Modified 4-by-4 tic-tac-toe

Digital logic & design project

Professor Farhan Khan

November 20th,2022

Introduction:

The objective of this final project is to produce a modified version of the well-known "tic-tac-toe" game for two players. Instead of a 3x3 grid, a 4x4 grid will be used in this enhanced version of the game, and the player must complete either a diagonal or a vertical/horizontal pattern. The winner is the player who successfully completes the pattern with 3 consecutive colors that are either diagonal or vertical/horizontal. The game will be a draw if neither player creates a 3-consecutive pattern and the grid is completely filled.

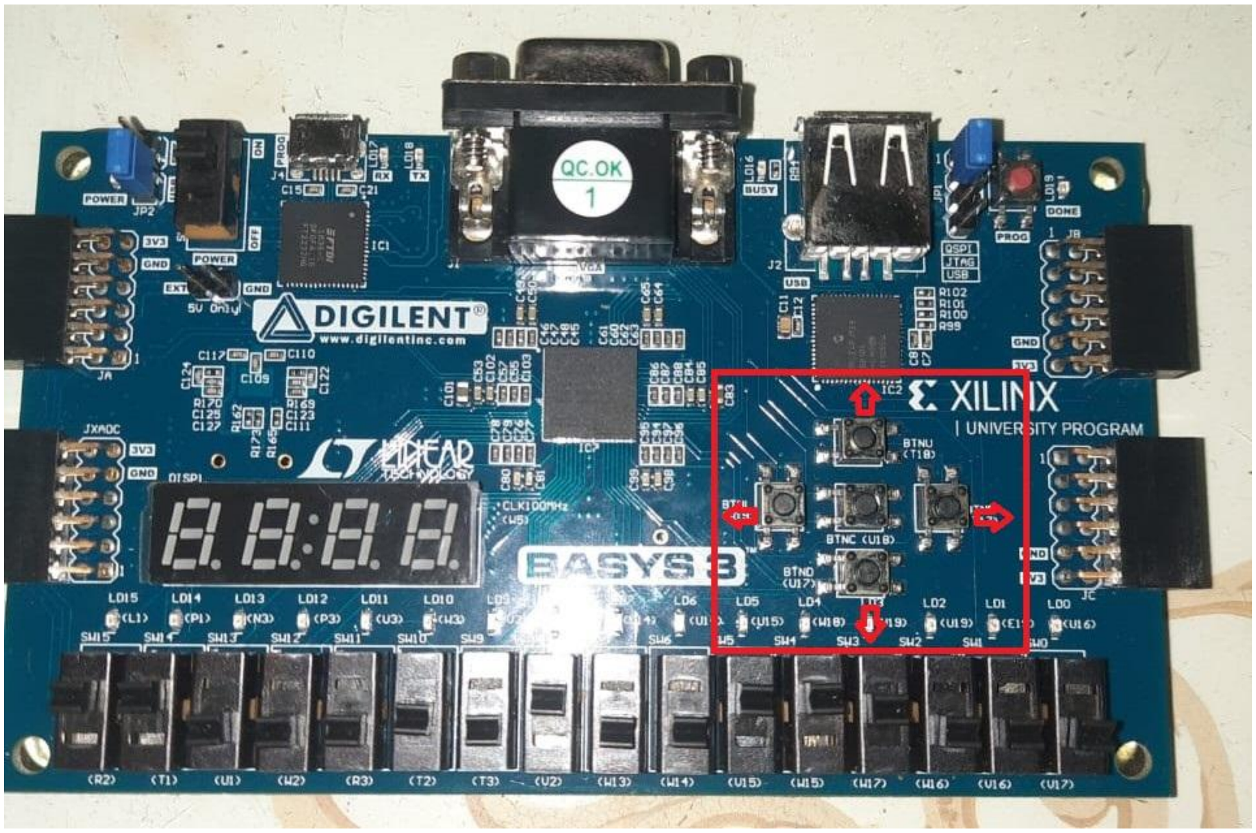
This project was selected because the various elements that the game requires may be properly tied together using digital design concepts. The game's goal is to capture player input through the FPGA board, which will then be shown on a screen. The challenge presented by the project was how to develop a special 4*4 type of tic-tac-toe that required three times as many pattern checking conditions as a 3*3 version while also displaying the complicated grid on the screen. By initially establishing an 8*8 checkerboard grid on the screen and then shrinking it to a 4*4 Checkerboard, we are able to solve the first issue. The second issue was subsequently solved by making a list of every combination that might lead to a win as well as every move that might be considered illegal. The code was then implemented in Verilog after that.

Playing Instructions:

The player uses Basys buttons to navigate and select where to place a mark on the grid. There will be a blue box that will highlight the cell, the player is choosing from. The five Basys buttons will act as left, right, up, down buttons and the center button will be used for selecting the cell of the grid the player will want to place a mark on.

Green will represent ‘O’ or player 1’s marks and Red will represent ‘X’ or player 2’s marks. Whichever player gets three in a row horizontally, vertically or diagonally will win the game. The game will then display either green or red on the entire screen declaring the winner. The game will reset shortly after.

If all cells of the grid are filled, then the game will show blue screen and reset shortly after.

