication Engineering 2018 batch

Kothadiya Princekumar (SC18B078) (Roll No.-15)
Akash Niladri Ganguly (SC18B072) (Roll No.-14)

18^{th} July 2020

Department of ECE(Avionics)

Contents

Abstract

Well Known Pen-and-paper game - Tic-Tac-Toe (X-O) can be converted into an interesting electronics game using Verilog Code on FPGA.The game can be between two players or between a player and a computer. This game has always been one of most go to games for kids in their pass time. Even adults find the game quite nostalgic. The investment for the original game is very negligible as it requires only paper and pen. Converting it to an electronic game can increase its craze more and will still remain a pass time game in this digital age as well/

Input (such as positions and player's turn) can be taken from the player through switches/Push buttons of FPGA board though Key-board Interface using PS-2 designis not fully completed.

Output can also be displayed on screen through VGA connector using Verilog coding of VGA Interface.

When the players play the game by pressing their corresponding button, a 2-bit value is stored into one of the nine positions registers (as per given input position by respective player) in the 3x3 grid like Xs/ Os in the real paper-and-pencil version.

Binary value 00 is stored into a position and showed blank on screen through VGA when none of the player played in that position. Similarly, *binary number 01 (X)* is the value to be stored when the player-1 played in the position and *binary number 10 (O)* is the value to be saved when the player-2 played in the position while *RED* and *GREEN* colour box is showed on screen through VGA connector respectively.

Similarly, Illegal moves, winner and draw of the game can be identified by the system through Verilog Code and shown on the side pannel of the screen by *the respective colours* for winner, *BLUE* for illegal moves and *YELLOW* for Draw.

Chapter 1

Introduction

Tic-tac-toe, noughts and crosses or Xs and Os is a most popular paper-and-pencil game for two players, X and O, who take turns marking the spaces in a 3x3 grid. Usually the game starts with move of the person who chose an X or a cross as his/her symbol followed by the player with 0 or nought. They then place their symbols one after the other in any one of nine defined places if it is not filled. The player who succeeds in placing three of their marks in a horizontal, vertical, or diagonal row is declared as the winner.

Figure 1.1: A completed tic-tac-toe game

1.1 Contribution -

Though we both planned together this project and worked together online in this current situation with the helped of given references, We tried to divide the work contribution based on few parameters :

1. **Prince (SC18B078)** : Mostly worked on Sequential Part of designing and TOP level modules and Simulation Aspects
   - Player Position (small parts of Chapter 2)
   - Clock divider
   - Datapath main module (Chapter 4)
   - FSM Controller
   - VGA interface (Chapter 3)
     - Horizontal Counter
     - Vertical Counter
     - VGA color
   - The TEST BENCH & Simulation part (Chapter 6)
2. **Akash (SC18B072)** : Mostly Worked on Combinational Part of designing and Lower Level Modules
   - Illegal move detector (Mostly Chapter-2)
   - NoSpace detector

Winner detector(Both modules)

Position decoder

Position Register

Key-Board Interface Study

Chapter 2

X-O game module

2.1 Players_positions-

This module is for assigning the given input position value to the particular player positions whenever respective player has to be allowed to play.

Verilog Code :

```
‘timescale 1ns / 1ps
module P_position(
input play,  // player enable signal
input play2, // player2 enable signal
input [3:0] x_position, // position given by users
output reg [3:0] player_position,player2_position // players position
);

always @ (play or play2) begin

case({play,play2})

2’b01 : player2_position<=x_position;
2’b10 : player_position<=x_position;
default :
begin player_position<=player_position;
player2_position<=player2_position;
end

endcase
end

endmodule
```

RTL Sychematic :

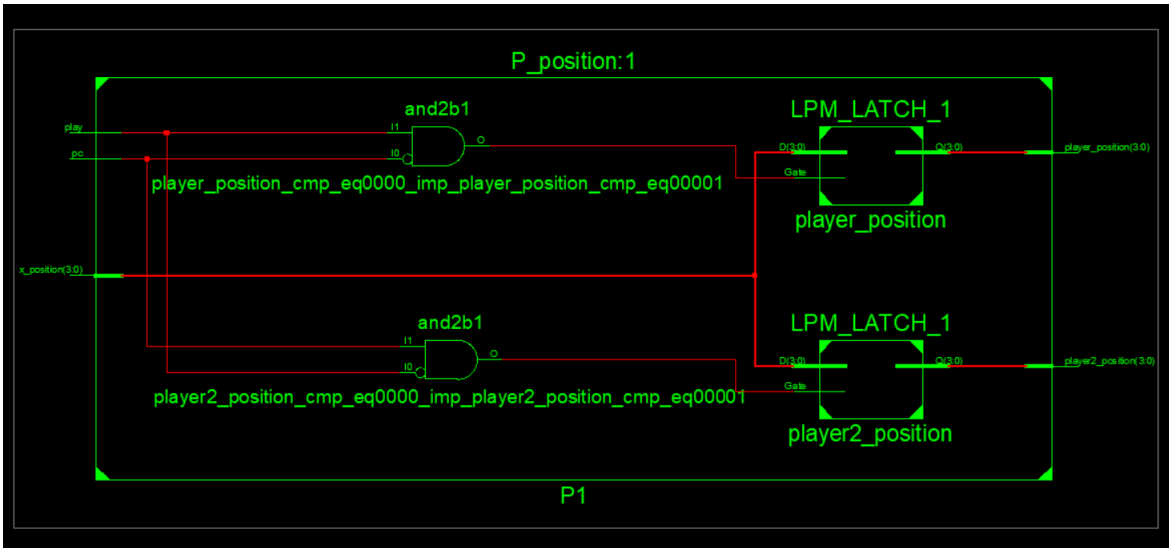


Figure 2.1: RTL Schematic of Player position

Explanation :

- As input will be given from FPGA(or PS2 key-board) it will be stored in *x_position*.
- when play is high (player-1 is playing) and play2 is low, *x_position* will be stored in the *player_position*.
- Similarly, *player*2*_position* can be changed with the *x_position* whenever play is low and play2 is high.
- And both *player_position* and *player*2*_position* will remains as it is for the other combination of the input of ***play*** *and* ***play2***.

2.2 Position decoder-

This module contains simple *4X16 Decoder* with enable signal used for decoding the respective player position into the 16 bit number whenever enable signal comes from controller.

Verilog Code :

```
‘timescale 1ns / 1ps
module position_decoder(
input[3:0] in, input enable, output wire [15:0] out_en);
reg[15:0] temp1;
assign out_en = (enable==1’b1)?temp1:16’d0; // enable is from FSM Controller
always @(*)
begin
case(in)
4’d0: temp1 <= 16’b0000000000000001;
4’d1: temp1 <= 16’b0000000000000010;
4’d2: temp1 <= 16’b0000000000000100;
4’d3: temp1 <= 16’b0000000000001000;
4’d4: temp1 <= 16’b0000000000010000;
```

```
4'd5: temp1 <= 16'b0000000000100000;
4'd6: temp1 <= 16'b0000000001000000;
4'd7: temp1 <= 16'b0000000010000000;
4'd8: temp1 <= 16'b0000000100000000;
4'd9: temp1 <= 16'b0000001000000000;  // Only needed till 9 because number of positions in the game is 9.
4'd10: temp1 <= 16'b0000010000000000;
4'd11: temp1 <= 16'b0000100000000000;
4'd12: temp1 <= 16'b0001000000000000;
4'd13: temp1 <= 16'b0010000000000000;
4'd14: temp1 <= 16'b0100000000000000;
4'd15: temp1 <= 16'b1000000000000000;
default: temp1 <= 16'b0000000000000001;
endcase
end
endmodule
```

RTL Sychematic :

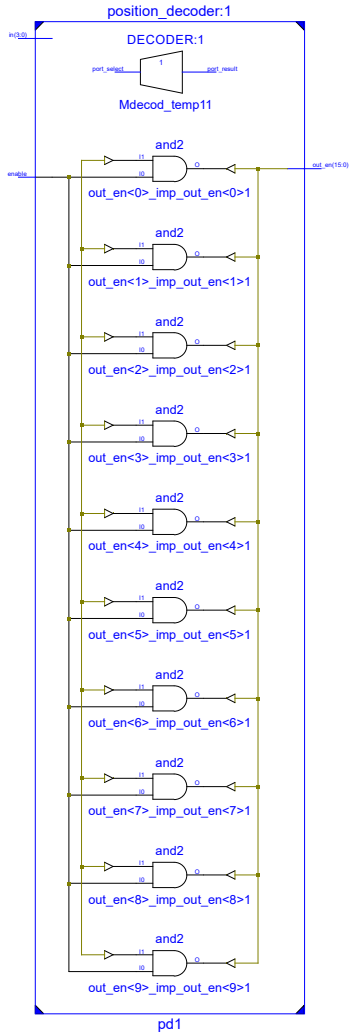


Figure 2.2: RTL Schematic of Position decoder

Explanation :

- Although it is providing the 16 bit, we need to consider till the input is 10 in decimal because there is only 10 positions in the game.
- So whenever the enable signal comes from *FSM Controller*, decoder will work and give decoded output for the respective input.
- while signal from the controller is low, decoder output will remain as zero.
- As, mentioned It will be used to decode the input position of the respective player.

2.3 Position register-

This module contains 9 register for the 9 position of the game as decribed and store the values 10, 01 and 00 for the Player-1, Player-2 and unoccupied positions respectively.

Verilog Code :

```
‘timescale 1ns / 1ps
module position_registers(
input clock, // clock of the game
input reset, // reset the game
input illegal_move, // need to stop when an illegal move is detected
input [9:1] PL2_en, // Player2 enable signals
input [9:1] PL_en, // Player enable signals
output reg[1:0] pos1,pos2,pos3,pos4,pos5,pos6,pos7,pos8,pos9// positions stored
);

// Position 1
always @(posedge clock or posedge reset) // Active high logic for reset
begin
if(reset)
pos1 <= 2’b00;
else begin
if(illegal_move==1’b1)
pos1 <= pos1;// keep previous position
else if(PL2_en[1]==1’b1)
pos1 <= 2’b10; // store player2 data
else if (PL_en[1]==1’b1)
pos1 <= 2’b01;// store player data
else
pos1 <= pos1;// keep previous position
end
end
// Position 2
always @(posedge clock or posedge reset) // Active high logic for reset
begin
if(reset)
pos2 <= 2’b00;
else begin
if(illegal_move==1’b1)
pos2 <= pos2;// keep previous position
else if(PL2_en[2]==1’b1)
pos2 <= 2’b10; // store player2 data
else if (PL_en[2]==1’b1)
```

```
pos2 <= 2'b01;// store player data
else
pos2 <= pos2;// keep previous position
end
end
// Position 3
always @(posedge clock or posedge reset)  // Active high logic for reset
begin
if(reset)
pos3 <= 2'b00;
else begin
if(illegal_move==1'b1)
pos3 <= pos3;// keep previous position
else if(PL2_en[3]==1'b1)
pos3 <= 2'b10; // store player2 data
else if (PL_en[3]==1'b1)
pos3 <= 2'b01;// store player data
else
pos3 <= pos3;// keep previous position
end
end
// Position 4
always @(posedge clock or posedge reset)  // Active high logic for reset
begin
if(reset)
pos4 <= 2'b00;
else begin
if(illegal_move==1'b1)
pos4 <= pos4;// keep previous position
else if(PL2_en[4]==1'b1)
pos4 <= 2'b10; // store player2 data
else if (PL_en[4]==1'b1)
pos4 <= 2'b01;// store player data
else
pos4 <= pos4;// keep previous position
end
end
// Position 5
always @(posedge clock or posedge reset) // Active high logic for reset
begin
if(reset)
pos5 <= 2'b00;
else begin
if(illegal_move==1'b1)
pos5 <= pos5;// keep previous position
else if(PL2_en[5]==1'b1)
pos5 <= 2'b10; // store player2 data
else if (PL_en[5]==1'b1)
pos5 <= 2'b01;// store player data
else
pos5 <= pos5;// keep previous position
end
end
// Position 6
always @(posedge clock or posedge reset)   // Active high logic for reset
begin
if(reset)
pos6 <= 2'b00;
else begin
```

```
if(illegal_move==1'b1)
pos6 <= pos6;// keep previous position
else if(PL2_en[6]==1'b1)
pos6 <= 2'b10; // store player2 data
else if (PL_en[6]==1'b1)
pos6 <= 2'b01;// store player data
else
pos6 <= pos6;// keep previous position
end
end
// Position 7
always @(posedge clock or posedge reset)  // Active high logic for reset
begin
if(reset)
pos7 <= 2'b00;
else begin
if(illegal_move==1'b1)
pos7 <= pos7;// keep previous position
else if(PL2_en[7]==1'b1)
pos7 <= 2'b10; // store player2 data
else if (PL_en[7]==1'b1)
pos7 <= 2'b01;// store player data
else
pos7 <= pos7;// keep previous position
end
end
// Position 8
always @(posedge clock or posedge reset)     // Active high logic for reset
begin
if(reset)
pos8 <= 2'b00;
else begin
if(illegal_move==1'b1)
pos8 <= pos8;// keep previous position
else if(PL2_en[8]==1'b1)
pos8 <= 2'b10; // store player2 data
else if (PL_en[8]==1'b1)
pos8 <= 2'b01;// store player data
else
pos8 <= pos8;// keep previous position
end
end
// Position 9
always @(posedge clock or posedge reset) // Active high logic for reset
begin
if(reset)
pos9 <= 2'b00;
else begin
if(illegal_move==1'b1)
pos9 <= pos9;// keep previous position
else if(PL2_en[9]==1'b1)
pos9 <= 2'b10; // store player2 data
else if (PL_en[9]==1'b1)
pos9 <= 2'b01;// store player data
else
pos9 <= pos9;// keep previous position
end
end
endmodule
```

RTL Sychematic :

