Project Final Report

FPGA Tic Tac Toe Game with VGA Display

Breeha Qasim - 08283
Namel Shahid - 08327
Irfan Sohail - 07353
Aiman Rizwan - 08513

December 2023

Digital Logic and Design

Contents

1 Introduction

The FPGA Tic Tac Toe project represents an advanced iteration of the traditional game, wherein the application of sophisticated digital logic circuits enhances the overall gaming experience. Utilizing an FPGA board as the foundational platform, this project demonstrates a commitment to leveraging cutting-edge technology for recreational purposes.

Within this intricate framework, two players engage in the game by inputting their moves through a meticulously integrated keypad. Each player is represented by a distinct symbol, such as a square (□), with the added nuance of individualized color assignments for visual clarity. The underpinning digital logic circuits, comprising registers, decoders, and precisely synchronized clocks, collaboratively govern the gameplay dynamics, ensuring a responsive and engaging experience for participants.

As the game unfolds, the system diligently processes and assesses the moves, employing its intricate architecture to identify the victorious player. The denouement of the game is visually communicated on the display, underscoring the triumph of the winning participant.

This comprehensive undertaking not only underscores the practical application of digital logic circuits but also serves as an educational endeavor, offering insights into FPGA-based game development. From the meticulous implementation of registers and decoders to the precise synchronization of time with clocks, each facet of this project contributes to a nuanced understanding of FPGA programming and its tangible implications in crafting interactive and entertaining systems.

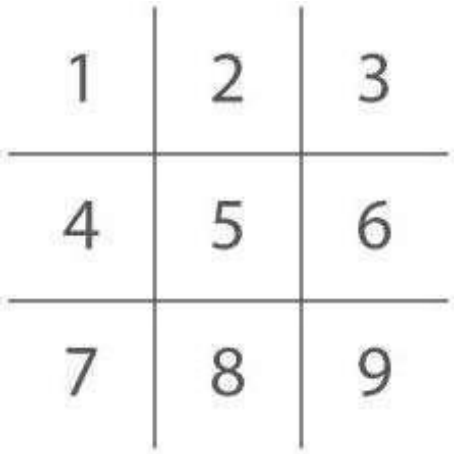


Figure 1: Tic Tac Toe board

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:

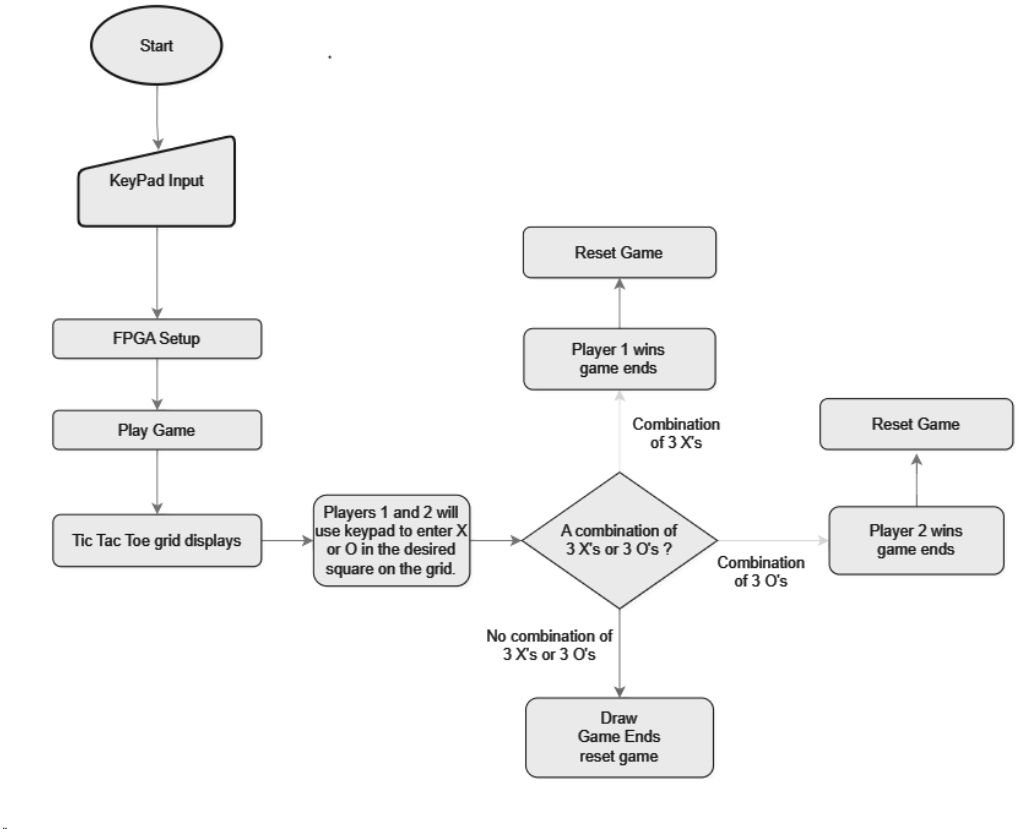


Figure 2: User Flow Diagram

2.2 Explanation:

The illustrated flowchart serves as a comprehensive guide to the sequential steps involved in playing a straightforward game of Tic-Tac-Toe. Players then take turns strategically placing their respective symbols on the game board, prompting continuous checks for a winning combination following each move. In the event of a determined victor, the game culminates with the display of the winning player. Should no winner emerge, the game persists until either a player achieves victory or the board reaches full capacity, resulting in a draw. Notably, a reset button is incorporated, affording users the option to restart the game and commence a new round at their discretion.

2.3 Division of Task:

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

1. The primary responsibility of the input block in our Tic-Tac-Toe project lies in detecting and responding to user actions, specifically the selection of the desired game mode. This includes recognizing key inputs such as □ and conveying this information to the control block for further processing.

2. The control block manages the configuration of the FPGA based on the chosen game mode. It oversees the turn-based placement of symbols on the game board, continuously checks for a winning combination, and facilitates the conclusion of the game by displaying the winner or handling a draw scenario. The control block ensures the seamless execution of the game logic and transitions between different states.

3. The output block is tasked with visually representing the outcomes of the Tic-Tac-Toe game based on the decisions made by the control block. It displays the results of the game, such as indicating the winning player or signaling a draw. The output block may also be responsible for managing the user interface elements, ensuring that the game state is appropriately communicated to the player. In the event of a reset, the output block plays a role in updating the display to initiate a new round of the game.