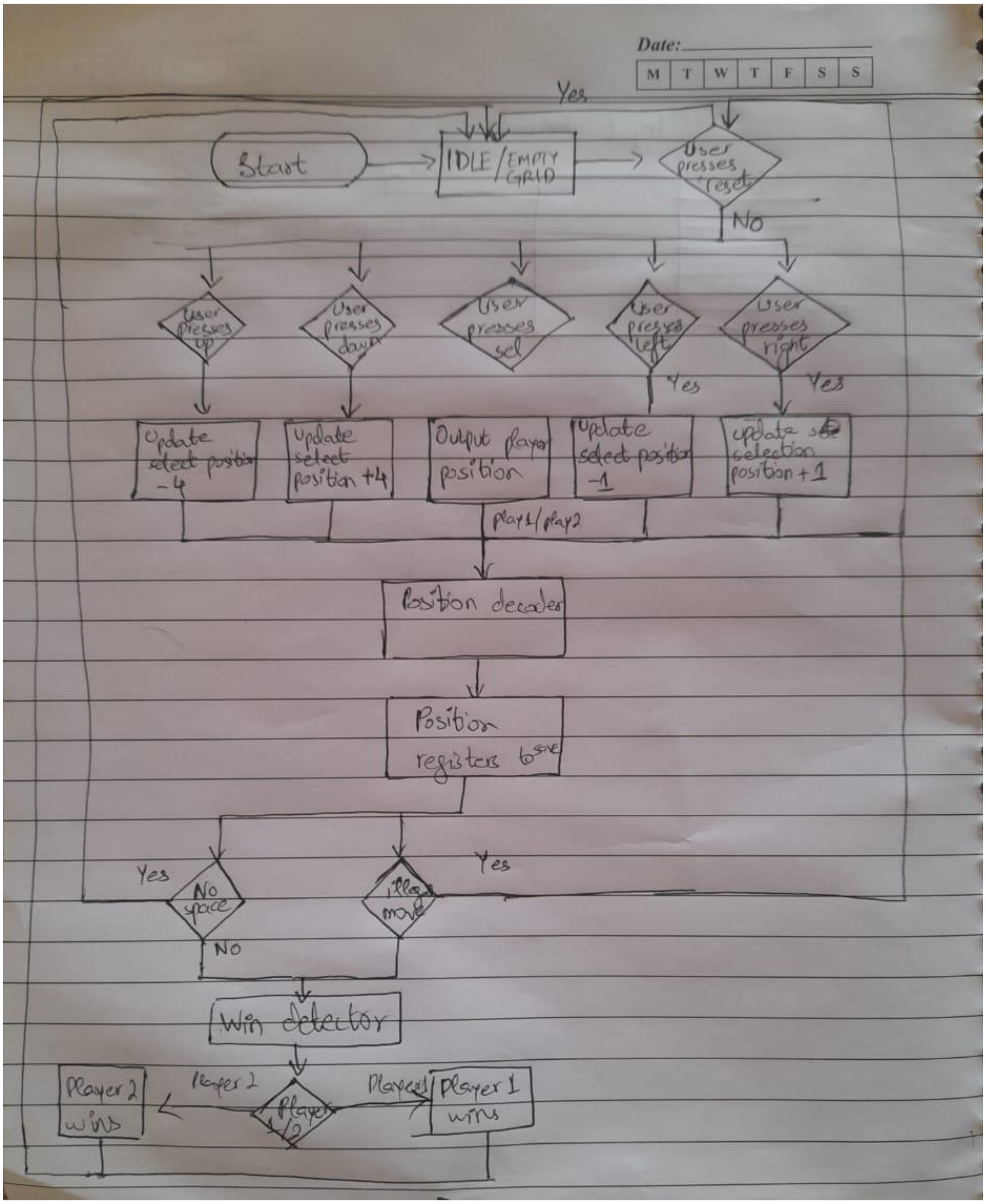


Input Block: Our input block consists of one module only and will take input clock, a single bit reset, one bit signifying turn of player i.e. play1 or play 2, five buttons left, right, up, down and center assigned to FPGA boards. It will output encoded player positions, and also selection position which will be updated in real time on display.

Control Block: Control block consists of 7 modules which will take input 4-bit encoded player-1 and player-2 positions, clock, single bit reset and single bit play1 or play 2. It will output two bit position bits for each cell and the potential winner if there is one.

Output Processing Block: Output block will take input clock and all sixteen two-bit position variables. Each one of these two bit variables will represent a cell on the grid. It will output a checkered display with blue moving boxes as the player moves through the grid, selecting which box to pick. Meanwhile, the

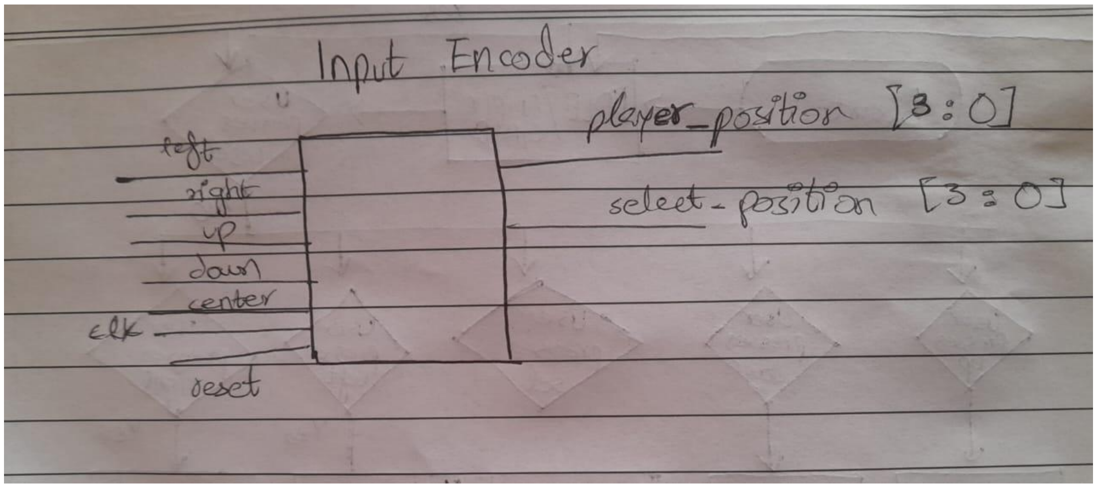
Flow chart:


Date:
M T W T F S S
Yes
Start
IDLE/EMPTY GRID
User presses reset
No
User presses up
User presses down
User presses sel
User presses left
User presses right
Yes
Yes
Update select position -4
Update select position +4
Output player position
Update select position -1
Update selection position +1
play1/play2
Position decoder
Position registers to save
Yes
Yes
No space
illegal move
No
Win detector
Player 2 wins
Player 2
Player 1/2
Player 1
Player 1 wins

Input Block:

Our input block consists of one module only:

1. Input Encoder:



The Input encoder module will be used entirely for translating the buttons pressed on the Basys by the player to an actual position on the grid. The player will be choosing between boxes. The default position encoded will be the 7th position i.e. the center of the grid. This will be encoded in Binary i.e. it will be represented as a four bit number. If the player presses the up or down key, the number will increment /decrement by 4, if the player presses the left or right key, the number will increment or decrement by 1 respectively. There will be conditions to stop the player from selecting beyond the scope of the grid. This selection process will be represented in real time by a blue box in the display.

Once the player has made his selection, the position will be output in a four bit binary number. This number will then be used by the rest of the code.