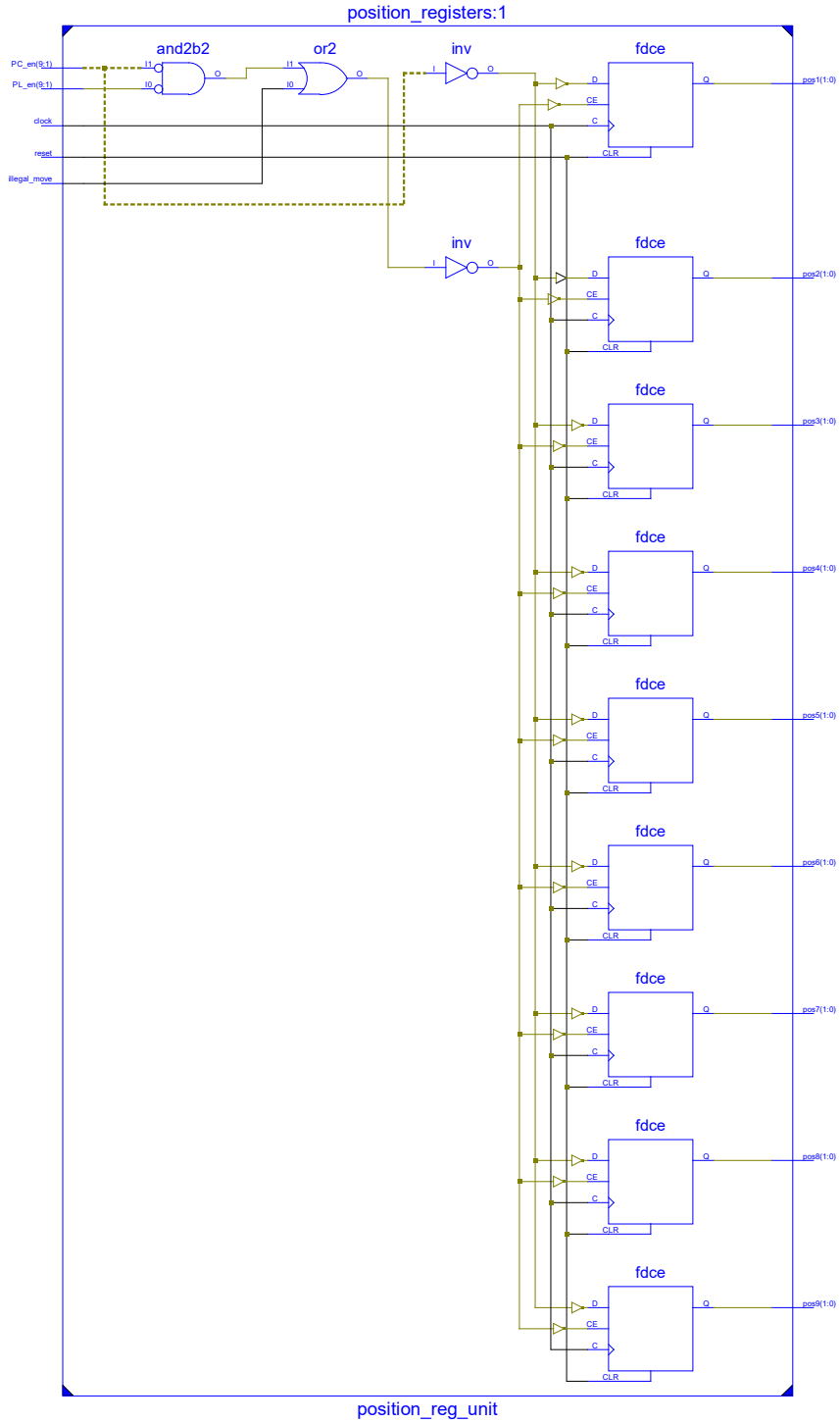


Figure 2.3: RTL Schematic of Position register

Explanation :

- It is the synchronous sequential design with clock to store the data in the register.
- As decribe earlier, 9 bit of decoded two input PL_en and PL2_en which will be get from the *position decoder module* can be used to store in respective positions for both players from available 9 position.
- If illegal move* is detected then registers value will be kept same as earlier rather than changing synchronously with clock.

2.4 Illegal move detector-

This module gives the illegal move signal high when any player try to access prefilled positions or both player try to play at same time and as we seen earlier illegal move detector will stop the playing till everything goes normal.

Verilog Code :

```
`timescale 1ns / 1ps
module illegal_move_detector(
input play,          // player-1 enable signal
input player2,       // player-2 enable signal
input [1:0] pos1,pos2,pos3,pos4,pos5,pos6,pos7,pos8,pos9, // position registers
input [9:1] PL2_en, PL_en, // decoded positions of both player
output wire illegal_move // illegal move signal
);
wire temp1,temp2,temp3,temp4,temp5,temp6,temp7,temp8,temp9;
wire temp11,temp12,temp13,temp14,temp15,temp16,temp17,temp18,temp19;
wire temp21,temp22;
// player : illegal moving
assign temp1 = (pos1[1] | pos1[0]) & PL_en[1];
assign temp2 = (pos2[1] | pos2[0]) & PL_en[2];
assign temp3 = (pos3[1] | pos3[0]) & PL_en[3];
assign temp4 = (pos4[1] | pos4[0]) & PL_en[4];
assign temp5 = (pos5[1] | pos5[0]) & PL_en[5];
assign temp6 = (pos6[1] | pos6[0]) & PL_en[6];
assign temp7 = (pos7[1] | pos7[0]) & PL_en[7];
assign temp8 = (pos8[1] | pos8[0]) & PL_en[8];
assign temp9 = (pos9[1] | pos9[0]) & PL_en[9];
// player2 : illegal moving
assign temp11 = (pos1[1] | pos1[0]) & PL2_en[1];
assign temp12 = (pos2[1] | pos2[0]) & PL2_en[2];
assign temp13 = (pos3[1] | pos3[0]) & PL2_en[3];
assign temp14 = (pos4[1] | pos4[0]) & PL2_en[4];
assign temp15 = (pos5[1] | pos5[0]) & PL2_en[5];
assign temp16 = (pos6[1] | pos6[0]) & PL2_en[6];
assign temp17 = (pos7[1] | pos7[0]) & PL2_en[7];
assign temp18 = (pos8[1] | pos8[0]) & PL2_en[8];
assign temp19 = (pos9[1] | pos9[0]) & PL2_en[9];
// intermediate signals
assign temp21 =(((((((temp1 | temp2) | temp3) | temp4) | temp5) | temp6) | temp7) | temp8) | temp9);
assign temp22 =(((((((temp11 | temp12) | temp13) | temp14) | temp15) | temp16) | temp17) | temp18) | temp19);
// output illegal move
assign illegal_move = temp21 | temp22 | (play & player2);
endmodule
```

*Illegal move can be detected by upcoming module Illegal_move_detector.

RTL Sychematic :

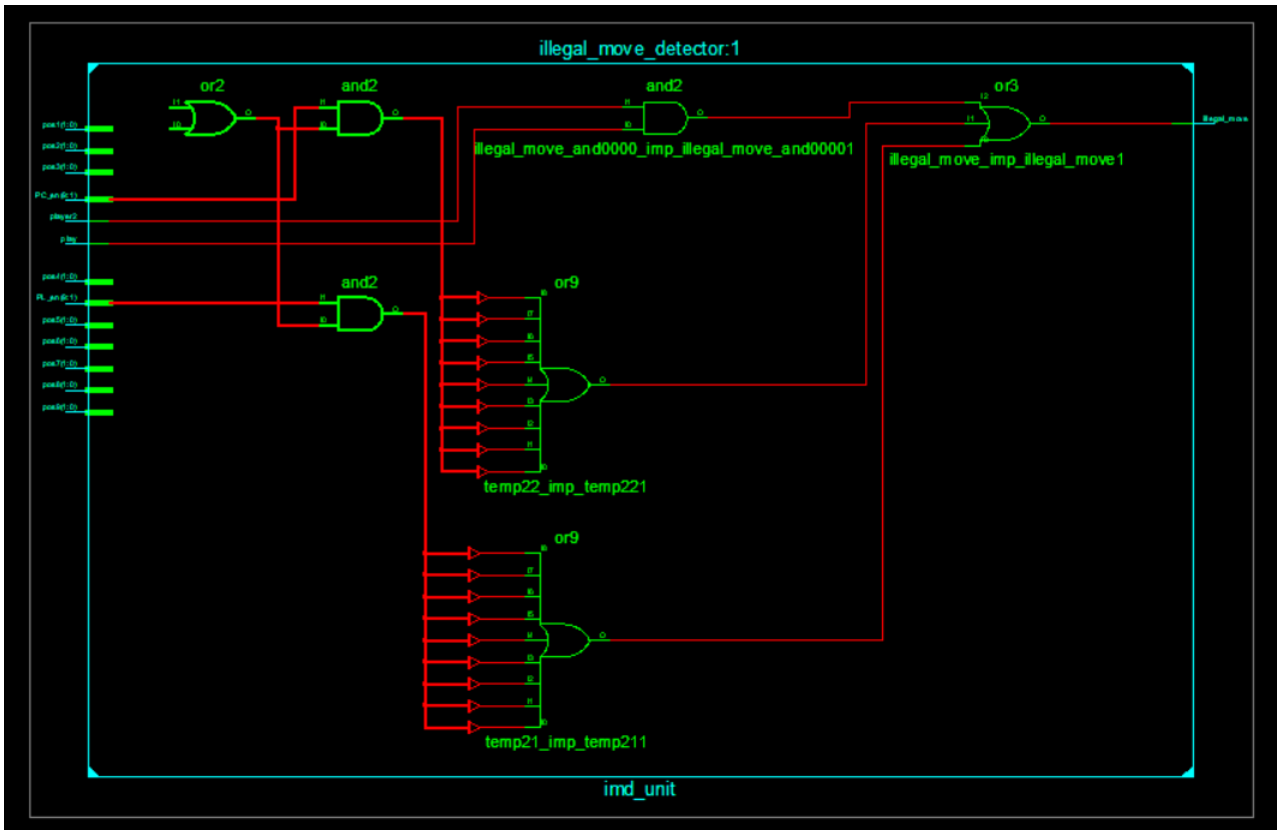


Figure 2.4: RTL Schematic of Illegal move detector

Explanation :

- It is purly combinational circuit as we have seen earlier.
- It is trying to check if person trys to play on pre-occupied position by taking bitwise *OR operation* of the 2 bit position register along with decoded player position which will be obtained from the input.

  Ex: suppose pos3 is pre occupied by player-1 then,

$$pos3 = 2'b01 \ (01 \ is \ for \ player - 1)$$

$$\implies pos3[0] \mid pos3[1] = 1$$

  Now if any player try to occupy it again then,

  Decoded input :

$$PL\_en[3] = 1$$

  or

$$PL2\_en[3] = 1$$

So,

$$temp3 = (pos3[1] \mid pos3[0]) \& PL\_en[3] = 1$$

or

$$temp13 = (pos3[1] \mid pos3[0]) \& PL2\_en[3] = 1$$

So, At the end,

$$illegal\_move = 1$$

- Similarly, when both the play and player2 are high at the same time (both player try to play at same time) illegal_move variable is high from the equation

$$illegal\_move = temp21 \mid temp22 \mid \textbf{(play \& player2)} = 1$$

2.5 NoSpace detector-

This module gives the nospace signal high when game is draw or when all the positions are occupied but no one wins till then.

Verilog Code :

```
‘timescale 1ns / 1ps
module nospace_detector(
input [1:0] pos1,pos2,pos3,pos4,pos5,pos6,pos7,pos8,pos9, // Positions register
output wire no_space    // no space signal
);
wire temp1,temp2,temp3,temp4,temp5,temp6,temp7,temp8,temp9;
// detect no more space
assign temp1 = pos1[1] | pos1[0];
assign temp2 = pos2[1] | pos2[0];
assign temp3 = pos3[1] | pos3[0];
assign temp4 = pos4[1] | pos4[0];
assign temp5 = pos5[1] | pos5[0];
assign temp6 = pos6[1] | pos6[0];
assign temp7 = pos7[1] | pos7[0];
assign temp8 = pos8[1] | pos8[0];
assign temp9 = pos9[1] | pos9[0];
// output
assign no_space =(((((((temp1 & temp2) & temp3) & temp4) & temp5) & temp6) & temp7) & temp8) & temp9); //
    same as bit-wise and operator
endmodule
```

RTL Sychematic :

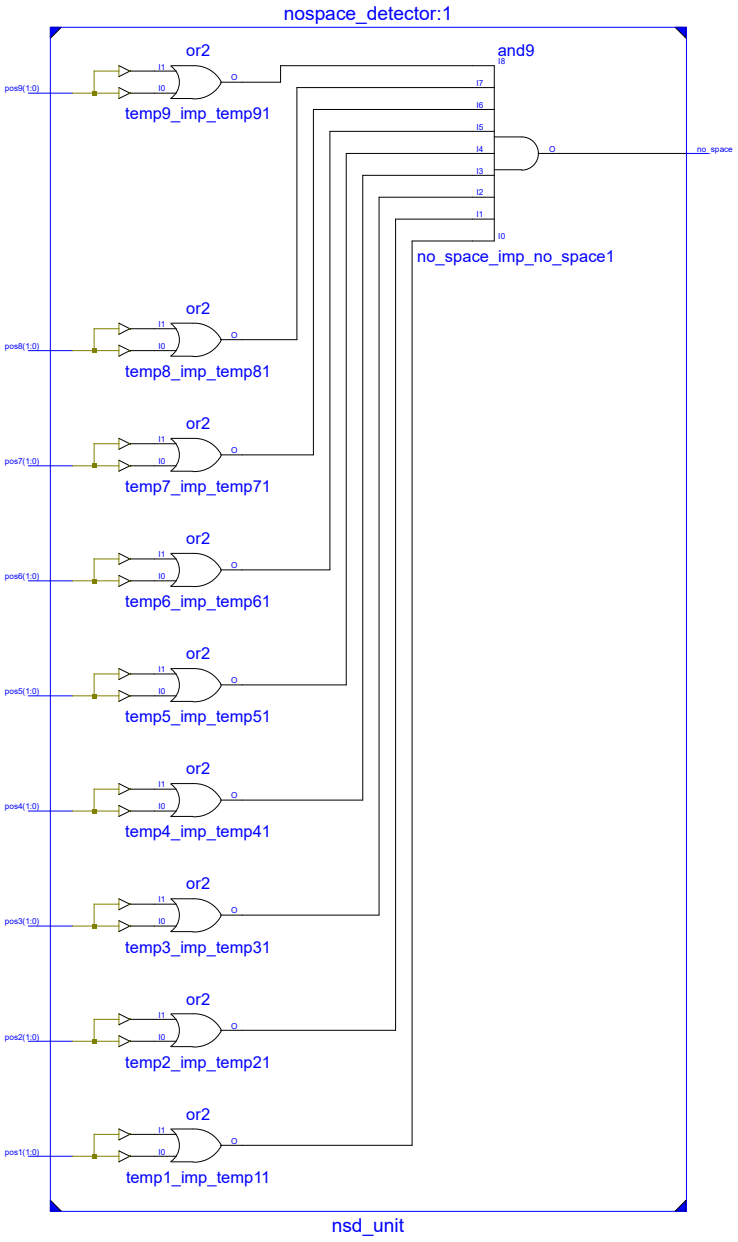


Figure 2.5: RTL Schematic of nospace detector

Explanation :

- It is purly combinational circuit as we have seen earlier.
- It is trying to check if all the positions are occipied but winner is not detected till that (basically draw condition) by bitwise *OR operator*.

Ex: suppose all positions are occupied except pos3 then,

Any position is having 10 or 01 bits depending on the player who occupied that position but pos3 is having 00 bit because it is not occupied till now.

So,

$$pos3 = 2'b00$$

(all others are 01 or 10)

$$\implies pos3[0] \mid pos3[1] = 0$$

and

$$posN[0] \mid posN[1] = 1$$

where, N = 0 to 9 except 3.

So, $temp3 = 0$ and $tempN = 0$

So, At the end after taking bitwise *AND operation* of all temp variable,

$$no\_space = 0$$

Now, if pos3 is also occupied then,
$temp3 = 1$ along with all other temp variables.
and at the end,

$$no\_space = 1$$

2.6 Winner detector (Inner module)-

This module gives the winner player code (01 or 10) and winner goes high if anyone having same three positions.