3 Input Block

3.1 Introduction to the input peripheral: The Keypad

A keypad equipped with six buttons serves as the input device for playing Tic-Tac-Toe. Each button is mapped to a specific position on the game grid. The player in turn uses this keypad to make their move on the board.

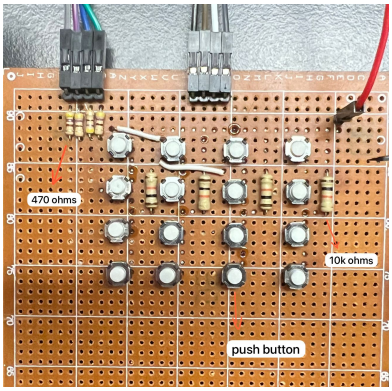


Figure 3: Keypad from inside

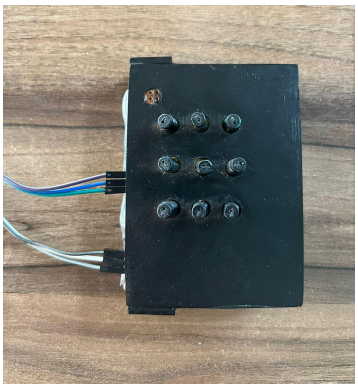


Figure 3: Keypad from outside

3.2 Inputs in the Project

The keypad is connected to the FPGA through seven wires. One wire is designated for VCC, while three wires serve as

outputs from the keypad to the FPGA. Additionally, three wires function as outputs from the FPGA to control the permissible inputs from the keypad. This configuration is implemented to prevent multiple placements on a single position on the grid, ensuring the game adheres to the rules of Tic-Tac-Toe.

3.3 Configuring the keypad

The input module(submit move) employs a 4-state finite state machine to systematically cycle through various column configurations, denoted as st_1 to st_4. The assignment of column values is contingent upon the current state within this finite state machine. As the module progresses, the next state is determined based on predefined rules. This utilization of a finite state machine enhances the control and sequencing of column configurations, contributing to the overall functionality and efficiency of the system.

3.4 Input translation to Output

The inputs from the keypad are translated into corresponding position variables, namely pos1 to pos9. Each of these variables signifies which player made a move at the respective position. Specifically, the value 01 is assigned to represent player 1's move, while 10 is assigned for player 2's move. Additionally, a variable named "player_out" stores information about the player who made the most recent move, thereby facilitating the tracking of player turns and moves within the system.

3.5 Input Code

```
`timescale 1ns / 1ps


module submission(
    input last_player, // last player who made a move
    output [4:1] x,// columns, signal is an output because we are driving them individually
    input [4:1] y,// rows
    input reset, //resets the game
    input reset2, // auto reset signal from the win detector fsm
    output player_out,//outputs the player who's move is being submitted
    input clk,//clock input
    output [3:0] z//output which is used to determine the position on the board being occupied in the next module
    );

reg [3:0] Z;
reg play = 1'b0;
wire sclk;

clk_div3 clk_div(
   .clk(clk),
   .sclk(sclk)
);

reg[1:0] PS, NS;
parameter st_1 = 2'b00, st_2 = 2'b01, st_3 = 2'b10, st_4 = 2'b11;
reg [4:1]col;

always@(posedge sclk)
PS <= NS;

always@(*)
begin
   begin
      case(PS)
         st_1:begin
            col = 4'b0111;
            NS = st_2;
         end

         st_2:begin
            col = 4'b1011;
            NS = st_3;
         end

         st_3:begin
            col = 4'b1101;
            NS = st_4;
         end

         st_4:begin
            col = 4'b1110;
            NS = st_1;
         end

         default: begin col = 4'b0111; NS = st_2; end
      endcase
   end
```

Figure 2: input code1

```
          default: begin col = 4'b0111; NS = st_2; end
        endcase
    end
    if (last_player == 1'b0)
       begin
       play = 1'b1;
       end
    else if(last_player == 1'b1)
       begin
       play = 1'b0;
       end
    else
       begin
       play = 1'b0;
       end
end
always@(posedge sclk)
begin
   begin
      if (col[1] == 0 && y[3] == 0)//translating input coordinates into a position
         begin
         Z = 4'b0001;
         end
      else if (col[1] == 0 && y[2] == 0)
         begin
         Z = 4'b0100;
         end
      else if (col[1] == 0 && y[1] == 0)
         begin
         Z = 4'b0111;
         end
      else if (col[2] == 0 && y[3] == 0)
         begin
         Z = 4'b0010;
         end
      else if (col[2] == 0 && y[2] == 0)
         begin
         Z = 4'b0101;
         end
      else if (col[2] == 0 && y[1] == 0)
         begin
         Z = 4'b1000;
         end
      else if (col[3] == 0 && y[3] == 0)
         begin
         Z = 4'b0011;
         end
      else if (col[3] == 0 && y[2] == 0)
         begin
         Z = 4'b0110;
         end
      else if (col[3] == 0 && y[1] == 0)
         begin
         Z = 4'b1001;
         end
      else if (col[4] == 0 && y[4] == 0) // included this case and the last else case to avoid inferring latches
         begin
         Z = 4'b1111;
```

Figure 3: input code2

```
111             begin
112             Z = 4'b1111;
113             end
114         else
115             begin
116             Z = 4'b1111;
117             end
118     end
119     if (reset == 1 || reset2 == 1)
120        begin
121             Z = 4'b0000;
122     end
123 end
124 
125 assign x = col;
126 assign player_out = play; //final assignment of registers to outputs
127 assign z = Z; //these outputs will be tied together as wires in the top module
128 // and used to determine which position is being played on the board
129 
130 endmodule
131 
132 module clk_div3 (
133     input clk,
134     output sclk
135 );
136 
137 integer MAX_COUNT = 100000; //1khz
138 integer div_cnt =0;
139 reg tmp_clk=0;
140 
141 always @ (posedge clk)
142 begin
143     if (div_cnt == MAX_COUNT)
144        begin
145        tmp_clk = ~tmp_clk;
146        div_cnt = 0;
147        end
148     else
149        div_cnt = div_cnt + 1;
150 end
151 assign sclk = tmp_clk;
152 
153 endmodule
154 
```

Figure 4: input code3

3.6 Input Elaborated Diagram

Here is the elaborated diagram of the input:

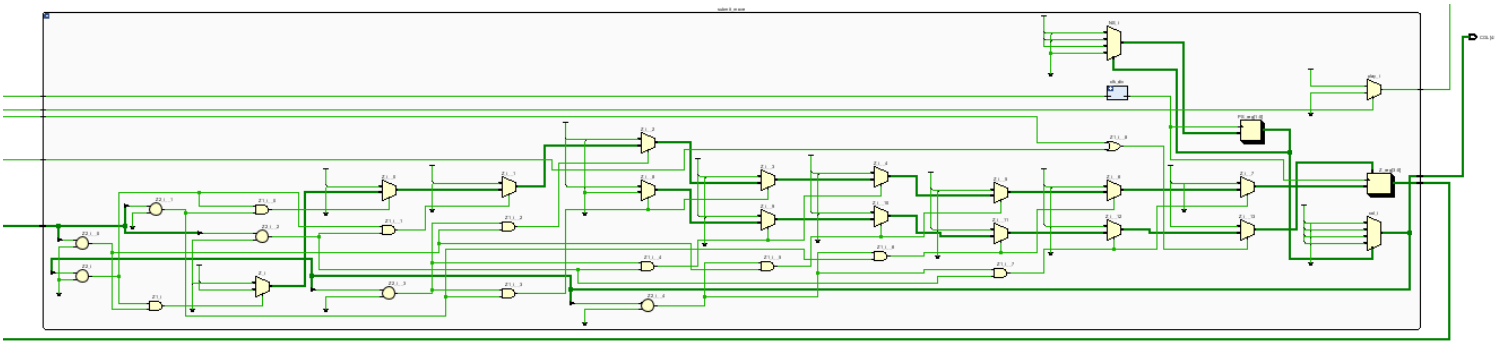


Figure 5: Elaborated Design of Input block

3.7 Input I/O Ports

ROW (4)	IN			☑
ROW[4]	IN		H1	☑
ROW[3]	IN		K2	☑
ROW[2]	IN		H2	☑
ROW[1]	IN		G3	☑

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. The following is how the group decided to map the pixels in the output display of our project.

4.2 Pixel Mapping

A grid pattern is defined by conditional statements in the always block, where the color of the pixels is determined based on the values of x and y coordinates. Different regions of the grid are assigned different colors, creating a visual display. The grid is further divided into cells, and the color of each cell is set based on the conditions specified in the code. The code also incorporates player positions (pos1 to pos9) and a winning condition (win). The RGB color for specific player positions is determined in the always block, assigning colors to represent players 1 and 2 (red and blue, respectively). When Player 1 Wins, Player 2 Wins or match is Drawn then a screen is displayed with text. To display game status text we used Bitmapping, we assigned each cell a color for it.

The draw module is responsible for setting the color of pixels based on the provided x and y coordinates, player positions, and winning condition. It creates a grid of wires (b_0_0 to b_15_11) representing individual cells on the display, and the color of each cell is determined based on the conditions specified for each block. The RGB color for each block is set in the always block, resulting in a dynamic and changing display based on the game logic and player move

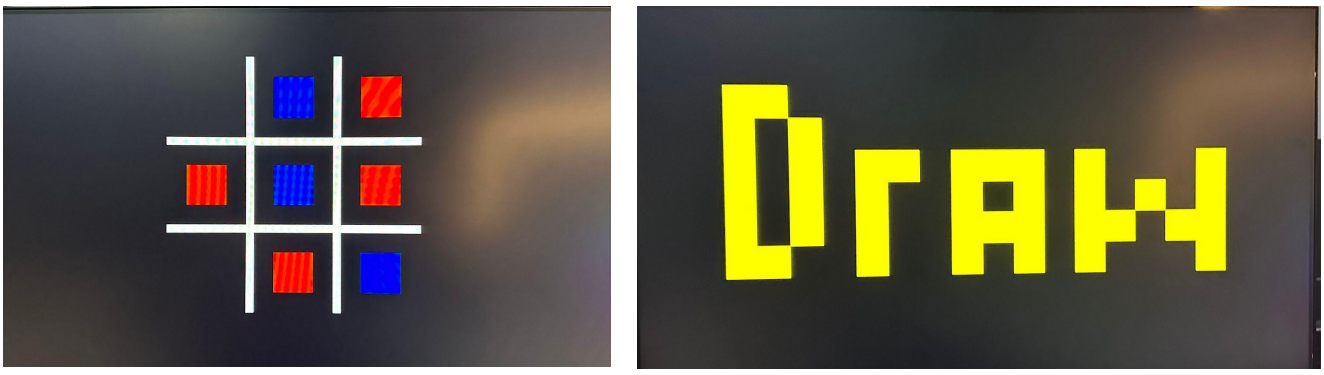


Figure 7: Display screen of the game

4.3 Output Demo Code

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

```
module vga_test
    (
        input wire clk, reset,
        input wire[1:0] pos1,pos2,pos3,pos4,pos5,pos6,pos7,pos8,pos9,win,
        output wire hsync, vsync,
        output wire [11:0] rgb
    );

    // register for Basys 3 8-bit RGB DAC
    reg [11:0] rgb_reg;

    // video status output from vga_sync to tell when to route out rgb signal to DAC
    wire video_on;
    wire [9:0] x,y;
        // instantiate vga_sync
        vga_sync vga_sync_unit (.clk(clk), .reset(reset), .hsync(hsync), .vsync(vsync),
                                .video_on(video_on), .p_tick(), .x(x), .y(y));

        // rgb buffer
// Counters for horizontal and vertical positions
    reg [3:0] h_counter = 0;
    reg [3:0] v_counter = 0;