Output Block:

HCounter:

This module is used to display horizontal lines on the display. It will take in input clock and it will output a 16 bit number. This will be a counter that will count from 0-800. However, the display will only use up to 640 so the range will be altered in the top module.

VCounter:

This module is used to display vertical lines on the display. It is a sequential circuit which will input the clock module, an enable bit and output a 16 bit number. This will be a counter that will count from 0-524. However, the display will only use up to 480 so the range will be altered in the top module.

ClockDivider:

Verilog's clock module operates at 100 mHz frequency. We need to tone the frequency down to the level of the VGA display which is 25mHz. To do this, we will use the formula used by the lab in the code in this component. It is:

(Input Frequency)/(2*desired Frequency) - 1 = 100/(2*25) - 1 = 50-1 = 49.

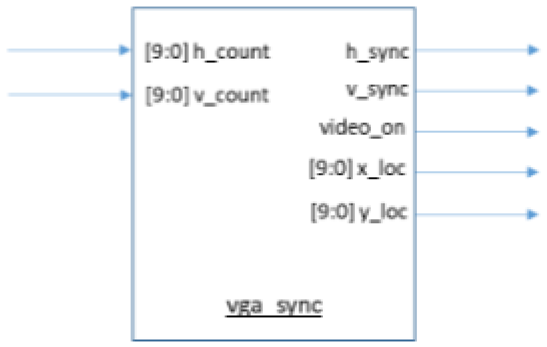
This will be the dividing factor for the code. We will create a separate clock, clock_d, which will only reset after the dividing factor, i.e. 49 number of resets the original clock will go through.

The output will be the clock with 25 Mhz frequency i.e. clock_d.

VGA Sync:

This module will be there to link the hcounter and vcounter together and output x and y locations of each pixel. It will also synchronize the h and v counters and output h_sync and v_sync which will be later used in the Pixel Generator module.

Its block diagram is attached below.



PixelGenerator:

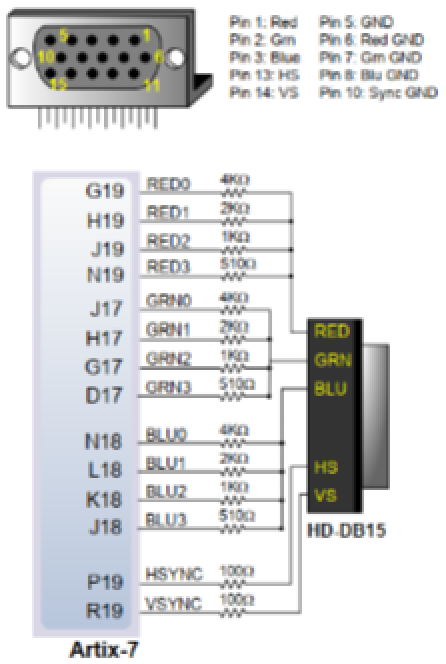
Pixel generator will take input the 4-bit selection position and all the 2-bit positions. It changes the state of the game based on the buttons. It tells which pixel to display what, which it determines from the input two bit positions. We haven't made progress with this module yet, this is why this description is very vague.

Top Display Module:

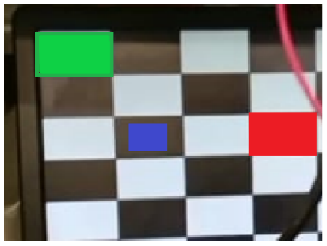
This will be the most complicated module in our project and we are still working on it. Until now, we have gathered that it will input the divided clock, 4-bit selection position, 2 bit tile variables and it will output single bit hsynq, vsynq and 4-bit colours Red, Green and Blue.

The Top Display module will call all of the modules mentioned above in the Output module. It will have conditions for each of the 16 two-bit positions and will render a position into a red or green box with 20 by 20 dimensions approximately depending on Player 1 or Player 2 mark. The selection position will be represented by a 10x10 blue box which will be moving in real time as the player navigates through the Grid.

This can be done using the pixel generator file but it may not be necessary. The pin configuration used will be standard as shown in the diagram below:



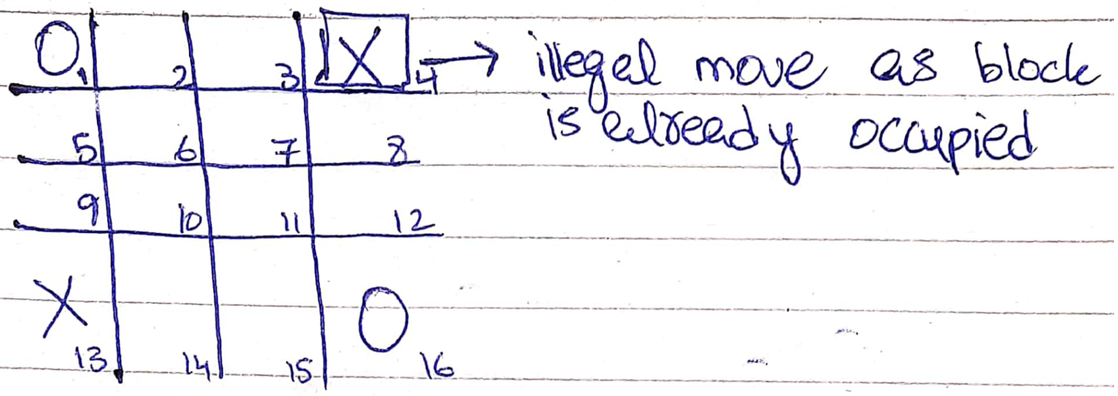
The display will look similar to a checkerboard. But each mark will be represented by Green or Red and selection will be represented by a smaller blue box.