Verilog Code :

```
`timescale 1ns / 1ps
module winner_detect_3( // inner module of winner detector
input [1:0] pos0,pos1,pos2, // 3 positions to check the winner
output wire winner,      // 1 bit winner signal
output wire [1:0]who     // 2 bit number of player who wins
);

wire [1:0] temp0,temp1,temp2;
wire temp3;
assign temp0[1] = !(pos0[1]^pos1[1]); // for checking if both bit are same by XNOR
assign temp0[0] = !(pos0[0]^pos1[0]);
assign temp1[1] = !(pos2[1]^pos1[1]);
assign temp1[0] = !(pos2[0]^pos1[0]);
assign temp2[1] = temp0[1] & temp1[1]; // for checking both temp variable
assign temp2[0] = temp0[0] & temp1[0];
assign temp3 = pos0[1] | pos0[0];   // to check non-empty position
// winner if 3 positions are similar and should be 01 or 10
```

```
assign winner = temp3 & temp2[1] & temp2[0];
// determine who the winner is
assign who[1] = winner & pos0[1];
assign who[0] = winner & pos0[0];

endmodule
```

RTL Sychematic :

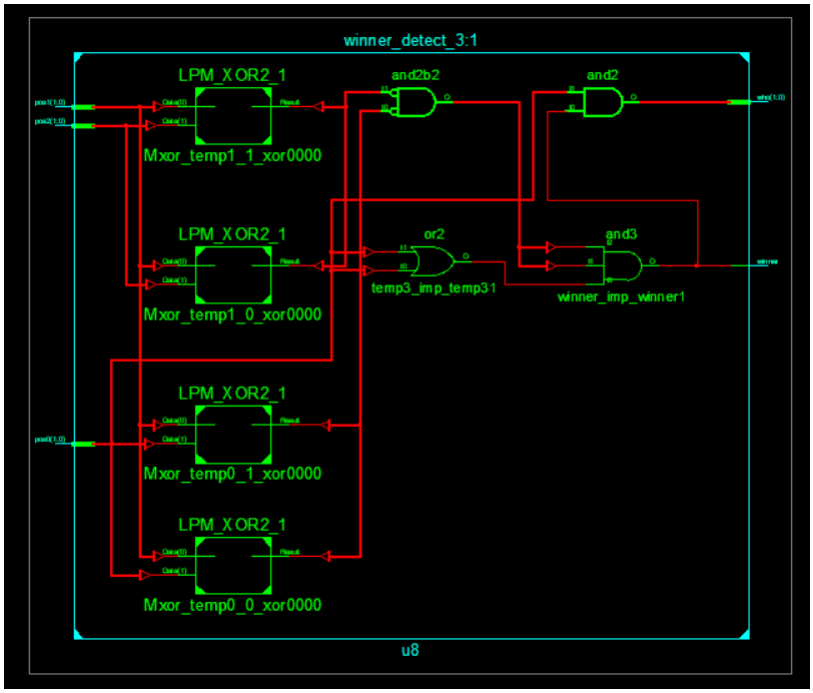


Figure 2.6: RTL Schematic of winner detector (Inner module)

Explanation :

- It is purly combinational circuit as we have seen earlier.
- It is trying to check if 3 given positions are occipied by players by *XNOR operator*.

  Ex: suppose all 3 given positions are occupied by player-1,

  So,

  $$posN = 2'b01$$

  where, N=0,1,2 (given positions).

  then, temp0 and temp1 will be 2'b11 by the XOR operation of respective position.

  $$\implies temp0 = 2'b11 \ \& \ temp1 = 2'b11$$

And 1 bit variable temp3 will ensure that positions are occupied like,

$$temp3 = pos0[1] \mid pos0[0] = 1 \ (because\ 01\ is\ stored\ in\ given\ positions)$$

So, At the end by after *AND operation*,

$$winner = 1$$

And *variable who* will denote the winner player's number (10) if $winner = 1$.

$$who[1] = winner\ \&\ pos0[1] = 1$$

and

$$who[0] = winner\ \&\ pos0[0] = 0$$

2.7 Winner detector (Overall module)-

This module gives the winner player code (01 or 10) and winner goes high if anyone having atleast one winning possible combination.

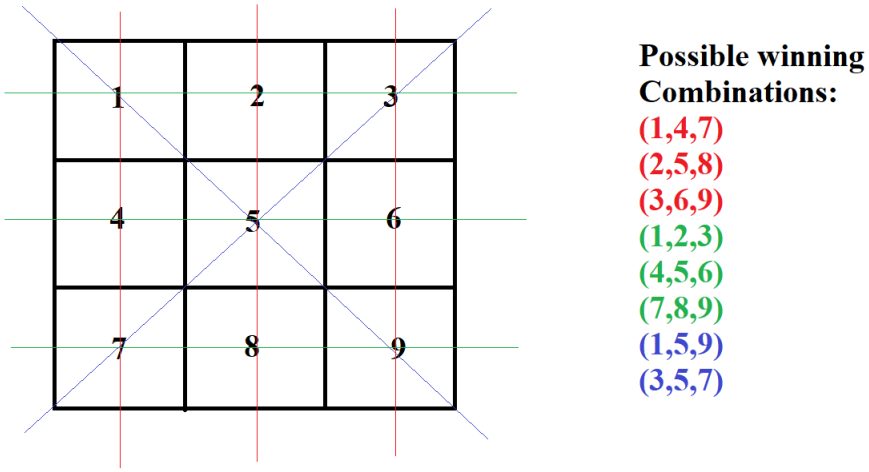


Figure 2.7: Possible combination of winning

Verilog Code :

```
`timescale 1ns / 1ps
module winner_detector(
input [1:0] pos1,pos2,pos3,pos4,pos5,pos6,pos7,pos8,pos9, // position registers
output wire winner, // winner variable
output wire [1:0]who // winner player number
);

wire win1,win2,win3,win4,win5,win6,win7,win8; // for checking all 8 winning combinations
```

```
wire [1:0] who1,who2,who3,who4,who5,who6,who7,who8; //for all 8 winning combinations
winner_detect_3 u1(pos1,pos2,pos3,win1,who1);// (1,2,3);
winner_detect_3 u2(pos4,pos5,pos6,win2,who2);// (4,5,6);
winner_detect_3 u3(pos7,pos8,pos9,win3,who3);// (7,8,9);
winner_detect_3 u4(pos1,pos4,pos7,win4,who4);// (1,4,7);
winner_detect_3 u5(pos2,pos5,pos8,win5,who5);// (2,5,8);
winner_detect_3 u6(pos3,pos6,pos9,win6,who6);// (3,6,9);
winner_detect_3 u7(pos1,pos5,pos9,win7,who7);// (1,5,9);
winner_detect_3 u8(pos3,pos5,pos6,win8,who8);// (3,5,7);
assign winner = (((((((win1 | win2) | win3) | win4) | win5) | win6) | win7) | win8); // if any winning
    combination works
assign who = (((((((who1 | who2) | who3) | who4) | who5) | who6) | who7) | who8);
endmodule
```

RTL Sychematic :

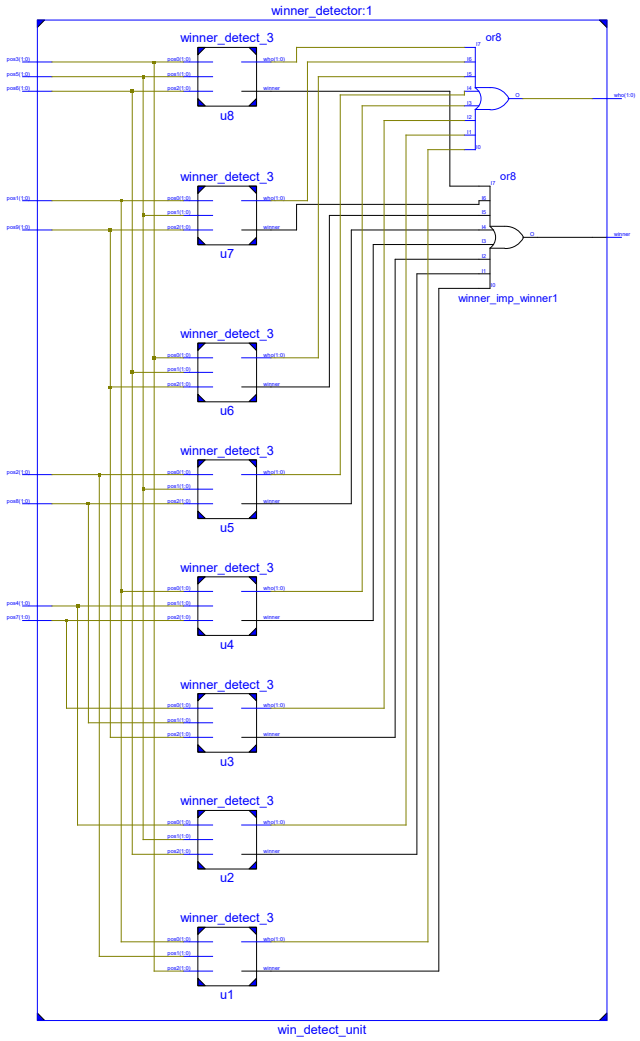


Figure 2.8: RTL Schematic of winner detector (Overall module)

Explanation :

- It is purly combinational circuit as we have seen earlier.
- It is Overall winning module which is using inner module for each and every 8 possible winnig combination.
- And at the final stage, by taking *OR operation* of the all 8 possible combination give the output and consider atleast one combination to win.

2.8 Clock divider-

Clock divider wil be used to get 25MHz for VGA connector[†] from 50MHz in built clock in FPGA.

Verilog Code :

```
‘timescale 1ns / 1ps
module clk_divider(
input clk_i, // input clock on FPGA
output clk_o // output clock after dividing the input clock by divisor
);
reg[7:0] counter=8’d0;
parameter DIVISOR = 8’d2;

always @(posedge clk_i)
begin
counter <= counter + 8’d1;
if(counter>=(DIVISOR-1))
counter <= 8’d0;
end
assign clk_o = (counter<DIVISOR/2)?1’b0:1’b1;

endmodule
```

[†]VGA connector specificaions will be upcoming

RTL Sychematic :

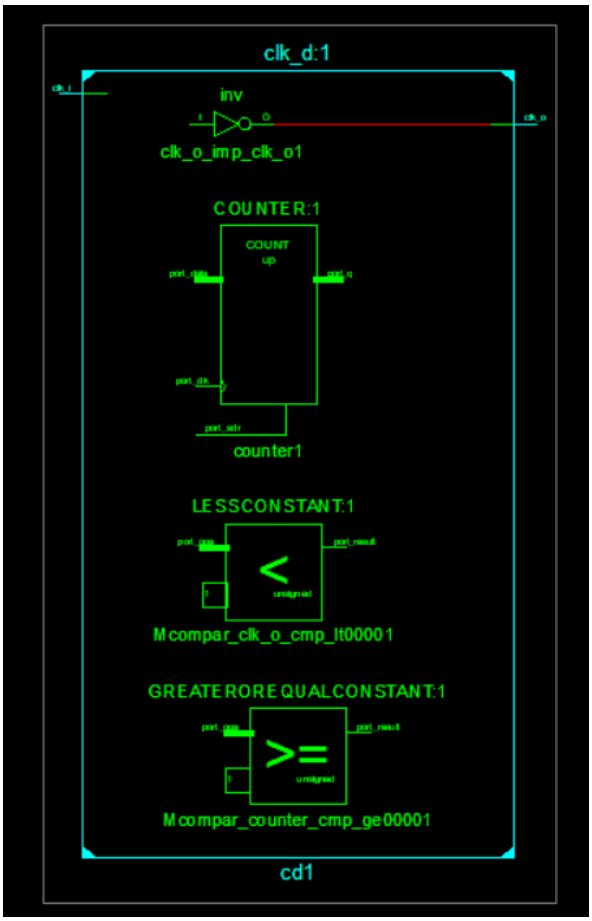


Figure 2.9: RTL Schematic of Clock divider

Explanation :

- It will be used to divide the clock frequency which will be obtained by internal clock of 50MHz from FPGA.
- Complete model is synchronized with the single clock frequency of 25 MHz.

Chapter 3

VGA Interface module

3.1 Introduction -

VGA interfacing is used when output is need to be displayed on the screen through VGA connector and in built VGA port on the FPGA Board.

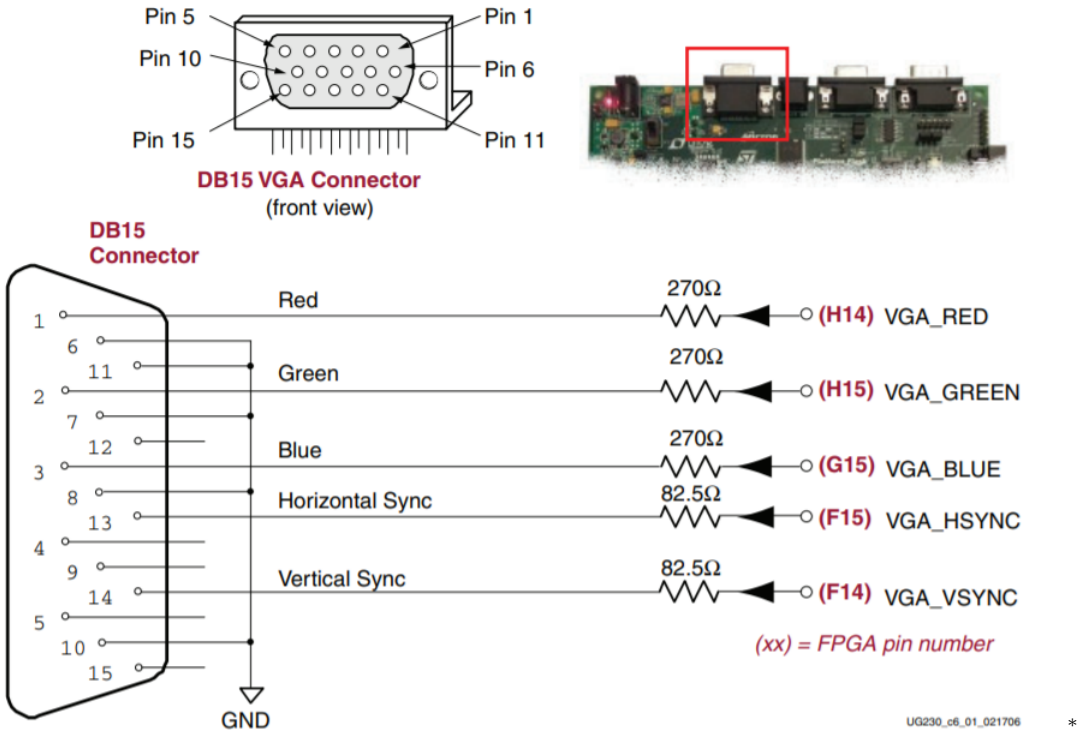


Figure 3.1: VGA port and signals

As shown earlier hsync and vsync port can be obtained from Horizontal Counter and Vertical Counter respecticely and RED, GREEN, BLUE (RGB) are the values of color for particular pixel(cell) made by horizontal and vertical position on the screen.