```

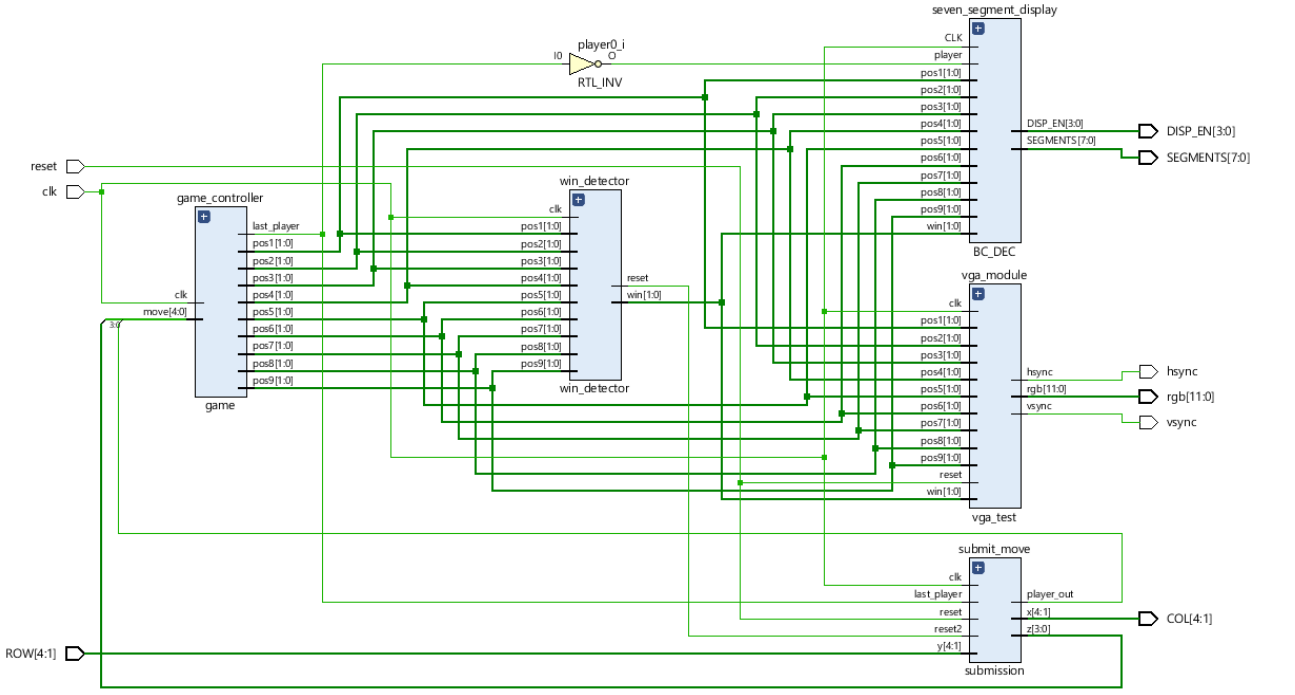
```
     if (reset) begin
         rgb_reg <= 12'b1111_0000_0000;
     end
     else
          begin
              // Update RGB color based on the 3x3 grid position
             if (
             (x >= 10'd260 && x < 10'd270&& y >= 10'd80 && y < 10'd400) //col1
             |
              (x >= 10'd370 && x < 10'd380&& y >= 10'd80 && y < 10'd400)//col2                          //game grid
              |
             (x >= 10'd160 && x < 10'd480&& y >= 10'd180 && y < 10'd190) //row 1
              |
             (x >= 10'd160 && x < 10'd480&& y >= 10'd290 && y < 10'd300) //row 2
             )
                 begin
                     rgb_reg <= 12'b1111_1111_1111; // grid color (white)
                 end
                  else
                     begin
                         // Default color (black)
                         rgb_reg <= 12'b0000_0000_0000; // bg color (black
                     end
```

```
    always @(posedge clk or posedge reset)
    begin
        if (reset)
        begin
            rgb_reg <= 12'b0000_0000_0000;
        end
        else
        begin


     //------------pos1---------------
            if ((x >= 10'd185 && x < 10'd235 && y >= 10'd325 && y < 10'd375&& win==2'b00))
            begin
                if (pos1 == 2'b01)
                begin
                    rgb_reg <= 12'b1111_0000_0000; //  red for player 1
                end
                else if (pos1 == 2'b10)
                begin
                    rgb_reg <= 12'b0000_0000_1111; //blue for player 2
                end
            end
```

4.4 Output Elaborated Diagram

Here is the elaborated diagram of the sample output:



4.5 Output I/O Ports

```
#VGA Connector
set_property PACKAGE_PIN G19 [get_ports {rgb[8]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {rgb[8]}]
set_property PACKAGE_PIN H19 [get_ports {rgb[9]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {rgb[9]}]
set_property PACKAGE_PIN J19 [get_ports {rgb[10]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {rgb[10]}]
set_property PACKAGE_PIN N19 [get_ports {rgb[11]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {rgb[11]}]
set_property PACKAGE_PIN N18 [get_ports {rgb[0]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {rgb[0]}]
set_property PACKAGE_PIN L18 [get_ports {rgb[1]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {rgb[1]}]
set_property PACKAGE_PIN K18 [get_ports {rgb[2]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {rgb[2]}]
set_property PACKAGE_PIN J18 [get_ports {rgb[3]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {rgb[3]}]
set_property PACKAGE_PIN J17 [get_ports {rgb[4]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {rgb[4]}]
set_property PACKAGE_PIN H17 [get_ports {rgb[5]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {rgb[5]}]
set_property PACKAGE_PIN G17 [get_ports {rgb[6]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {rgb[6]}]
set_property PACKAGE_PIN D17 [get_ports {rgb[7]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {rgb[7]}]
set_property PACKAGE_PIN P19 [get_ports hsync]
    set_property IOSTANDARD LVCMOS33 [get_ports hsync]
set_property PACKAGE_PIN R19 [get_ports vsync]
    set_property IOSTANDARD LVCMOS33 [get_ports vsync]

```

5 Control Block

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

5.1 States of our project

1. Start: Represents the state where a new game is initiated. Transition to Player 1's Turn occurs to start the game.
2. Player 1’s Turn: Indicates that it is Player 1's turn to make a move. Transition to Player 2's Turn happens after Player 1 makes a move.
3. Player 2’s Turn: Indicates that it is Player 2's turn to make a move. Transition to Player 1's Turn occurs after Player 2 makes a move.
4. Game Ends: Represents the state when the game is over, either due to a win by one of the players or a draw. Transition to Reset occurs after the game ends.
5. Reset: Represents the initial state or a state reached after the game ends. Transition to Player 1's Turn occurs on the start of a new game.