Control Block:

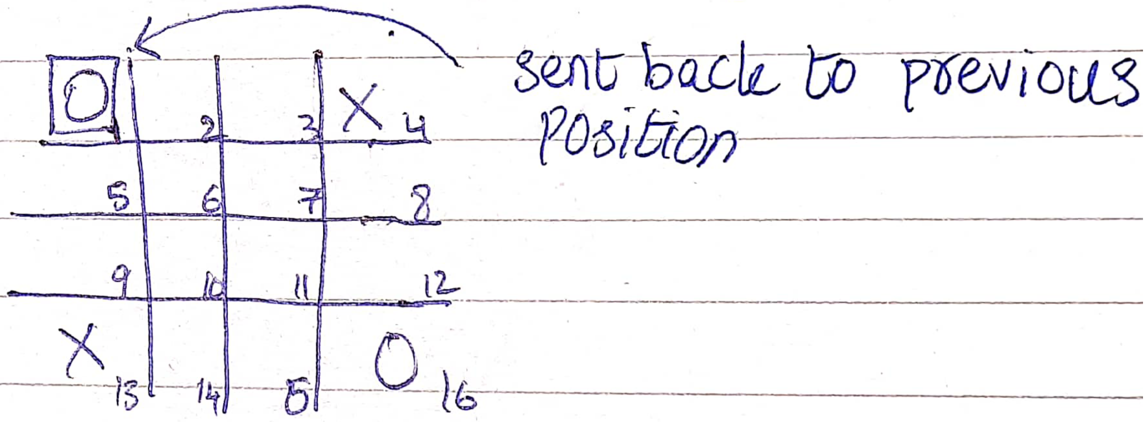
1. Position selection module:

The 16 entry boxes in the 4*4 grid can be chosen by either of the two players using either the green or blue color, as is known. We must be careful not to permit the player to make any improper selections, such as trying to choose an entry box that is already occupied. Our program will automatically shift the player's chosen entry back to its original location if they attempt that illegal move..



The only valid moves in this diagram are positions 2,3,5,6,7,8,9,10,11,12,14,and 15

[illegible]



We are going to record the player's chosen position in this module, and that position will be chosen using the user's preferred color, either green or blue. The inputs that we are going to take in this module are reset (reset of the game), clock (clock of the game), illegal_move (detects an illegal move), Pl1_en (player 1 enable signals), and Pl2_en_ (player 2 enable signals). The outputs for this module will be all the possible 16 entry box on the 4*4 tic-tac-toe grid defined by "pos1,pos2,pos3,pos4,pos5,pos6,pos7,pos8,pos9,pos10,pos11,pos12,pos13,pos14,pos15,pos16".

In this module, we will employ 16 conditions, and each of those 16 conditions will be applied to each of the 16 places. Each condition will look for an illegal move in the selection; if one is found, the selection box will move back to its previous location; otherwise, player 1 or player 2 will carry out the move.

A sample code for this module is given below:

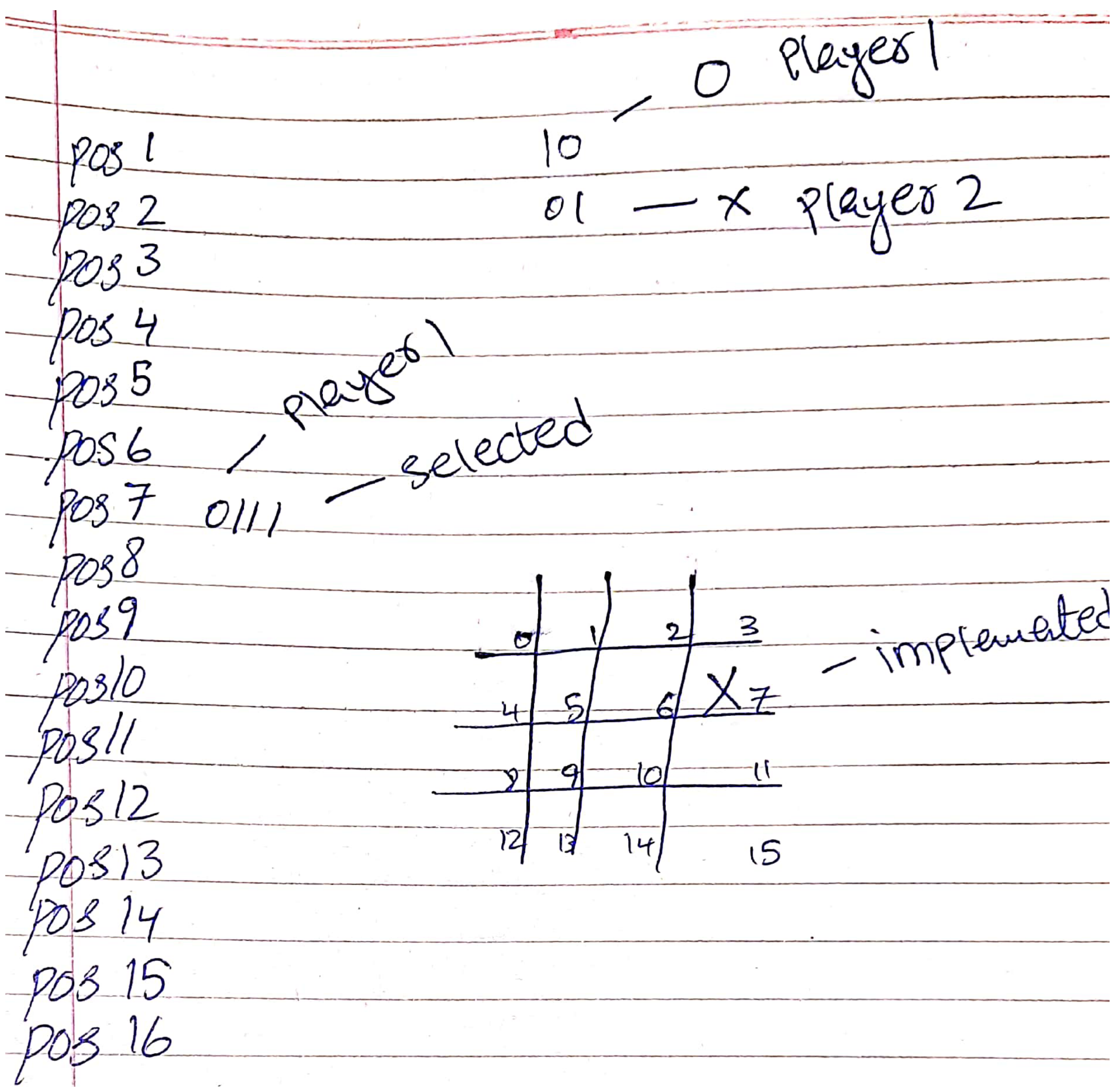
```
module position_selection(

          input clock, // clock of the game

          input reset, // reset the game

          input wrong_move, // disable writing when an illegal move is detected

          input [15:0] PL1_en, // player 1 enable signals

          input [15:0] PL2_en, // Player 2 enable signals

          output reg[1:0] p1,p2,p3,p4,

          p5,p6,p7,p8,p9, p10, p11, p12, p13, p14, p15, p16// positions stored

          );

 // Position 1

 always @(posedge clock or posedge reset)

 begin

  if(reset)

   p1 <= 2'b00;

  else begin

   if(wrong_move==1'b1)

          p1 <= p1;// keep previous position

   else if(PL1_en[0]==1'b1)

    p1 <= 2'b10; // store player 1 data

   else if (PL2_en[0]==1'b1)

          p1 <= 2'b01;// store player 2 data

   else
```

```
            p1 <= p1;// keep previous position
  end
 end
```

In this sample of code, we have shown only 1 condition. Conditions for the other 15 positions will almost be the same with the exception of different positions.