On the 640X480 display,

- Horizontal counter : counts from 0 to 799 for each vertical counter.
- Vertical counter : counts from 0 to 520.

Table 6-2: **640x480 Mode VGA Timing**

Symbol	Parameter	Vertical Sync			Horizontal Sync	
		Time	Clocks	Lines	Time	Clocks
T_S	Sync pulse time	16.7 ms	416,800	521	32 µs	800
T_{DISP}	Display time	15.36 ms	384,000	480	25.6 µs	640
T_{PW}	Pulse width	64 µs	1,600	2	3.84 µs	96
T_{FP}	Front porch	320 µs	8,000	10	640 ns	16
T_{BP}	Back porch	928 µs	23,200	29	1.92 µs	48

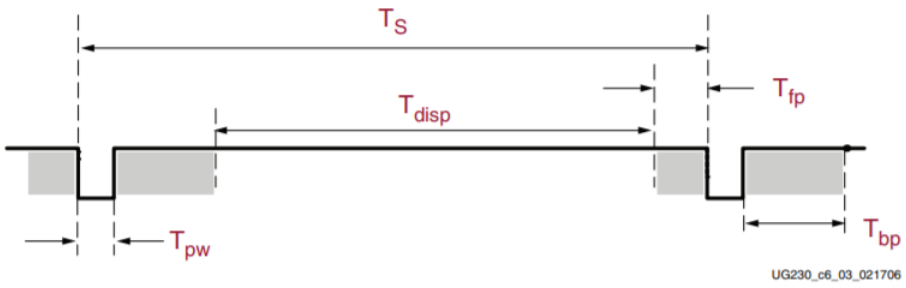


Figure 3.2: VGA Control and timing diagram

- Pulse width time : Synchronous signal will be enable for pulse width time.

Like this, Counter will count through out the screen on every pixel.

$totalpixel = 800 * 521 = 416800$

$clock\_frequency = 25MHz$

$\implies clock\_period = \frac{1}{25M} = 40ns$

40 ns is the time taken to count one pixel.

total time taken to count every pixel once :

$$T = (40ns) * 416800 = 16.672ms$$

Hense, **There should be time gap of** $\boxed{\textbf{T=16.672 ms}}$ **between any changing input signal.**

3.2 Horizontal Count-

Verilog Code :

```
‘timescale 1ns / 1ps
module H_count(
input clk_25, // clock signal of 25MHz frequency
output reg vcount_signal=0, // to activate the v_count
output reg [15:0] hcount=0 // Horizontal counter
);
```

```
always @ (posedge clk_25) begin
if (hcount < 799) begin // H_count counts from 0 to 799
hcount <= hcount + 1;
vcount_signal <= 0;
end
else if (hcount == 799)
begin
hcount <= 0;
vcount_signal <= 1;   // v_count counts 1 when H_count completes the counts
end
end
endmodule
```

RTL Sychematic :

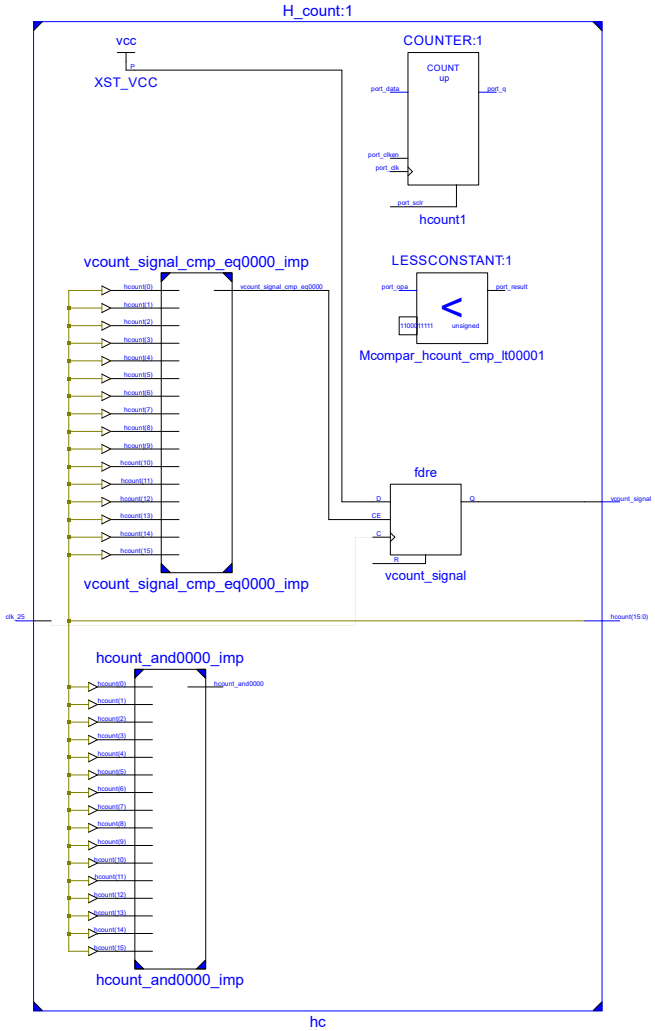


Figure 3.3: RTL Schematic of Horizontal Counter

Explanation :

- It counts from 0 to 799 and generate v_count signal to counts 1 unit in vertical position when it arrives at 799.

3.3 Vertical Count-

As discribed earlier V_count will count from 0 to 520.

Verilog Code :

```
‘timescale 1ns / 1ps
module V_count(
input clk_25,   // clock signal of 25MHz frequency
input vcount_signal,     // Vertical counter enable signal
output reg [15:0] vcount=0 // Vertical counter
);

always @ (posedge clk_25) begin
if (vcount_signal==1’b1) begin
if (vcount < 520) vcount <= vcount + 1;
else  vcount <= 0;
end
end
endmodule
```

RTL Sychematic :

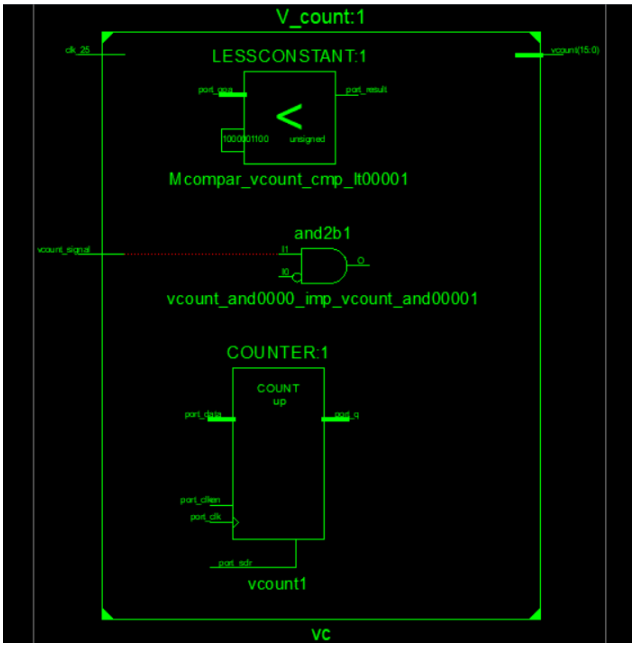


Figure 3.4: RTL Schematic of Vertical Counter

Explanation :

- It only counts one(1) unit whenever v_count signal goes high starting from 0 to 520.
- for each value of v_count, H_count will count from 0 to 799.

3.4 VGA controller-

This module take input as Horizontal counter, Vertical counter, positions and few other variables like illegal move, noSpace, winner etc. and give the RED, GREEN and BLUE values for each and every pixel counted by both counter.

As shown earlier, display pixels ranges are :

- Horizontal pixels : 144 to 784 (640 pixels)
- Vertical pixels : 31 to 511 (480 pixels)
- Horizontal sync : 0 to 96
- Vertical sync : 0 to 2

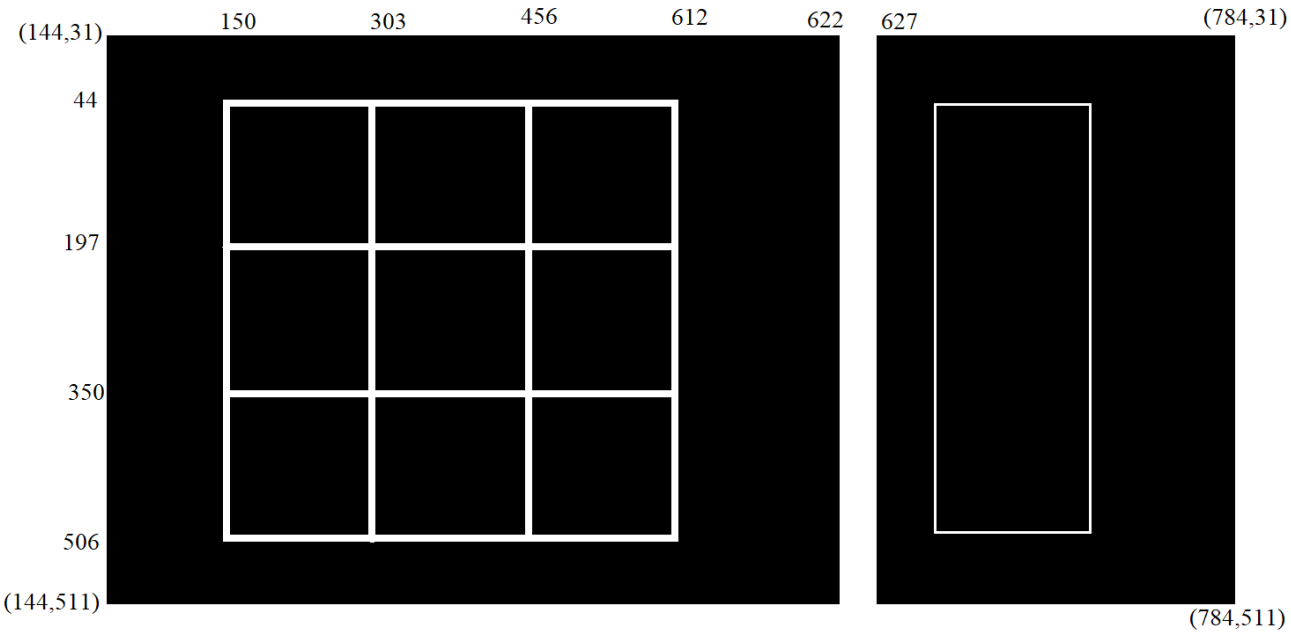


Figure 3.5: Screen display design

Verilog Code :

```
`timescale 1ns / 1ps
module VGA_colour(
input [17:0] pos,   // all position registers
input illegal_move, // illegal move signal
input no_space,     // no space signal
input [1:0] who,    // winner player number
```

```
input [15:0] H_counter, // Horizontal counter
input [15:0] V_counter, // vertical counter
output hsync,
output vsync,
output reg [2:0] red,   // RED
output reg [2:0] green, // GREEN
output reg [1:0] blue   // BLUE
);

wire [1:0] pos1,pos2,pos3,
pos4,pos5,pos6,pos7,pos8,pos9; // position registers extractions

assign pos1 = pos[17:16];
assign pos2 = pos[15:14];
assign pos3 = pos[13:12];
assign pos4 = pos[11:10];
assign pos5 = pos[9:8];
assign pos6 = pos[7:6];
assign pos7 = pos[5:4];
assign pos8 = pos[3:2];
assign pos9 = pos[1:0];

assign hsync = (H_counter < 96 ) ? 1 : 0;
assign vsync = (V_counter < 2 ) ? 1 : 0;

always @ (*) begin
if ((H_counter > 143 && H_counter < 784 && V_counter > 31 && V_counter < 511 )) begin // Display pixel ranges

if  (((H_counter > 150 && H_counter < 153) || (H_counter > 303 && H_counter < 306) || (H_counter > 456 &&
    H_counter < 459) || (H_counter > 609 && H_counter < 612)) && V_counter > 44 && V_counter < 506) begin //
    Board outline pixels
red<=3'b111;
green<=3'b111;
blue<=2'b11;
end

else if (((H_counter > 153 && H_counter < 303) || (H_counter > 306 && H_counter < 456) || (H_counter > 459 &&
    H_counter < 609)) && ((V_counter > 44 && V_counter < 47) || (V_counter > 197 && V_counter < 200) ||
    (V_counter > 350 && V_counter < 353) || (V_counter > 503 && V_counter < 506) )) begin // Board outline
    pixels
red<=3'b111;
green<=3'b111;
blue<=2'b11;
end

else if ((H_counter > 622 && H_counter < 627)) begin      // Side pannel pixels
red<=3'b111;
green<=3'b111;
blue<=2'b11;
end

// pos1
else if(H_counter > 168 && H_counter < 288 && V_counter > 62 && V_counter < 182) begin
case (pos1)

2'b01:begin         // If player-1 occupied position - RED
red<=3'b111;
green<=3'b000;
blue<=2'b00;
```

```
end

2'b10:begin        // If player-2 occupied position - GREEN
red<=3'b000;
green<=3'b111;
blue<=2'b00;
end

default:begin    // If position is non occupied - BLACK
red<=3'b000;
green<=3'b000;
blue<=2'b00;
end

endcase
end

//pos2
else if(H_counter > 321 && H_counter < 441 && V_counter > 62 && V_counter < 182) begin
case (pos2)

2'b01:begin
red<=3'b111;
green<=3'b000;
blue<=2'b00;
end

2'b10:begin
red<=3'b000;
green<=3'b111;
blue<=2'b00;
end

default:begin
red<=3'b000;
green<=3'b000;
blue<=2'b00;
end

endcase
end

//pos3
else if(H_counter > 474 && H_counter < 594 && V_counter > 62 && V_counter < 182) begin
case (pos3)

2'b01:begin
red<=3'b111;
green<=3'b000;
blue<=2'b00;
end

2'b10:begin
red<=3'b000;
green<=3'b111;
blue<=2'b00;
end

default:begin
```

```
red<=3'b000;
green<=3'b000;
blue<=2'b00;
end

endcase
end

//pos4
else if(H_counter > 168 && H_counter < 288 && V_counter > 215 && V_counter < 335) begin
case (pos4)

2'b01:begin
red<=3'b111;
green<=3'b000;
blue<=2'b00;
end

2'b10:begin
red<=3'b000;
green<=3'b111;
blue<=2'b00;
end

default:begin
red<=3'b000;
green<=3'b000;
blue<=2'b00;
end

endcase
end

//pos5
else if(H_counter > 321 && H_counter < 441 && V_counter > 215 && V_counter < 335) begin
case (pos5)

2'b01:begin
red<=3'b111;
green<=3'b000;
blue<=2'b00;
end

2'b10:begin
red<=3'b000;
green<=3'b111;
blue<=2'b00;
end

default:begin
red<=3'b000;
green<=3'b000;
blue<=2'b00;
end

endcase
end

//pos6
```

```
else if(H_counter > 474 && H_counter < 594 && V_counter > 215 && V_counter < 335) begin
case (pos6)

2'b01:begin
red<=3'b111;
green<=3'b000;
blue<=2'b00;
end

2'b10:begin
red<=3'b000;
green<=3'b111;
blue<=2'b00;
end

default:begin
red<=3'b000;
green<=3'b000;
blue<=2'b00;
end

endcase
end

//pos7
else if(H_counter > 168 && H_counter < 288 && V_counter > 368 && V_counter < 488) begin
case (pos7)

2'b01:begin
red<=3'b111;
green<=3'b000;
blue<=2'b00;
end

2'b10:begin
red<=3'b000;
green<=3'b111;
blue<=2'b00;
end

default:begin
red<=3'b000;
green<=3'b000;
blue<=2'b00;
end

endcase
end

//pos8
else if(H_counter > 321 && H_counter < 441 && V_counter > 368 && V_counter < 488) begin
case (pos8)

2'b01:begin
red<=3'b111;
green<=3'b000;
blue<=2'b00;
end
```

```
2'b10:begin
red<=3'b000;
green<=3'b111;
blue<=2'b00;
end

default:begin
red<=3'b000;
green<=3'b000;
blue<=2'b00;
end

endcase
end

//pos9
else if(H_counter > 474 && H_counter < 594 && V_counter > 368 && V_counter < 488) begin
case (pos9)

2'b01:begin
red<=3'b111;
green<=3'b000;
blue<=2'b00;
end

2'b10:begin
red<=3'b000;
green<=3'b111;
blue<=2'b00;
end

default:begin
red<=3'b000;
green<=3'b000;
blue<=2'b00;
end

endcase
end

// Side pannel
else if(H_counter > 642 && H_counter < 762 && V_counter > 47 && V_counter < 503) begin

//illegal Move
if (illegal_move==1'b1) begin // If illegal move detected - BLUE
red<=3'b000;
green<=3'b000;
blue<=2'b11;
end

//no Space
if (no_space==1'b1) begin // If no space detected (or draw) - GREEN
red<=3'b111;
green<=3'b111;
blue<=2'b00;
end

// winner
case (who)         // Indentify winner by player number
```

```
2'b01:begin        // RED for player-1
red<=3'b111;
green<=3'b000;
blue<=2'b00;
end

2'b10:begin        // GREEN for player-2
red<=3'b000;
green<=3'b111;
blue<=2'b00;
end

default:begin      // If winner not detected and places are empty
red<=3'b000;
green<=3'b000;
blue<=2'b00;
end

endcase
end

else begin
red<=3'b000;
green<=3'b000;
blue<=2'b00;
end

end
end
endmodule
```