5.2 State Transition Diagram

Here is the state transition diagram:

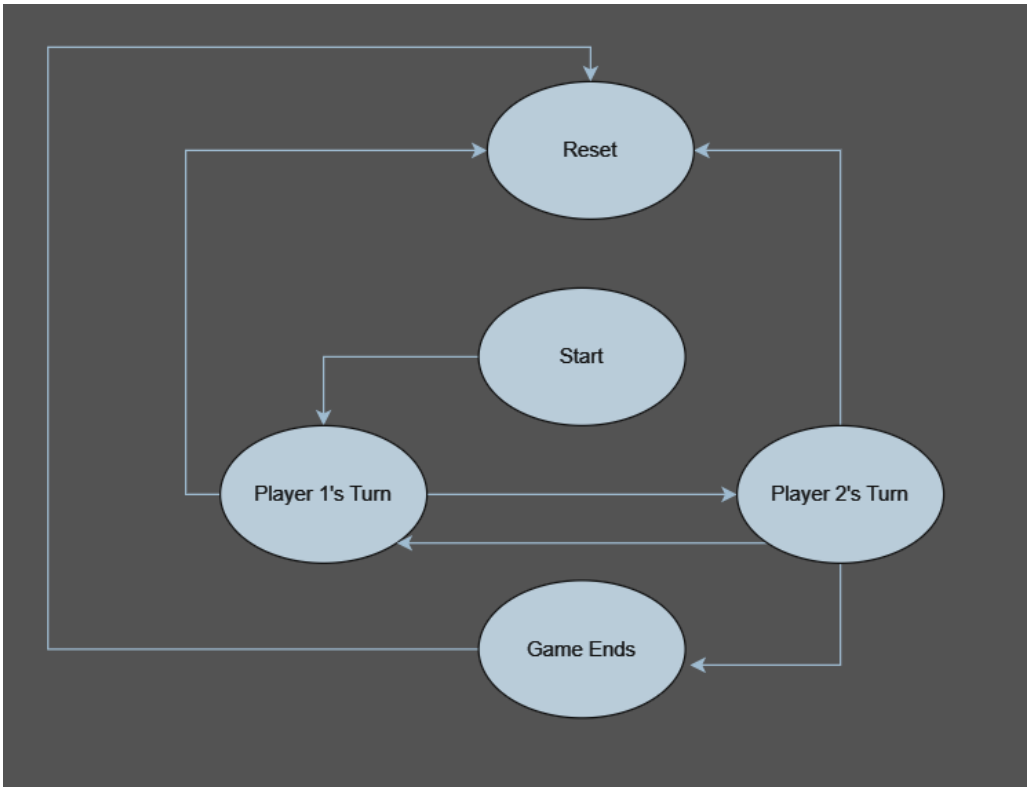


Figure 8

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 in either a win, lose or draw state.

6.2 State Table

State	Logic
Game End	111
Reset	100
Player 1's turn	001
Player 2's turn	010

X=player 1 move, y=player 2 move, z=reset

Current state	Input (x/y/z)	Next state	outputs
001	000	001	0
001	100	010	1
001	010	001	0
001	001	100	1
010	000	010	0
010	100	010	0
010	010	001	1
010	001	100	1
111	000	100	1
111	100	111	0
111	010	111	0
111	001	100	1
100	000	100	0
100	100	010	1
100	010	100	0
100	001	100	0

Figure 9: State table

6.3 Gate level Implementation of FSM

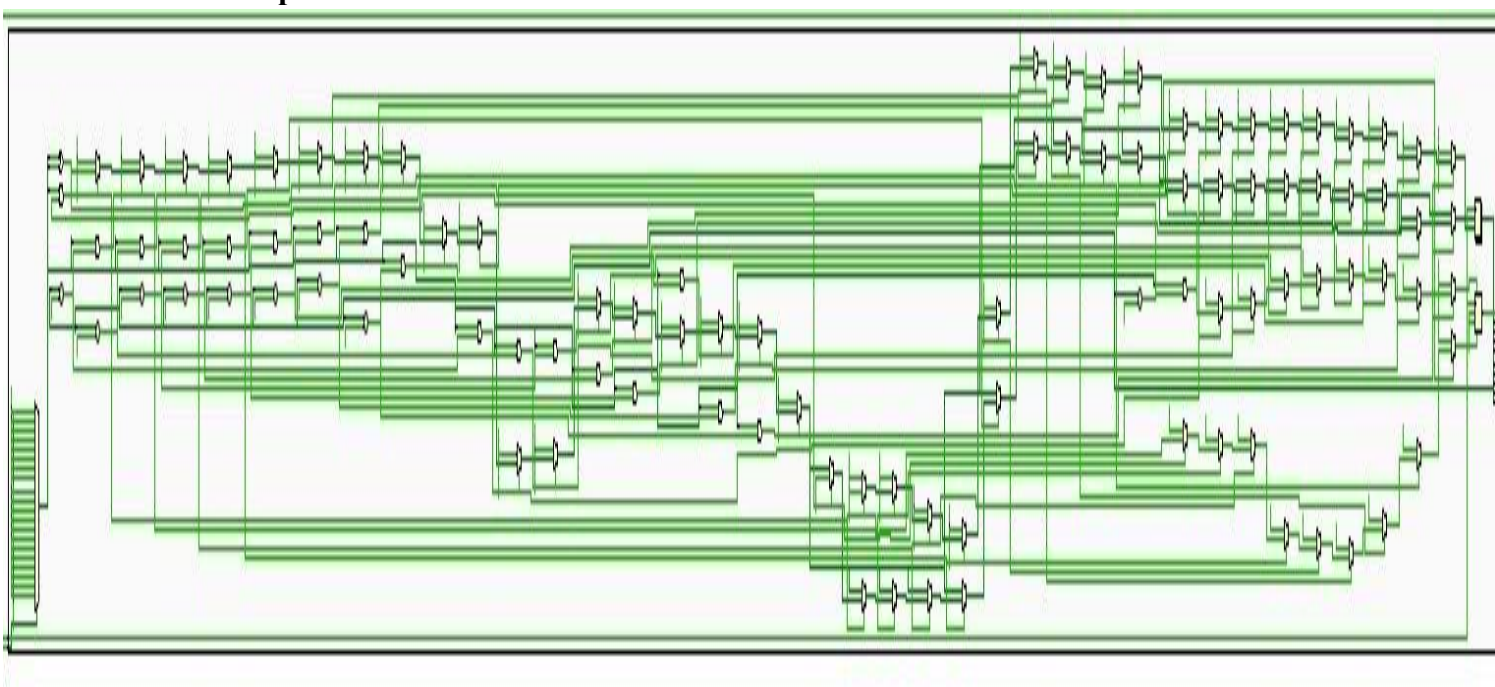


Figure 12: FSM on a gate level

7 Final Game

7.1 Elaborated Design

This is the elaborated design of the game displayed in our RTL schematic. it shows the inputs/outputs going in and out of each module used in our game. The game modules are described more in detail down below under each heading.