Process:

The box will be encoded to a 4 bit binary digit as soon as the player chooses a box for the entry and it is legal. As an illustration, if player 1 selects position 7 on the grid, position 7 will be encoded into 111 and then saved. The stored binary digit will then be decoded in a decoder after that. All of the places will be mapped according to the enabled bits by this decoder. For instance, if the stored binary was 111, the decoder would enable the seventh position of the grid and, as player 1 had just made a move, it would choose 01 (a cross) for him or her and execute the move on the seventh position of the grid. This procedure continues until the grid fills up.



2. Illegal Move Detector Module

The detection of an illegal move in the illegal_move_detector module works by checking the 2-bit tile variables as well as the position in which the first player or second player are placing their variable. If these positions overlap then the illegal move condition activates.

As an example take a single tile variable as input [1:0] tile1 with each bit representing which player placed their symbol within it. If either player has done this prior then it would be an illegal move for either to attempt to place a symbol again within this tile.Therefore an or statement will be needed to specify this condition. This can be written as:

(tile[1]| tile[0])

The first player and second player will be the 16 bit [15:0] PC_player and UR_player respectively,with each bit representing one of the possible tiles they may choose to place their symbol in. For this example we can assume that PC_player[0] and UR_Player[0] both correspond to tile1. Separate statements can be written for each case. In the case of the first player the statement could be written by attaching an and condition to the previous code:

illegal1=(tile[1]| tile[0])& PC_player[0]

The first 16 illegal conditions for the computer can be aggregated into a single statement:

PC_illegal=(((illegal1| illegal2)|illegal3)| illegal4)......... and so on for all 16 conditions.

For the second player, the condition can be written as the following:

illegal17=(tile[1]|tile[0]) & UR_Player[0]

Thes conditions can also be aggregated into a separate variable which may be written as:

UR_illegal=(((illegal17| illegal18)| illegal19)| illegal20)....and so on for the next sixteen conditions

Together, these can be aggregated for illegal_move ouput.

illegal_move=UR_illegal | PC_illegal

3. Draw Detector Module

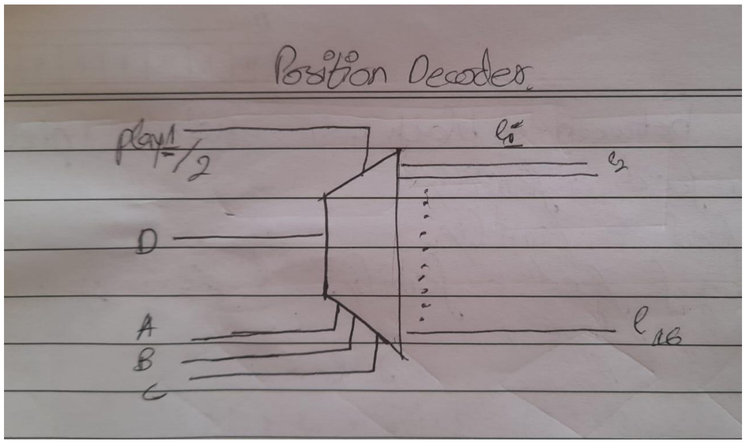
The draw_detector works by checking the two bit tile variable and using that to generate an output no_space. As stated earlier each bit of the tile represents one of the players having placed a symbol within the tile. Therefore either bit being active indicates that the tile is full. This can be written with an or statement for each tile.

tfull1= tile[1] | tile[0]

Sixteen such statements can be written with one for each tile. These can be aggregated to get the condition for a draw. That condition is when all the tiles are full yet there is no winner assigned.

no_space =((tfull1 & tfull2) & tfull3)...... and so on.

4. Position Decoder:



When the player finally selects the position, the Position Decoder Module converts the player 's selected position and returns the output position as per grid.

Inputs:

Enable Bit – Enable Bit will work as active high and it enables the decoder when Enable Bit is 1, otherwise, disables it.

# of Bits in Enable Bit = 1

Position Input – Position Input will contain user selected position in 4 bits. For example, if user selects tile no ‘10’ on grid, then the position input will be ‘1010’.

# of Bits in Position Input = 4

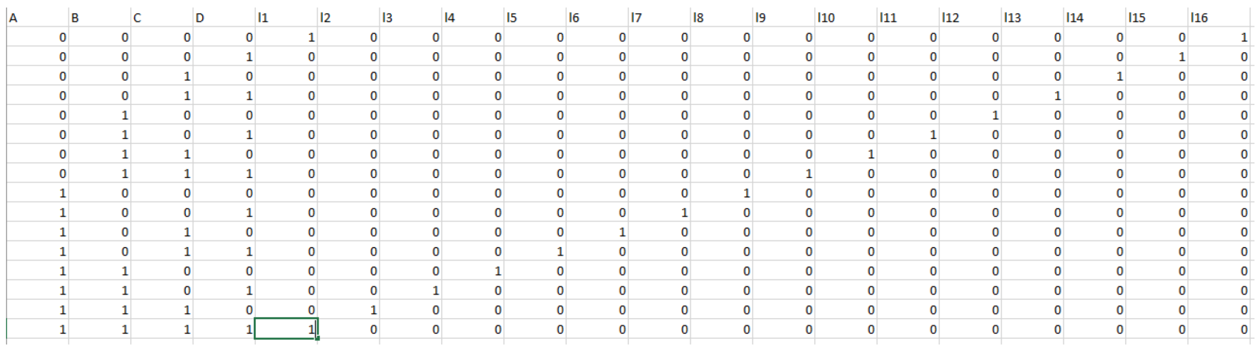
Outputs:

Decoded Position – It is a 16 bits binary number. All bits of Decoded Position will be ‘0’ except the nth bit where nth bit will be decided as per position input decimal value. For example, if position input is ‘1010’, then it’s ‘10’ in decimal. Hence, the 10th LSB bit will be ‘1’.

e.g ‘000001000000000’

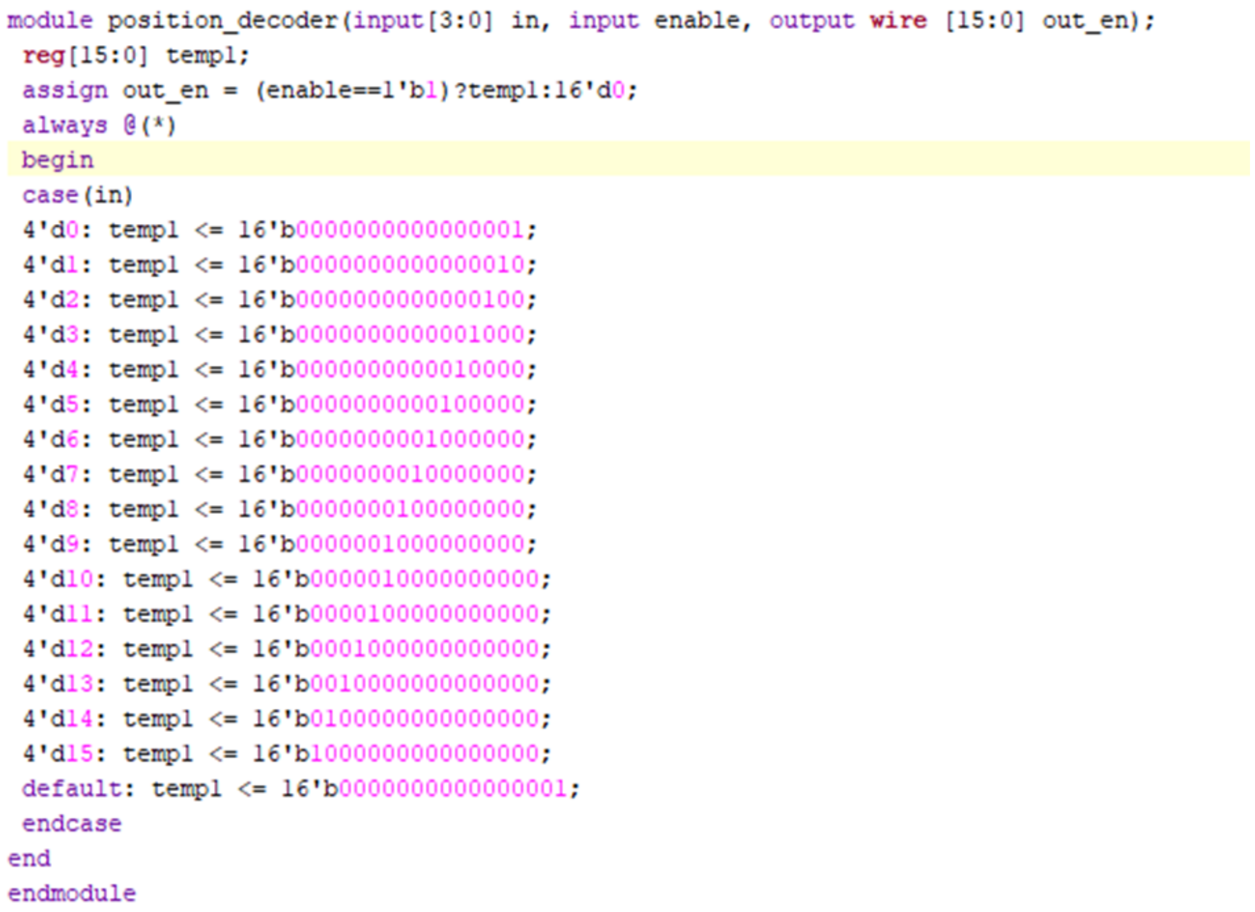
# of Bits in Decoded Position = 16

Truth Table:



A	B	C	D	I1	I2	I3	I4	I5	I6	I7	I8	I9	I10	I11	I12	I13	I14	I15	I16
0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
0	1	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
0	1	1	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
1	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
1	0	1	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
1	1	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Code Snippet:



```
module position_decoder(input[3:0] in, input enable, output wire [15:0] out_en);
 reg[15:0] templ;
 assign out_en = (enable==1'b1)?templ:16'd0;
 always @(*)
 begin
 case(in)
 4'd0: templ <= 16'b0000000000000001;
 4'd1: templ <= 16'b0000000000000010;
 4'd2: templ <= 16'b0000000000000100;
 4'd3: templ <= 16'b0000000000001000;
 4'd4: templ <= 16'b0000000000010000;
 4'd5: templ <= 16'b0000000000100000;
 4'd6: templ <= 16'b0000000001000000;
 4'd7: templ <= 16'b0000000010000000;
 4'd8: templ <= 16'b0000000100000000;
 4'd9: templ <= 16'b0000001000000000;
 4'd10: templ <= 16'b0000010000000000;
 4'd11: templ <= 16'b0000100000000000;
 4'd12: templ <= 16'b0001000000000000;
 4'd13: templ <= 16'b0010000000000000;
 4'd14: templ <= 16'b0100000000000000;
 4'd15: templ <= 16'b1000000000000000;
 default: templ <= 16'b0000000000000001;
 endcase
end
endmodule
```

5. Win Detector:

Win Detector Module checks for every possible winning combination of current grid after every valid change in position. If any player wins the match, he will be assigned as the winner.

1.1 Inputs:

All Position Inputs – It will contain 16 inputs. These 16 inputs are 2 bits position inputs and contains three states: “X” , “O” and the other one is undefined (i.e when the tile is not filled because it is possible that tiles may not be filled and any player won the game).

# of Bits in Position Inputs Bit = 2

Outputs:

Winner – Return ‘1’ if anyone is winner else returns ‘0’

# of Bits in Winner = 1

WinnerID – Since there are two players: Player 1 and Player 2. The Player 1 ID is ‘01’ and Player 2 ID is ‘10’. It returns any of the Player ID who wins the game.