Explanation :

- All positions are displayed by 120X120 pixels in a box of 150X150 pixels.
- According to User guide, ucf files RED and GREEN will be size of 3 bits and BLUE will be size of 2 bits.
- **Player-1** displayed by *RED* colour.
- **Player-2** displayed by *GREEN* colour.
- Side pannel shows the ***winner colour*** at the end of the ganme if any palyer wins otherwise it will show *YELLOW* colour for **no space (or draw)** in the game.
- Side pannel also shows the **illegal move** by *BLUE* colour untill game runs perfectly.

- Screen pixels distributions are as follow :

parameters size	Horizontal pixels	Vertical pixels
Screen	144 to 784	31 to 511
Borad	150 to 612	44 to 506
Borders	(150 to 153) (303 to 306) (456 to 459) (609 to 612)	(44 to 47) (197 to 200) (350 to 353) (503 to 506)
Partition	622 to 627	31 to 511
Side pannel	642 to 777	47 to 503
pos 1	168 to 288	62 to 182
pos 2	321 to 441	62 to 182
pos 3	474 to 594	62 to 182
pos 4	168 to 288	215 to 335
pos 5	321 to 441	215 to 335
pos 6	474 to 594	215 to 335
pos 7	168 to 288	368 to 488
pos 8	321 to 441	368 to 488
pos 9	474 to 594	368 to 488

RTL Sychematic :

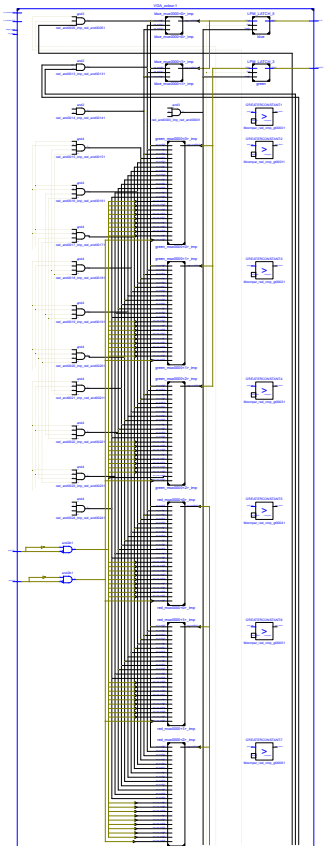


Figure 3.6: RTL Schematic of VGA colour module

Chapter 4

Datapath & Controller Module

4.1 Block diagram-

In the block diagram given below,

INPUT SIGNAL :

- clk_in (FPGA standard clock - 50MHz)
- reset (overall reset)
- play (player-1 play signal)
- play2 (player-2 play signal)
- x_position (input position)

OUTPUT SIGNAL :

- hsync (Horizontal Synchronous)
- vsync (Vertical Synchronous)
- RED
- GREEN
- BLUE

These output signal is linked with VGA port on FPGA board and that will be given to the VGA connector and output can be displayedd over the screen through VGA connector.

INTERNAL SIGNAL* :

- Datapath to Controller :
  - play
  - play2
  - no_space
  - illegal_move
  - winner
- Controller to Datapath :
  - player_play
  - player2_play

*to communicate with datapath and controller

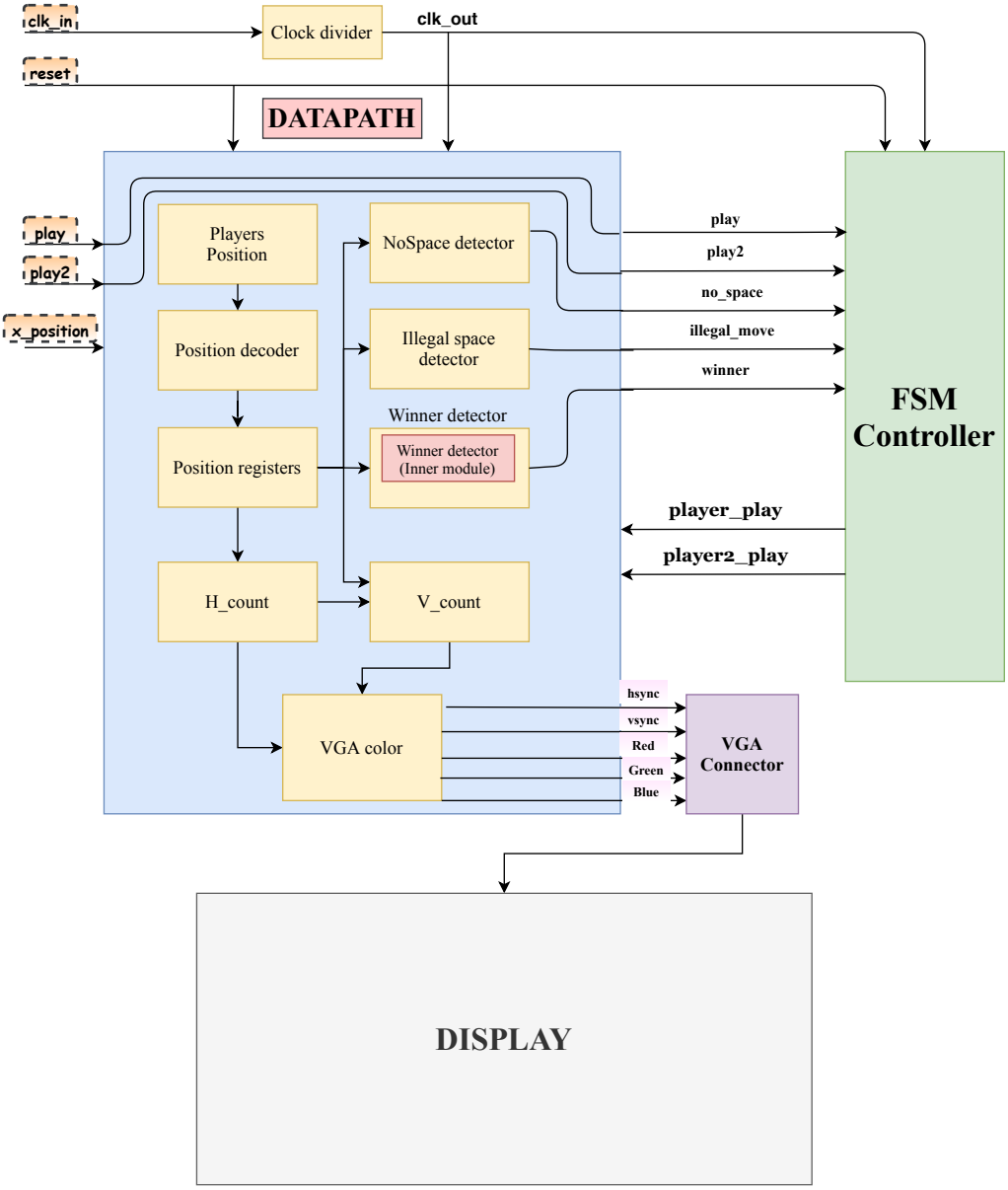


Figure 4.1: Block diagram model

4.2 Datapath (main) module-

this is the main datapath module which connects all the modules to gether and works with FSM controller.

Verilog Code :

```
‘timescale 1ns / 1ps
module main(input clock, // clock of the game from FPGA
input reset, // reset button to reset the game
input play, // play button to enable player to play
input play2, // play2 button to enable PLAYER2 to play
input [3:0] x_position,
// positions to play

output hsync, // Horizontal synchronous
output vsync, // Vertical synchronous
output [2:0] red, // RED
output [2:0] green, // GREEN
output [1:0] blue  // BLUE
);

wire [3:0] player2_position,player_position; // individual positions of both players

wire [1:0] pos1,pos2,pos3,
pos4,pos5,pos6,pos7,pos8,pos9; // position registers // 01: Player-1 & 10: Player-2

wire[1:0]who; //  winner player number

wire [15:0] PL2_en;// Player2 decoded signals
wire [15:0] PL_en; // Player decoded signals
wire illegal_move; // illegal move detecter signal
wire win; // winner signal
wire player2_play; // PLAYER2 enabling signal
wire player_play; // player enabling signal
wire no_space; // no space signal

wire clk2; // synchronous clock

// FOR VGA controller
wire [15:0] H_counter; // Horizontal counter
wire [15:0] V_counter; // Vertical Counter
wire v_signal; // Vertical counter enable signal

clk_d cd1(clock,clk2); // clk2 devider to get 25MHz from FPGA clock

// individual position identifier
P_position P1(play,play2,x_position,player_position,player2_position);

// position registers
position_registers
    position_reg_unit(clk2,reset,illegal_move,PL2_en[9:1],PL_en[9:1],pos1,pos2,pos3,pos4,pos5,pos6,pos7,pos8,pos9);

// winner detector
winner_detector win_detect_unit(pos1,pos2,pos3,pos4,pos5,pos6,pos7,pos8,pos9,win,who);

// position decoder for PLAYER2
position_decoder pd1(player2_position,player2_play,PL2_en);

// position decoder for player
position_decoder pd2(player_position,player_play,PL_en);
```

```
// illegal move detector
illegal_move_detector imd_unit(play,play2,pos1,pos2,pos3,pos4,pos5,pos6,pos7,pos8,pos9, PL2_en[9:1],
    PL_en[9:1], illegal_move);

// no space detector
nospace_detector nsd_unit(pos1,pos2,pos3,pos4,pos5,pos6,pos7,pos8,pos9,no_space);

// FSM controller
fsm_controller tic_tac_toe_controller(
clk2,   // clock of the circuit
reset,  // reset
play,   // player plays
play2,  // PLAYER2 plays
illegal_move,  // illegal move detected
no_space,      // no_space detected
win,           // winner detected
player2_play,  // enable PLAYER2 to play
player_play    // enable player to play
);

// VGA Controller Units

//Horizontal counter
H_count hc(clk2,v_signal,H_counter);

//Vertical Counter
V_count vc(clk2,v_signal,V_counter);

//VGA controller main unit
VGA_colour
    VGA1({pos1,pos2,pos3,pos4,pos5,pos6,pos7,pos8,pos9},illegal_move,no_space,who,H_counter,V_counter,hsync,vsync,
red,green,blue);

endmodule
```

RTL Sychematic :

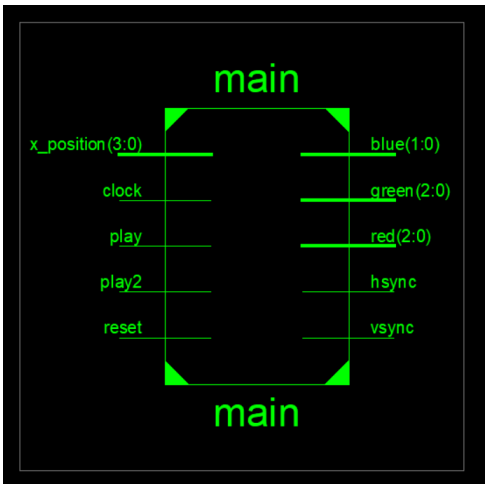


Figure 4.2: Datapath main unit (Upper layer)