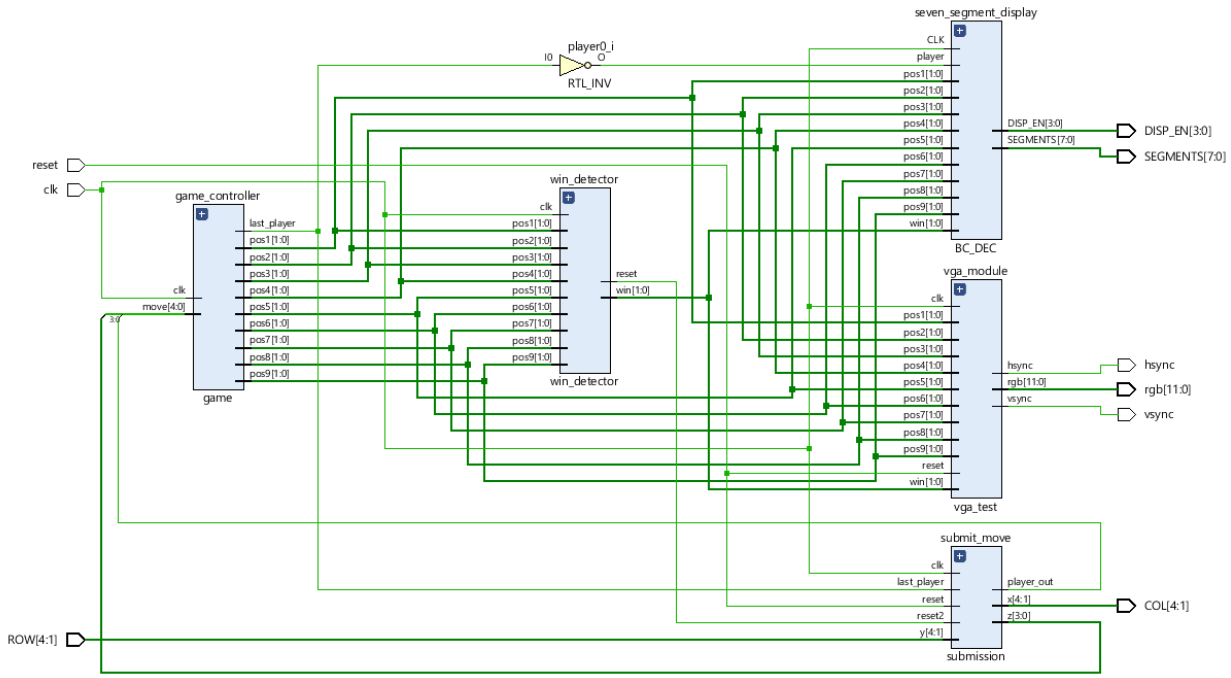


Figure 13: Elaborated design for the game

7.2 Submit Move

This Verilog module implements a clock divider (clk_div3) and a 4-state finite state machine (FSM) for managing the column patterns in a grid-based game. The module takes inputs such as clk for the clock signal, reset for a general reset, reset2 for a specific reset mechanism, and last_player indicating the player who made the last move. Outputs include a 3-bit column position (x), a 3-bit board position (z), and a player indicator (player_out).

The FSM is designed to ensure that the player making the next move is different from the one who made the last move. The states are encoded as follows:

2’b00: idle state

2’b11: Draw state

2'b01 : State for Player 1's turn.

2'b10: State for Player 2's turn.

The module utilizes a clock edge-triggered process to handle state transitions, and a combinational process to determine the next state and update outputs based on the current state.

The state transitions to either 2’b01 or 2’b10 based on the last_player input. During each player's turn, the module updates the column (x), board position (z), and player indicator (player_out) accordingly.

```
module submission(
    input last_player, // last player who made a move
    output [4:1] x,// columns, signal is an output because we are driving them individually
    input [4:1] y,// rows
    input reset, //resets the game
    input reset2, // auto reset signal from the win detector fsm
    output player_out,//outputs the player who's move is being submitted
    input clk,//clock input
    output [3:0] z//output which is used to determine the position on the board being occupied in the next module
    );

reg [3:0] Z;
reg play = 1'b0;
wire sclk;

clk_div3 clk_div(
   .clk(clk),
   .sclk(sclk)
);

reg[1:0] PS, NS;
parameter st_1 = 2'b00, st_2 = 2'b01, st_3 = 2'b10, st_4 = 2'b11;
```

```
PS <= NS;

always@(*)
begin
   begin
      case(PS)
         st_1:begin
            col = 4'b0111;
            NS = st_2;
         end

         st_2:begin
            col = 4'b1011;
            NS = st_3;
         end

         st_3:begin
            col = 4'b1101;
            NS = st_4;
         end

         st_4:begin
            col = 4'b1110;
```

```
            NS = st_1;
         end

         default: begin col = 4'b0111; NS = st_2; end
      endcase
   end
   if (last_player == 1'b0)
      begin
      play = 1'b1;
      end
   else if(last_player == 1'b1)
      begin
      play = 1'b0;
      end
   else
      begin
      play = 1'b0;
      end
end
```

7.3 Game Controller

The game controller module receives two inputs: a clock divider signal (clk_div3) and a 5-bit move input. The move input is generated by the first module and has its most significant bit (MSB) representing the player information. Specifically, when the MSB is 0, it indicates Player 1's move, and when it's 1, it signifies Player 2's move. The remaining four bits convey the details of the player's move.

The module produces several outputs. First, there's last_player, a 1-bit output that indicates whether Player 1 (last_player = 0) or Player 2 (last_player = 1) was the last to make a move. Additionally, there are nine 1-bit outputs (pos1 through pos9) representing individual positions on the game board.

These position outputs store the information about the moves made by players at the corresponding positions on the board. pos1 corresponds to the first position, pos2 to the second, and so on, up to pos9 for the ninth position.