# of Bits in WinnerID = 2

Code Snippet:

```
module winner_detector(input [1:0] p1,p2,p3,p4,p5,p6,p7,p8,p9, output wire winner, output wire [1:0]who);

wire win1,win2,win3,win4,win5,win6,win7,win8,win9;

wire [1:0] who1,who2,who3,who4,who5,who6,who7,who8, who9;



winner_detect_3 u1(p1,p2,p3,win1,who1);// (1,2,3);

winner_detect_3 u2(p4,p5,p6,win2,who2).......
```

This will keep going to check for each position

Where p1, p2, p3……..are positions for each tile in 4x4 tic tac toe grid.

6. Win Detect

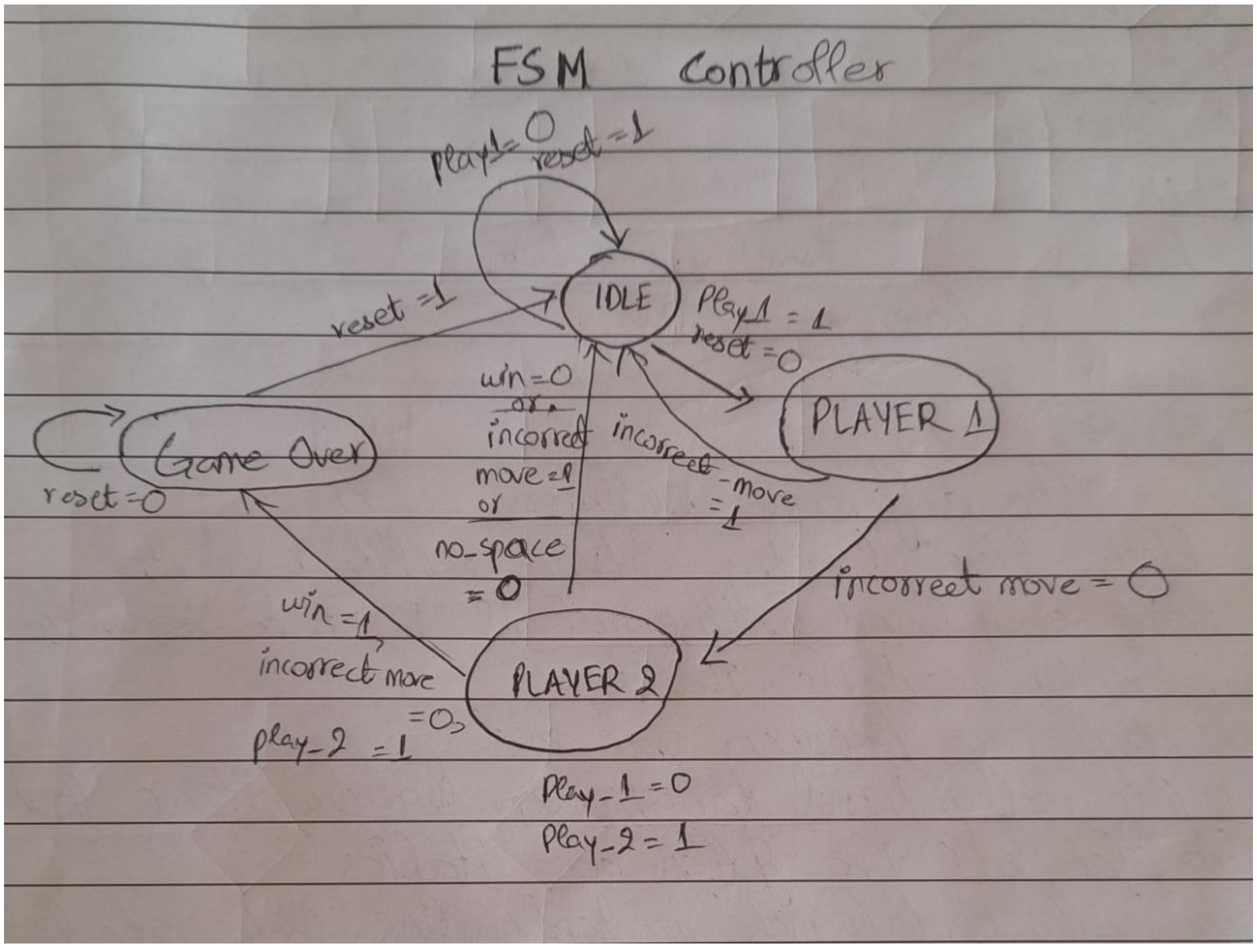
Win Detect is a simple module that is used to simplify the code of the Win Detector module. Its input is three 2-bit positions and 2-bit winner. The positions are three because you need three in a row to win this game. The positions are all the ones in which winning is possible whether vertically, horizontally or diagonally. This module checks whether these three positions have the same 2 bit number or not. If all of them are either 10 or 01, then the program can declare player 1/player 2 the winner respectively.

The code snippet is very simple:

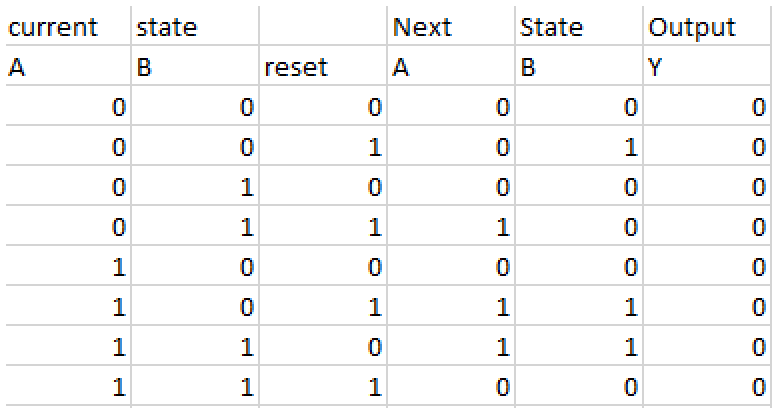
Code snippet:

```
module winner_detect_3(input [1:0] pos0,pos1,pos2, output wire winner, output wire [1:0]who);

wire [1:0] temp0,temp1,temp2;

wire temp3;

assign temp0[1] = !(pos0[1]^pos1[1]);

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];

assign temp2[0] = temp0[0] & temp1[0];

assign temp3 = pos0[1] | pos0[0];

// 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];
```