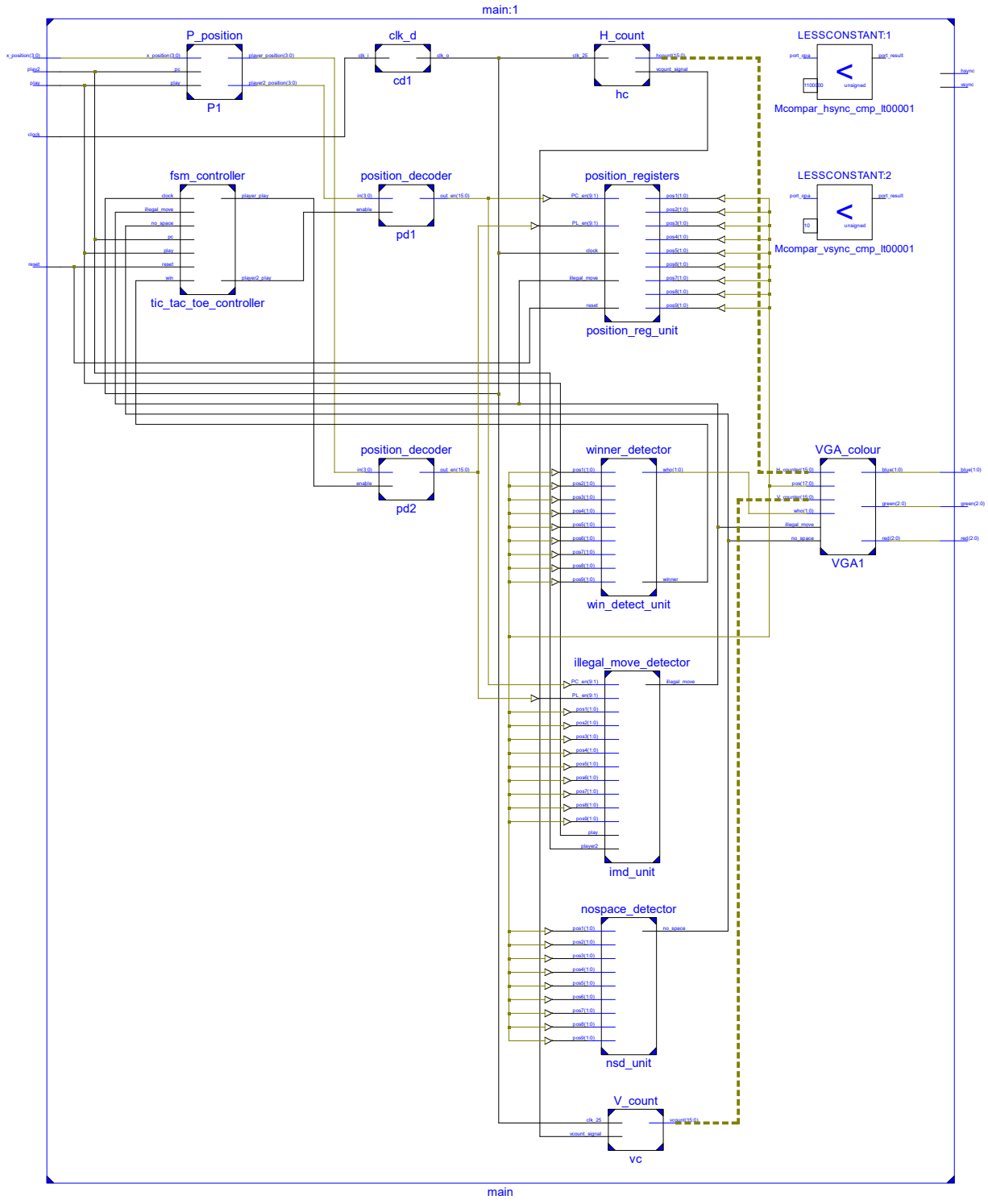
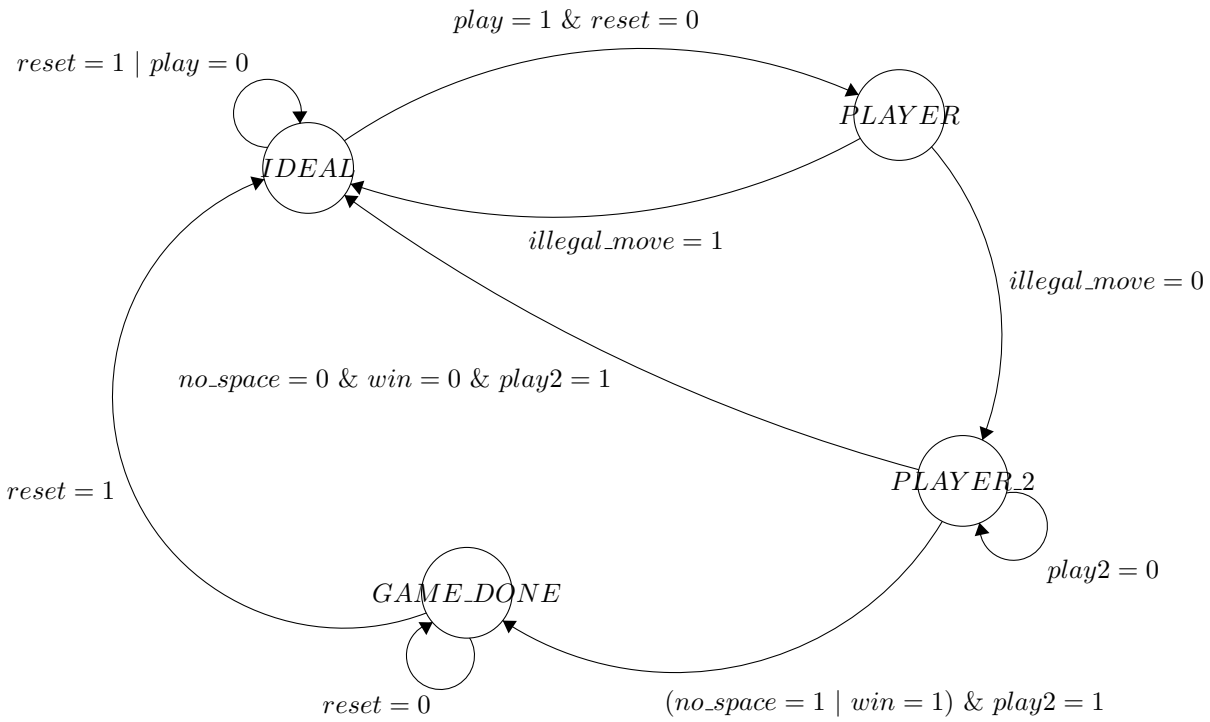


Figure 4.3: RTL Schematic of Datapath main Unit

4.3 FSM Controller-

This is *Finite State Machine(FSM) Controller* module which is sequential and control the other modules by enabling signals as shown above in block diagram.

FSM - State diagram



Verilog Code :

```
‘timescale 1ns / 1ps
module fsm_controller(
input clock,// clock of the circuit
input reset,// reset
input play, // player plays
input play2,// PLAYER2 plays
input illegal_move,// illegal move detected
input no_space, // no_space detected
input win, // winner detected
output reg player2_play, // enable PLAYER2 to play
output reg player_play // enable player to play
);

// FSM States
parameter IDLE=2’b00;
parameter PLAYER=2’b01;
parameter PLAYER2=2’b10;
parameter GAME_DONE=2’b11;

reg[1:0] current_state, next_state;

// current state registers
```

```
always @(posedge clock or posedge reset)
begin
if(reset)
current_state <= IDLE;
else
current_state <= next_state;
end

// next state
always @(*) begin
case(current_state)

IDLE: begin
if(reset==1'b0 && play == 1'b1) next_state <= PLAYER; // player to play
else next_state <= IDLE;
player_play <= 1'b0;
player2_play <= 1'b0;
end

PLAYER:begin
player_play <= 1'b1;
player2_play <= 1'b0;
if(illegal_move==1'b0) next_state <= PLAYER2; // PLAYER2 to play
else next_state <= IDLE;
end

PLAYER2:begin
player_play <= 1'b0;
if(play2==1'b0) begin
next_state <= PLAYER2;
player2_play <= 1'b0;
end

else if(win==1'b0 && no_space == 1'b0) begin
next_state <= IDLE;
player2_play <= 1'b1;// PLAYER2 to play when play2=1
end

else if(no_space == 1 || win == 1'b1) begin
next_state <= GAME_DONE; // game done
player2_play <= 1'b1;// PLAYER2 to play when play2=1
end
end

GAME_DONE:begin // game done
player_play <= 1'b0;
player2_play <= 1'b0;
if(reset==1'b1) next_state <= IDLE;// reset the game to IDLE
else  next_state <= GAME_DONE;
end

default: next_state <= IDLE;
endcase
end
endmodule
```

RTL Sychematic :

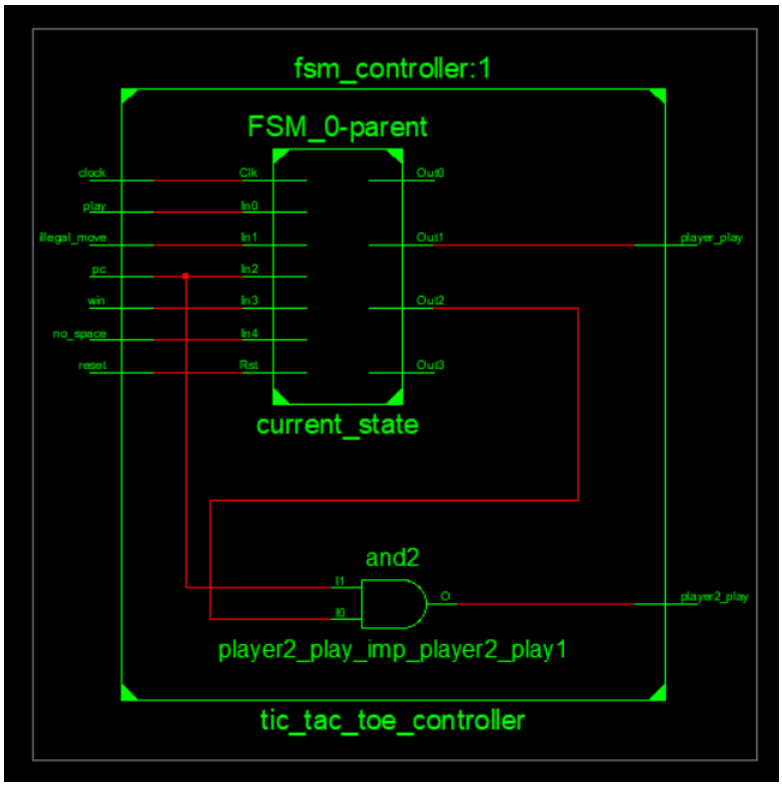


Figure 4.4: RTL schematic of FSM Controller

Chapter 5

Keyboard Interface

5.1 Introduction

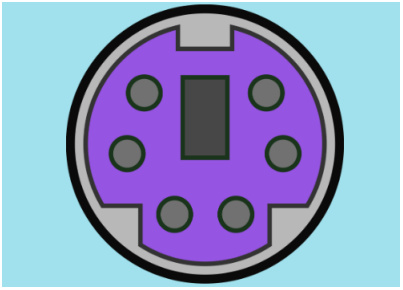


Figure 5.1: PS2 Port

Keyboard interface for Verilog will play an important role in taking the input from the user(s) about where to place their symbols turn by turn. Today's world is working with USB type keyboards but since the resources were limited, we learnt about the working a PS2 connection based keyboard for the project. It is basically an older connection type but still works fine.

5.2 Working

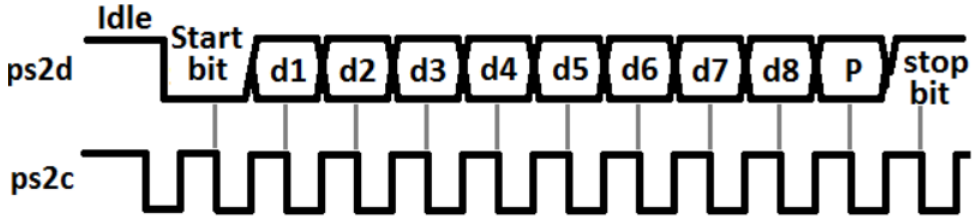


Figure 5.2: PS2 Signals Via Wires

A PS2 type connection uses two wires to transmit data, which are called DATA and CLOCK. When we press the button on the keyboard, the micro-controller inside the keyboard sends packets of 11 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 7 bits for DATA0-6, then a PARITY bit and finally STOP bit. After the data is received, and checked that it is correct (START, PARITY and STOP bits have adequate

values) you only need to keep DATA0-6 bits and get rid of the rest.

Now looking broader, when a button is pressed, the keyboard sends one packet of 11 bits to tell which button was pressed. 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. This is a little different for different for so called extended buttons, where it sends one more packet and to know what code must be received you can read manual or lets say here.

The frequency it transmits data is quite low and can vary from 10 kHz to 16.7 kHz. This becomes important, when you use an FPGA with speeds of 50 MHz and larger. You have to scale the frequency down by some factor, otherwise you will be reading the wrong CLOCK signal. Scaling the FPGA board to around 30-50 kHz should do the job, but again this will depend on the keyboard.

Finally it is important to describe what will happen if you receive less than 11 bits as sometimes this happens as well. Very rarely, but still happens... What I did here is put a timer to tell if enough time has passed from receiving the bit and if the time period is too long until receiving the next bit, it flushes everything that it received until this point.

5.3 Verilog Code

Two counters are used within this module:

- DOWNCOUNTER is used to bring the frequency 250 times as TRIGGER is triggered wether that happens.
- $count_reading$ counts up if not a full packet of 11 bits is received yet. (later it checks if this number reached 4000...)
- This module receives the information and checks if it OK.
- If the TRIGGER is triggered, then check if the $PS2\_CLK$ changed its state
- If the state changed and if the clock is on falling edge do:

  Add up the DATA bit that was currently received to the previous bits

  Mark down that one more bit was received

  If 11 bits were received, trigger out another signal, telling that it finished reading

  Check the parity bit if the received information is OK

  If 11 bits were not received check if it took more than 4000 times for $count_reading$ to count up after it received the previous bit.

  and reset everything if it took more. Else skip this.