In summary, the game controller module serves as an interface between the clock divider and the move input, storing the details of each player's move in specific positions on the game board. The last_player output provides information about which player made the last move, aiding in the coordination of the game logic.

```
4
5  module game(
6  //input reset,
7  input [4:0] move, //5-bit input receiving the submitted move from the first module the MSB indicates the player(0 == player 1, 1 == p
8  input clk,
9  output last_player,
10 output [1:0] pos1,pos2,pos3,pos4,pos5,pos6,pos7,pos8,pos9//9 position signals corresponding to spaces on the board, these are receive
11 //output [1:0] win
12     );
13 reg [17:0] pos_reg = 18'b000000000000000000;
14 reg last_play;
15 //reg [1:0] win_signal = 2'b00;
16
17 always@(posedge clk)
18     begin
19     if(move == 5'b00001 && pos_reg[1:0] == 2'b00) //translation and storing of moves into registers, will not overwrite space if play
20         begin
21         pos_reg[1:0] = 2'b01; //pos1 <= player1
22         last_play = 1'b0;
23         end
24     else if(move == 5'b00010 && pos_reg[3:2] == 2'b00)
25         begin
26         pos_reg[3:2] = 2'b01; //pos2 <= player1
```

```
27         last_play = 1'b0;
28         end
29     else if(move == 5'b00011 && pos_reg[5:4] == 2'b00)
30         begin
31         pos_reg[5:4] = 2'b01; //pos3 <= player1
32         last_play = 1'b0;
33         end
34     else if(move == 5'b00100 && pos_reg[7:6] == 2'b00)
35         begin
36         pos_reg[7:6] = 2'b01; //pos4 <= player1
37         last_play = 1'b0;
38         end
39     else if(move == 5'b00101 && pos_reg[9:8] == 2'b00)
40         begin
41         pos_reg[9:8] = 2'b01; //pos5 <= player1
42         last_play = 1'b0;
43         end
44     else if(move == 5'b00110 && pos_reg[11:10] == 2'b00)
45         begin
46         pos_reg[11:10] = 2'b01; //pos6 <= player1
47         last_play = 1'b0;
48         end
```

7.4 Win Detector

The win detector module analyzes inputs from pos1 through pos9, each representing a position on a Tic Tac Toe board. It also takes in the clock divider signal (clk_div3). The module's outputs include a 1-bit signal win and a reset signal. The primary function of the win detector module is to determine the winner of the game—either Player 1 or Player 2—or declare a draw if none of the winning conditions are met.

The win signal is set to 1 to indicate a win, and the reset signal is activated to reset the game after a conclusive result is determined.

The module meticulously examines each position on the board, assessing conditions that would lead to a win for either Player 1 (win = 2'b01) or Player 2 (win = 2'b10). If none of these conditions are satisfied, and the game reaches a draw, win is set to 2'b11.

Following the assignment of the final game outcome or draw conditions, the module triggers a reset, bringing the game to its initial state for subsequent plays.

In essence, the win detector module acts as the arbitrator of game results, ensuring fair play and accurately determining the winner or declaring a draw in the context of a Tic Tac Toe game.

```
module win_detector(
input clk,
input [1:0] pos1,pos2,pos3,pos4,pos5,pos6,pos7,pos8,pos9,
output [1:0] win,
output reset
    );
wire finclk;
wire auto_reset;
clk_div4 clk_div4(
.clk(clk),
.sclk(finclk)
);

count_to_five count_to_five(
.clk(finclk),
.in(win),
.reset(auto_reset)
);

reg res;
reg [1:0]  win_signal;
always@(posedge clk)
```

```
    begin
    if (auto_reset == 0)
    begin
    if(pos1 == 2'b01 && pos2 == 2'b01 && pos3 == 2'b01 || //win detection for player 1
        pos4 == 2'b01 && pos5 == 2'b01 && pos6 == 2'b01 ||
        pos7 == 2'b01 && pos8 == 2'b01 && pos9 == 2'b01 ||
        pos1 == 2'b01 && pos5 == 2'b01 && pos9 == 2'b01 ||
        pos7 == 2'b01 && pos5 == 2'b01 && pos3 == 2'b01 ||
        pos1 == 2'b01 && pos4 == 2'b01 && pos7 == 2'b01 ||
        pos2 == 2'b01 && pos5 == 2'b01 && pos8 == 2'b01 ||
        pos3 == 2'b01 && pos6 == 2'b01 && pos9 == 2'b01)
        begin
            win_signal = 2'b01;
            res = 0;
        end
    else if(pos1 == 2'b10 && pos2 == 2'b10 && pos3 == 2'b10 || //win detection for player 2
            pos4 == 2'b10 && pos5 == 2'b10 && pos6 == 2'b10 ||
            pos7 == 2'b10 && pos8 == 2'b10 && pos9 == 2'b10 ||
            pos1 == 2'b10 && pos5 == 2'b10 && pos9 == 2'b10 ||
            pos7 == 2'b10 && pos5 == 2'b10 && pos3 == 2'b10 ||
            pos1 == 2'b10 && pos4 == 2'b10 && pos7 == 2'b10 ||
```

```
            pos2 == 2'b10 && pos5 == 2'b10 && pos8 == 2'b10 ||
            pos3 == 2'b10 && pos6 == 2'b10 && pos9 == 2'b10)
            begin
                win_signal = 2'b10;
                res = 0;
            end
    else if(pos1 > 0 && pos2 > 0 && pos3 > 0 && pos4 > 0 &&
            pos5 > 0 && pos6 > 0 && pos7 > 0 && pos8 > 0 &&
            pos9 > 0 && win_signal != 2'b10 && win_signal != 2'b10) // tie detector
        begin
        win_signal = 2'b11;
        res = 0;
        end

    else//default win signal of 00 which means game is still being played
        begin
        win_signal = 2'b00;
        res = 0;
        end
    end
```

7.5 Seven Segment Display

The "BC_DEC" module serves as a specialized 7-segment display driver tailored for a four-letter word game. It relies on clock dividers, namely "clk_div" and "clk_div2," to create essential signals like "sclk" and "disp_clk." This module effectively multiplexes four seven-segment displays, coordinating with different game states.

Inputs to the module include the clock signal ("CLK"), player positions ("pos1" to "pos9"), the "win" signal, and a player indicator. On the output side, it provides display enable signals ("DISP_EN") and segment data ("SEGMENTS") for the seven-segment displays.