Another little module extracts the information:

- Wait for the trigger, which checks if the full pack of 11 bits was received
- If there was an error in the received packet, discard everything.
- However extract the DATA bits if the information was OK.

```
`timescale 1ns / 1ps
module Keyboard(
   input CLK, //board clock
   input PS2_CLK, //keyboard clock and data signals
   input PS2_DATA,
// output reg scan_err,    //These can be used if the Keyboard module is used within a another module
// output reg [10:0] scan_code,
// output reg [3:0]COUNT,
// output reg TRIG_ARR,
```

```
// output reg [7:0]CODEWORD,
   output reg [7:0] LED //8 LEDs
   );

   wire [7:0] ARROW_UP = 8'h75; //codes for arrows
   wire [7:0] ARROW_DOWN = 8'h72;
   //wire [7:0] ARROW_LEFT = 8'h6B;
   //wire [7:0] ARROW_RIGHT = 8'h74;
   //wire [7:0] EXTENDED = 8'hE0; //codes
   //wire [7:0] RELEASED = 8'hF0;

   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;
      LED = 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)
            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
          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  codeword
      else CODEWORD <= 8'd0;        //not a full packet received, thus reset codeword
    end
    else CODEWORD <= 8'd0;          //no clock trigger, no   data
  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,  thats  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 == EXTENDED)      //For example you can check here if specific codewords were received
```

```
        //if (CODEWORD == RELEASED)
      //end
// end
   end

endmodule
```

5.4 RTL Schematic

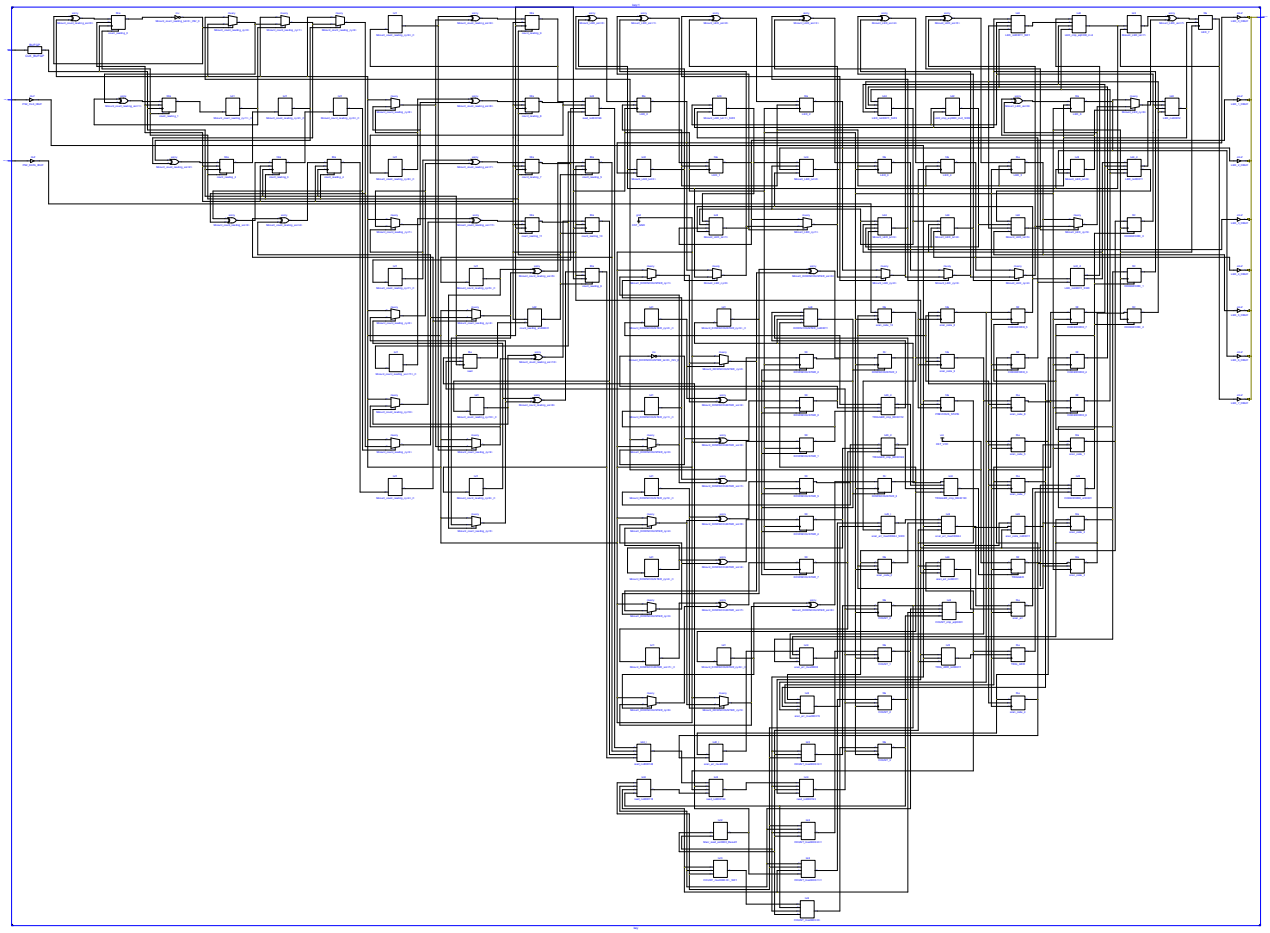
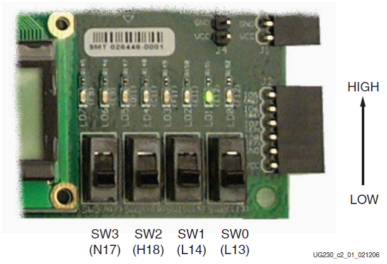


Figure 5.3: RTL Schematic of Key-Board Interface

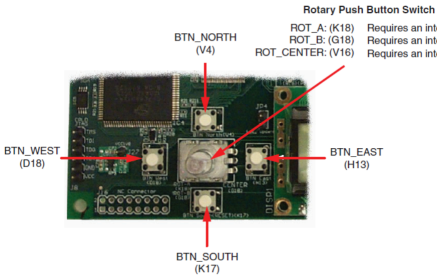
Chapter 6

Overall Explanation

- Since Key-board Interface in not designed completely, Input can be given from the switches and push-buttons on the FPGA board.



(a) FPGA Switches



(b) FPGA Pushbuttons

- As shown above, Four switches can be used to positions as 4 digit binary number and pushbuttons can be used to play by players and for reset.
- Given position by users will be stored in the $x\_position$ and can be separated into $player\_position$ and $player2\_position$ with the input signal play and play2 respectively.
- Once $player-1$ played, $player-2$ will get chance to play and stored $x\_position$ will be re stored into $player2\_position$ on the input signal play2.
- Like this, Both player will play the game one after the an other and their chosen position will be stored into the unique $positionregisters$ of 9 places with number 01 for player-1, 10 for $player-2$ and 00 for the empty positions.
- If in-case illegal move happens by the giving the preoccupied positions as the input or both player trys to play at same time that will be detected and $illegal\_move$ variable goes high and stops the game until everything goes normal.
- Finally if anyone wins that will be detected by the $winner\_detect$ module and player's number will be stored in the who variable along with win variable goes high.
- If game goes into draw conditions or no space will be left on the game board then nospace_detect will detect that and variable named no_space goes high.
- all this information in the variables like 9 positions, illegal move, no space, who and win will be forwarded to the $VGAcolormodule$ and that will display information on the screen as discussed above by counting every pixels on the display.
- $Globalreset$ will reset the complete game and erase all the data stored in any variable and that will lead to intial board setup screen on the display.

Chapter 7

TEST BENCH & Simulation

7.1 Introduction -

This module contains the test bench and the simulation results of the overall project as name suggests. This helps us to get better understanding for actual hardware Implementation.

7.2 Test bench-

This is Test bench of the Datapath main module.

Verilog Code :

```
`timescale 1ns / 1ps
module main_tb;

// Inputs
reg clock;  // clock of FPGA 50MHz
reg reset;  // Global reset
reg play;   // player-1 signal
reg play2;  // player-2 signal
reg [3:0] x_position; // position

// Outputs
wire hsync;  // Horizontal sync
wire vsync;  // Vertical Sync
wire [2:0] red; // RED
wire [2:0] green; //GREEN
wire [1:0] blue; //BLUE

// Instantiate the Unit Under Test (UUT)
main uut (
.clock(clock),
.reset(reset),
.play(play),
.play2(play2),
.x_position(x_position),
.hsync(hsync),
.vsync(vsync),
.red(red),
.green(green),
```

```
.blue(blue)
);

initial begin
clock=0;
forever #10 clock=~clock; // clock of 50 MHz => time period = 20 ns
end

initial begin
// Initialize Inputs

reset = 1;
play = 0;
play2 = 0;
x_position = 1;

#18000000;  // Waiting time = 18 ms > 16.67 ms

reset = 0;
play = 1;
play2 = 0;
x_position = 1;

#18000000; // Waiting time = 18 ms > 16.67 ms

play = 0;
play2 = 0;
x_position = 5;

#18000000; // Waiting time = 18 ms > 16.67 ms

play = 0;
play2 = 1;
x_position = 5;

#18000000;  // Waiting time = 18 ms > 16.67 ms

play = 1;
play2 = 0;
x_position = 2;

// Waiting time = 18 ms > 16.67 ms
#18000000;

play = 0;
play2 = 1;
x_position = 7;

#18000000;  // Waiting time = 18 ms > 16.67 ms

play = 0;
play2 = 0;
x_position = 8;

#18000000;   // Waiting time = 18 ms > 16.67 ms

play = 1;
play2 = 0;
x_position = 3;
```

```
#18000000;    // Waiting time = 18 ms > 16.67 ms

play = 0;
play2 = 1;
x_position = 9;

#18000000;    // Waiting time = 18 ms > 16.67 ms

$finish;

end
endmodule
```

Explanation :

- As discussed in the VGA_connector module, there should be at least 16.672 ms waiting time between changes of any input to counts every pixels on screen, We considered 18 ms as waiting time here.

7.3 Simulation -

This is the simulation of the test bench of the main datapath verilog file which was presented earlier.

Though Simulation file is quite big because total time is 160ms, we try to put a part of simulation here.*

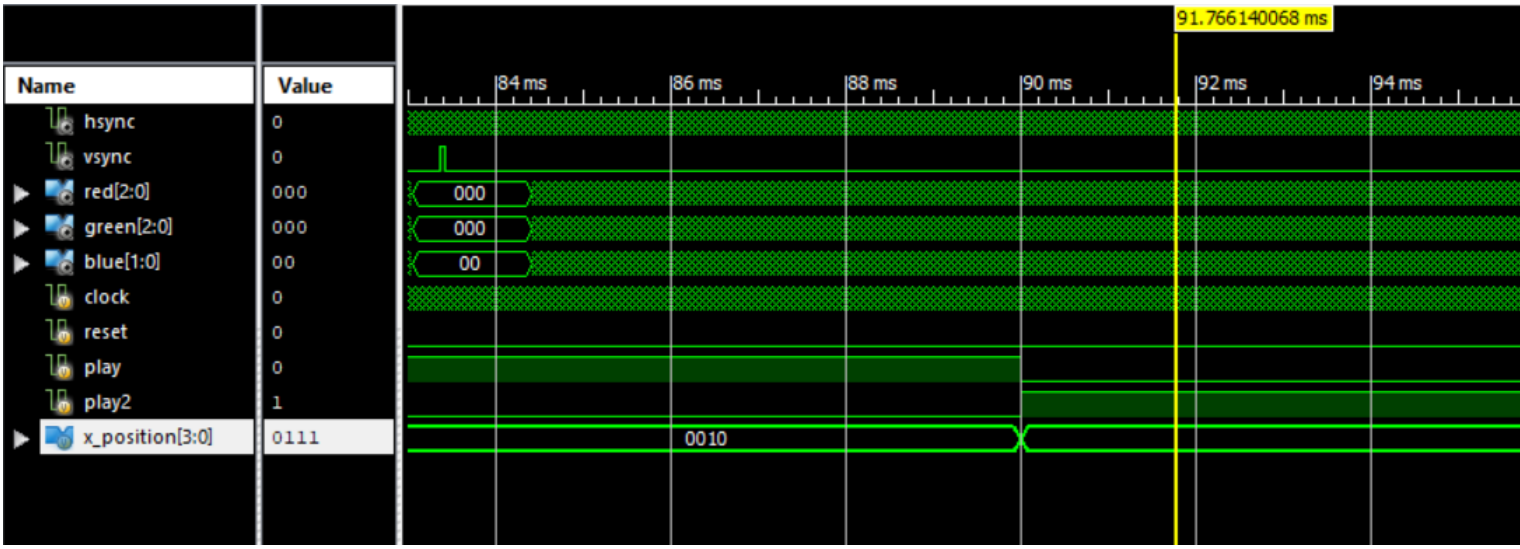


Figure 7.1: Part of Simulation file of Test bench

*More details can be found out from the simulation file send along with this report.

Explanation :

- *x_position* is stored into the respective players position and that will be lead to store players unique number *(Player-1–01 & Player-2–10)* into the position registers as shown below.

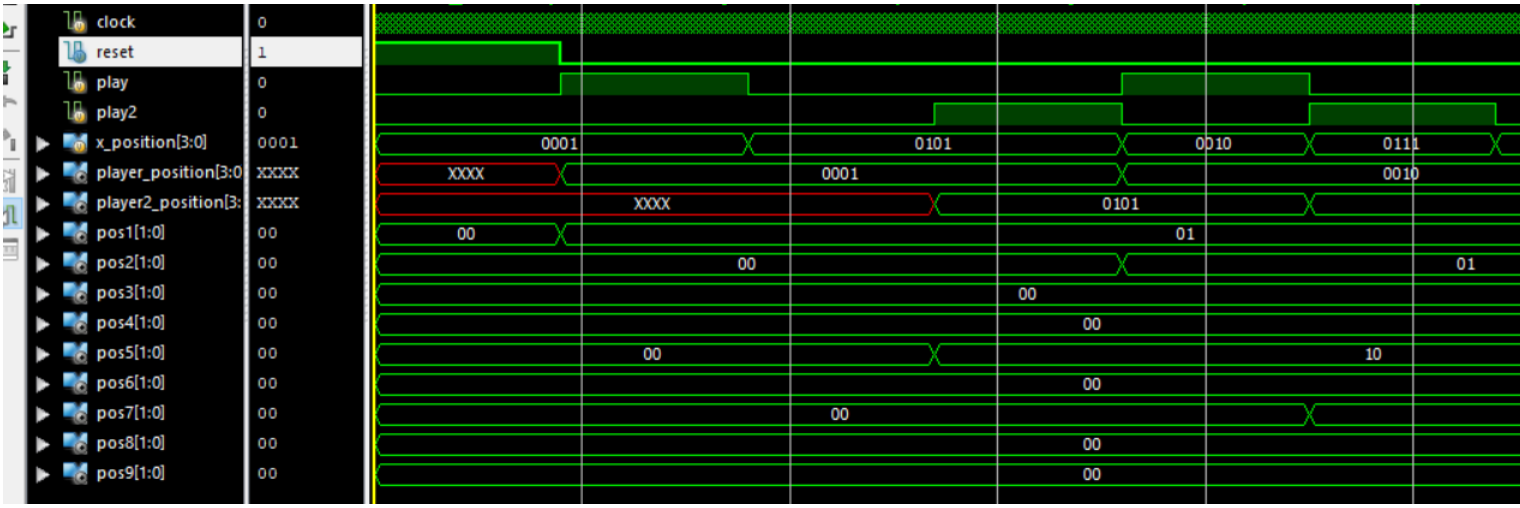


Figure 7.2: Stored positions into respective registers

- When any illegal move happens, *illegal_move* goes high as well as If anyone wins the game, win goes high and who stores the number of player as shown below.

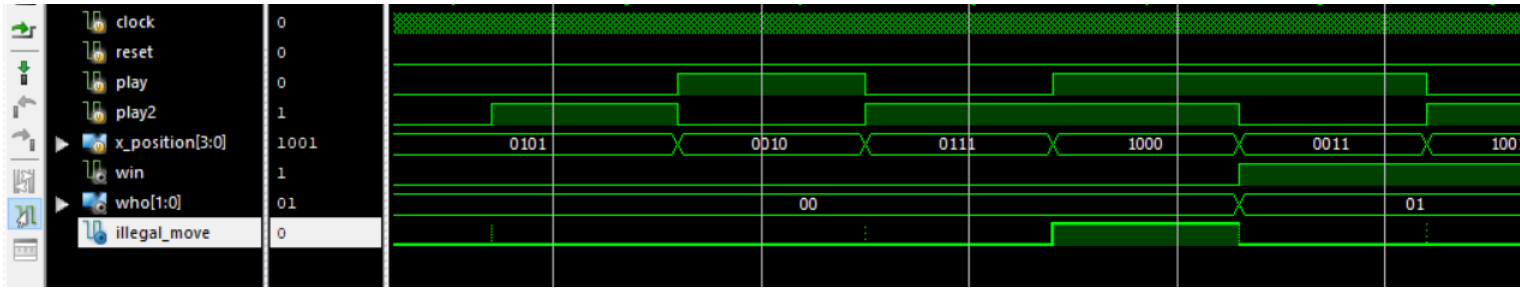


Figure 7.3: Detected winner and illegal move

- As we discussed, time taken by both counter to count every pixels on the screen requires 16.67 ms and that's also called *waiting time*.