The module is designed to handle diverse game scenarios, ensuring that the information and outcomes are efficiently presented on the four seven-segment displays. Through the use of clock dividers, it precisely controls the display timing, contributing to the overall effectiveness of the word game experience.

```
module BC_DEC(    input CLK,
                  input [1:0] pos1,pos2,pos3,pos4,pos5,pos6,pos7,pos8,pos9,
                  input [1:0] win,
                  input player,
                  output [3:0] DISP_EN,
                  output [7:0] SEGMENTS);


//   -- intermediate signal declaration ----------------------
   reg  [1:0] cnt_dig;
   reg [3:0]   digit;
   reg [7:0] seg = 8'b11111111;
   wire   sclk;
   wire disp_clk;

   clk_div my_clk(.clk(CLK), //calling specific clock divider modules
                  .sclk(sclk));

   clk_div2 clock_for_display(.clk(CLK),
                    .disp_clk(disp_clk));

//   -- advance the count (used for display multiplexing) -----
```

```
  always@ (posedge sclk)
   begin
         cnt_dig <= cnt_dig + 1;
   end

//   -- actuate the correct display --------------------------
   assign DISP_EN = (cnt_dig==0)? 4'b1110:
                    (cnt_dig==1)? 4'b1101:
                    (cnt_dig==2)? 4'b1011:
                    (cnt_dig==3)? 4'b0111: 4'b1111;

   always @ (posedge sclk)

   begin
   seg = 8'b11111111;
   if (win == 0) //will display current game if there is no winner yet
    begin
      case (cnt_dig)
        3:begin // left most seven segment display used to indicate which player is currently making a move
            case (player)
            0:seg <= 8'b10011111;
            1:seg <= 8'b00100101;
```

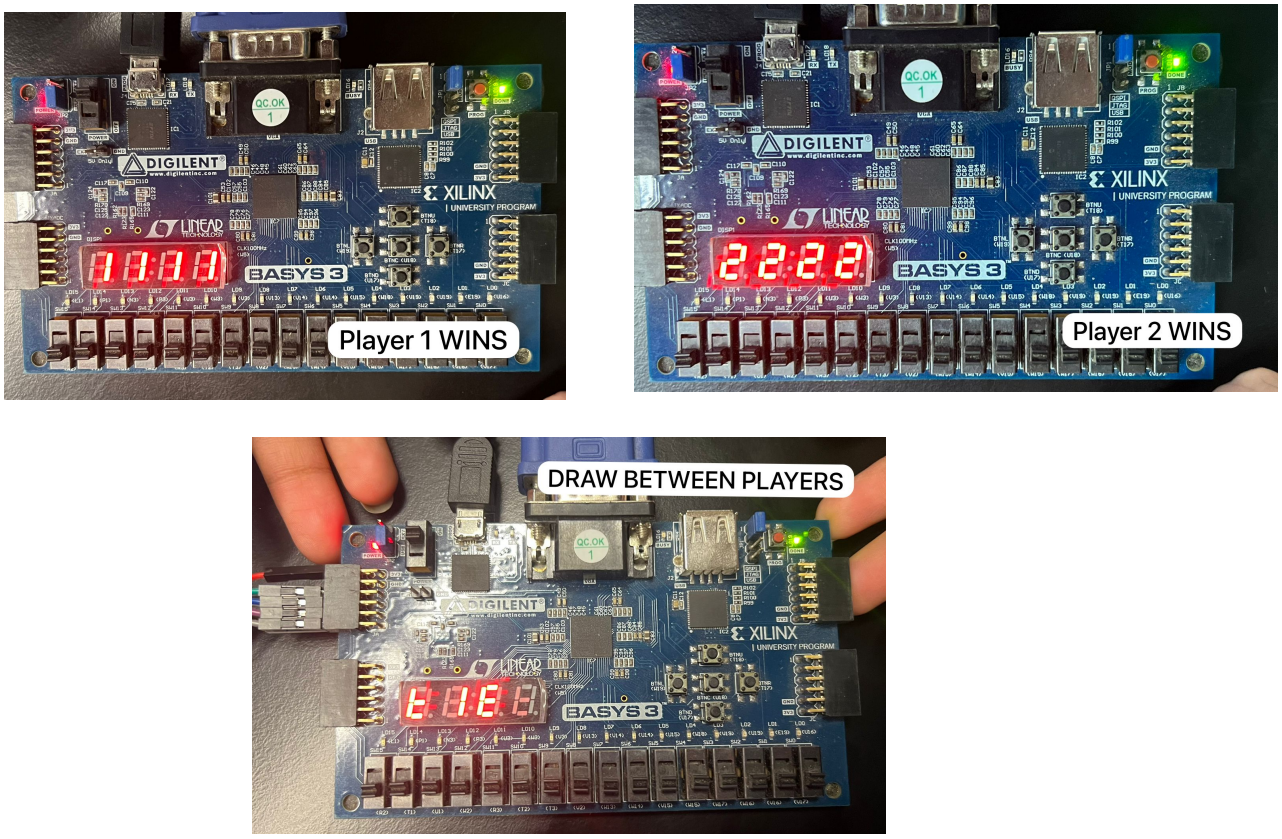


Figure 15: FPGA Display

8 Conclusion

In conclusion, our FPGA Tic Tac Toe game represents a meticulous design and implementation of the classic game within the constraints of hardware logic. This report sheds light on the various modules that constitute our project, emphasizing their roles and interactions.

The user-flow diagram provides a visual representation of the project's structure, illustrating the interconnection of key blocks. Notably, the input block shows the utilization of Input-Output (IO) pins, where player moves are translated through keypad inputs. Each move corresponds to a specific position on the Tic Tac Toe grid, ensuring accurate gameplay.

The control block plays a pivotal role in validating inputs, overseeing state transitions, and ensuring the generation of appropriate outputs. Through a state transition diagram, the control block efficiently manages the game's flow, facilitating a seamless player experience.

The output block takes center stage as the interface through which users interact with the game. It delves into the pixel division of game components, detailing the visual representation of player moves on the grid. In case of a winning scenario or a drawn game, the appropriate display signals are triggered, providing a clear indication of the outcome.

Our Tic Tac Toe game incorporates a reset mechanism. This ensures that if the game reaches a conclusive outcome and then the reset signal is triggered.

In essence, our FPGA Tic Tac Toe game encapsulates thoughtful design, efficient logic utilization, and a user-friendly interface. Through a robust combination of input processing, state management, and output visualization, our project delivers an engaging and responsive Tic Tac Toe gaming experience on the FPGA platform.

9 References

https://www.fpga4student.com/2017/06/tic-tac-toe-game-in-verilog-and-logisim.html?m=1

https://youtu.be/kN8cDM_YuWU?si=-Beb82OBmMiXsRYo

Key scan codes and IO pins: https://digilent.com/reference/programmable-logic/basys-3/start