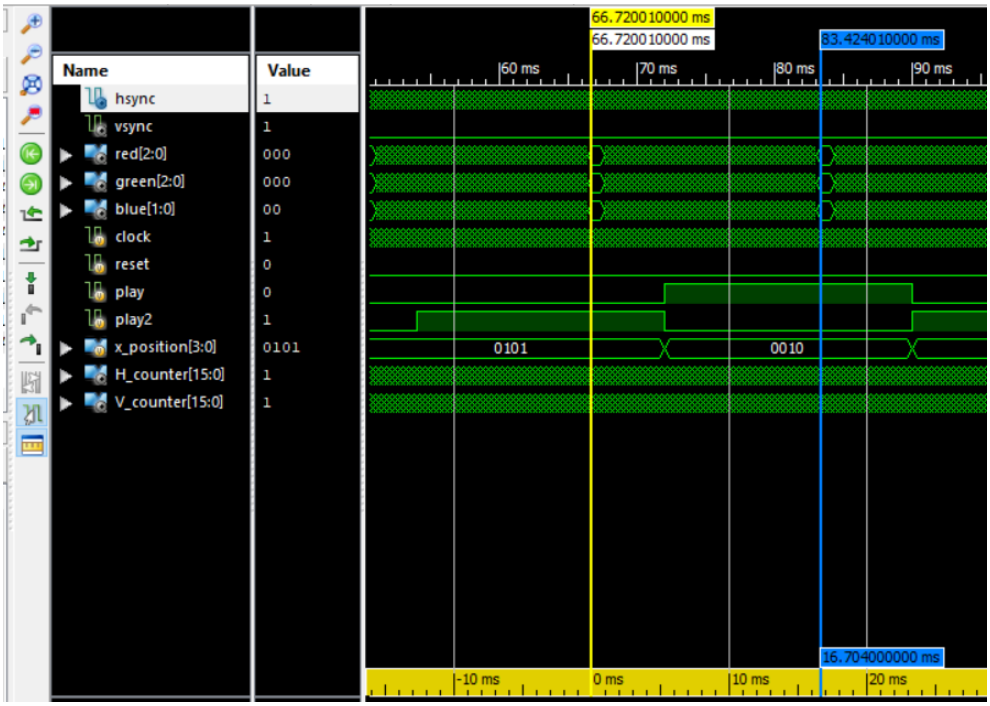


Figure 7.4: Cursor at counter position-1

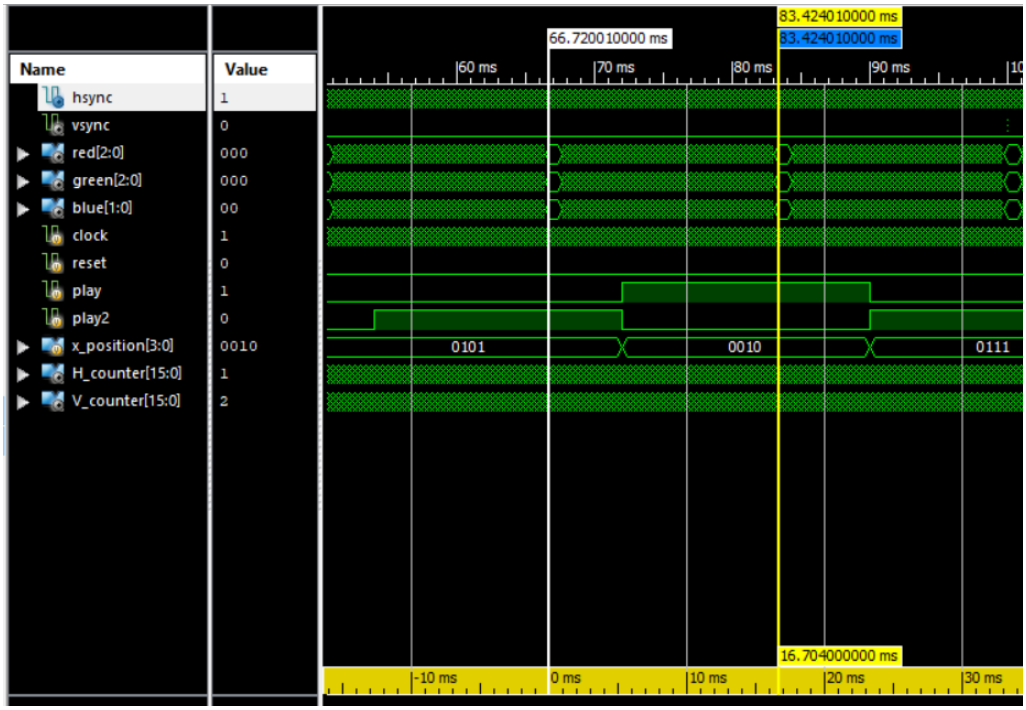


Figure 7.5: Cursor at counter position-2

- As shown above, *H_count* and *V_count* are at position 1 in the 1^{st} figure at the timeline 66.72 ms and H_count and V_count are at position 2 in the 2^{nd} figure at the timeline 83.42 ms. So, time difference is 16.7ms as shown on the timeline given in the bottom of figure.
- As we can see in the figure below, *RED*, *GREEN*, *BLUE* and other VGA parametters as *H_sync* and *V_sync* are also changing with *H_count* and *V_count* as we desigened earlier.

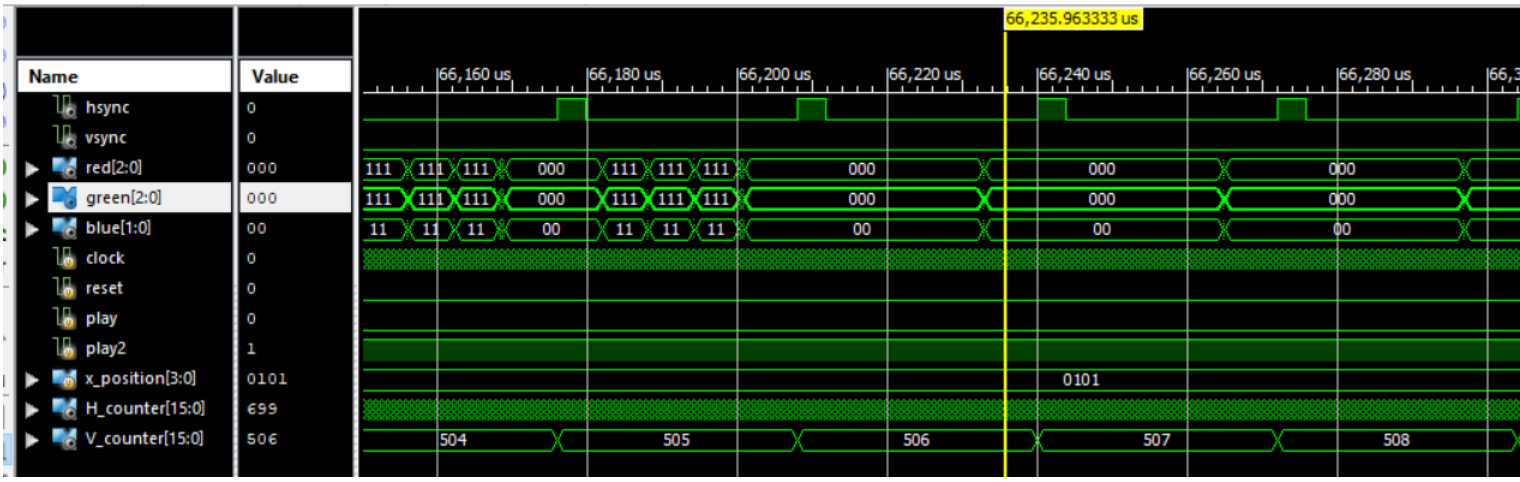


Figure 7.6: Change of RGB with H_count and V_count

Chapter 8

Expected Screen Results

8.1 Introduction -

Though we didn't able to implement our project on hardware, Here are some expected screen results from our design.

8.2 For the given TEST BENCH -

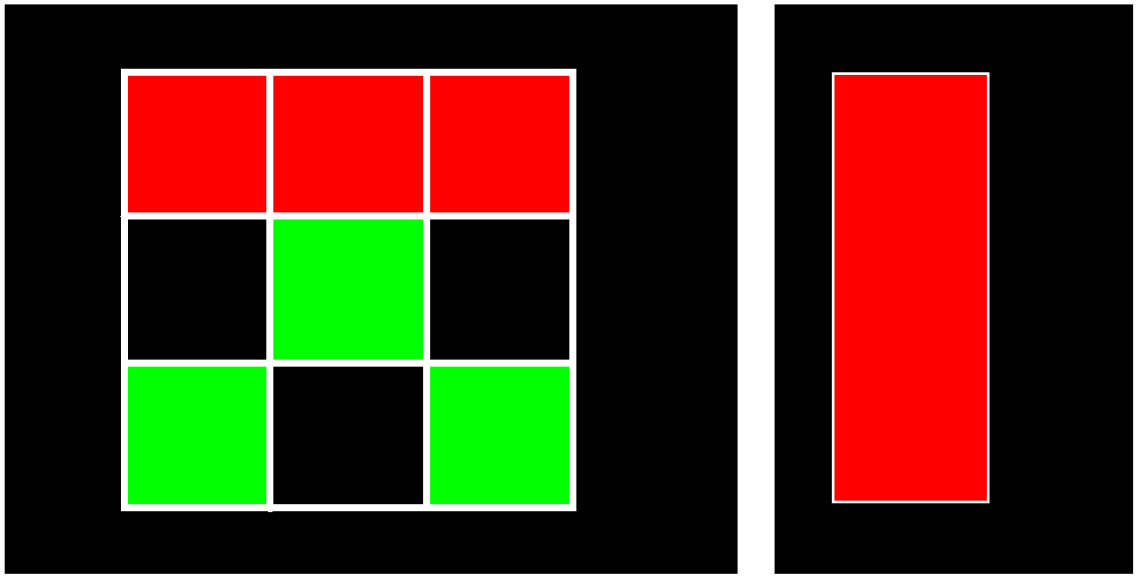


Figure 8.1: Expected Results on Screen by RGB

8.3 For the Illegal move possibility :

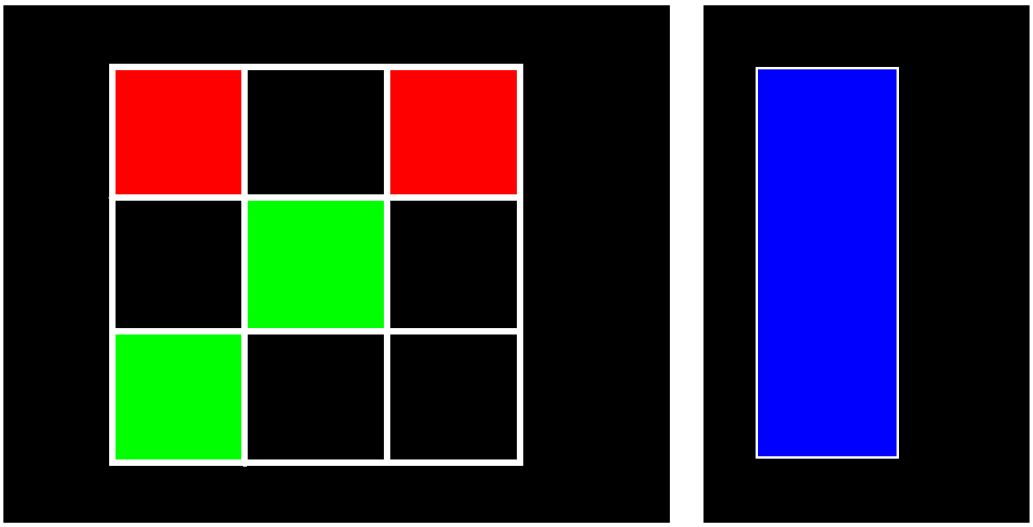


Figure 8.2: Expected Results on Screen by RGB

8.4 For the No Space (or Draw) possibility :

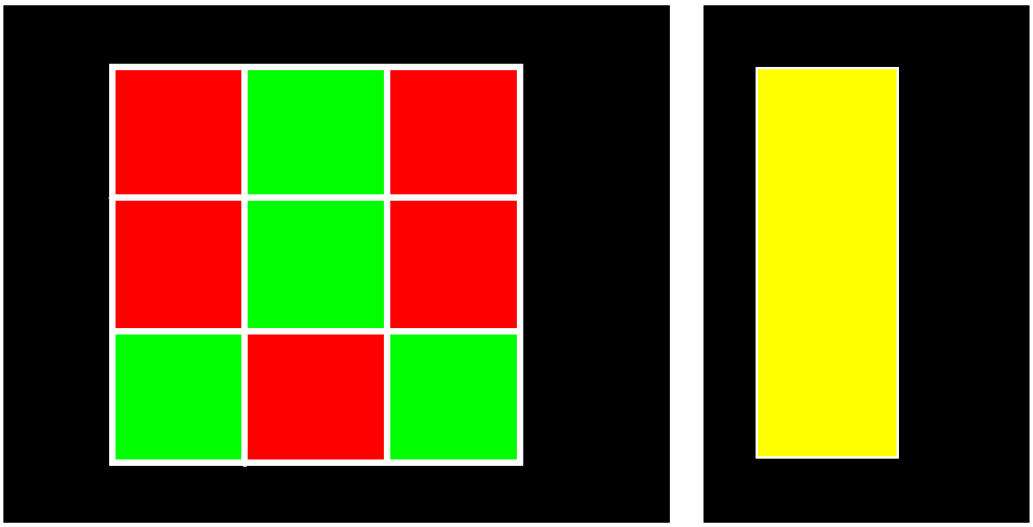


Figure 8.3: Expected Results on Screen by RGB

Chapter 9

Inferences & Bibliography

9.1 Inferences -

- We designed the Overall Game structure with VGA interface using the help our Subject knowledge and references.
- We learned few aspects of Verilog Coding by this Project and get better understandably and Ideas of the Verilog Language.
- We tried to utilize our best time and completed it as much as we can still we were unable to integrate PS2-Keyboard Interface with the project completely due to time and knowledge limit.The PS2 Keyboard Interface has been designed and checked individually via software based synthesis and simulations.
- As mentioned in the project, Due to some constrain we couldn't implement it on FPGA board but we tried to simulate it correctly hope it works correctly on hardware.

9.2 Bibliography -

- www.fpga4student.com
- https://en.wikipedia.org/wiki/Tic-tac-toe
- FPGA spaten 3E starter board-User Guide
- VGA Connector with FPGA : https://youtu.be/4enWoVHCykI
- Design and Implementation of VGA Controller on FPGA Renuka A. Wasu, Vijay R. Wadhankar
- for diagram : draw.io
- https://embeddedthoughts.com/2016/07/05/fpga-keyboard-interface/
- https://verilog-usb-keyboard.loadoutspastruese.fun/page.php