7. FSM Controller



We are using a mealy machine as a Finite Machine Satate controller for this game. According to the diagram above, there are four states: IDLE, PLAYER1, PLAYER2, GAME OVER. These states are encoded by 2-bit binary numbers. This will be used in the game to simulate different phases of the game whether its the first player's turn, the second player's, the game is over etc. The transition table is given below:



current	state		Next	State	Output
A	B	reset	A	B	Y
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	0
0	1	1	1	0	0
1	0	0	0	0	0
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	0	0	0

Code snippet:

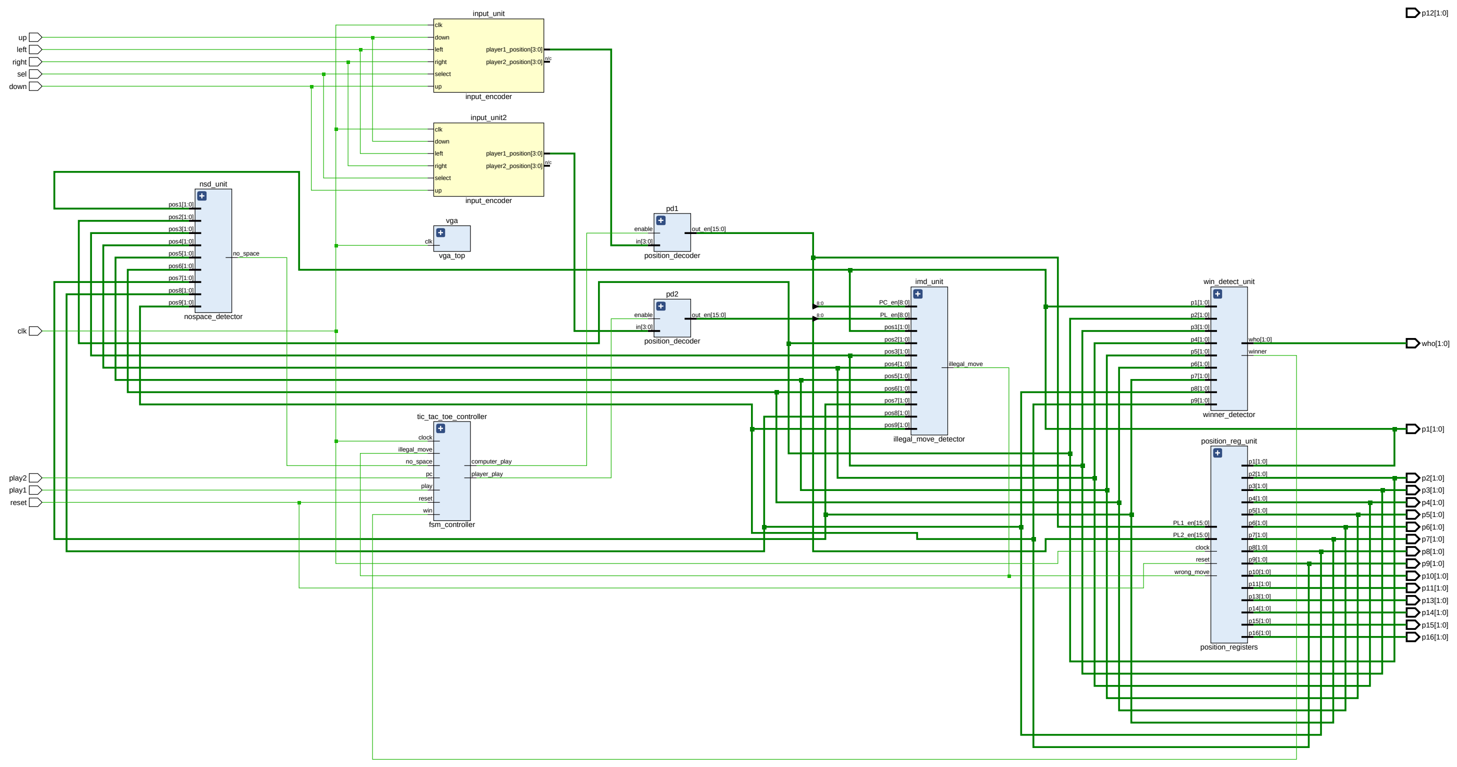
```
parameter IDLE=2'b00;

parameter PLAYER=2'b01;

parameter COMPUTER=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;
 computer_play <= 1'b0;
end
PLAYER:begin
 player_play <= 1'b1;
 computer_play <= 1'b0;
 if(illegal_move==1'b0)
  next_state <= COMPUTER; // computer to play
 else
  next_state <= IDLE;
end
```

8. Top module for tic-tac-toe:

In order to execute the game, we will put all the preceding modules together in one top module. The inputs for this module are clk (the game's clock), reset (the game's reset), play1 (the input for player1), play2 (the input for player2), and left, right, up, down, sel (the last of which is used to pick the option pointed to using the left, right, up, and down arrows).

The modules that are going to be combined in this top module will be position selector, FSM (Finite State Machine) controller, space detector, wrong move detector, position decoder, and 2 winner detector modules.This will be the schematic block diagram for the entire circuit with all blocks:



input_unit
clk
down
left
right
select
up
player1_position[3:0]
player2_position[3:0]
input_encoder
input_unit2
clk
down
left
right
select
up
player1_position[3:0]
player2_position[3:0]
input_encoder
up
left
right
sel
down
nsd_unit
pos1[1:0]
pos2[1:0]
pos3[1:0]
pos4[1:0]
pos5[1:0]
pos6[1:0]
pos7[1:0]
pos8[1:0]
pos9[1:0]
no_space
nospace_detector
vga
clk
vga_top
pd1
enable
in[3:0]
out_en[15:0]
position_decoder
pd2
enable
in[3:0]
out_en[15:0]
position_decoder
clk
imd_unit
PC_en[8:0]
PL_en[8:0]
pos1[1:0]
pos2[1:0]
pos3[1:0]
pos4[1:0]
pos5[1:0]
pos6[1:0]
pos7[1:0]
pos8[1:0]
pos9[1:0]
illegal_move
illegal_move_detector
tic_tac_toe_controller
clock
illegal_move
no_space
pc
play
reset
win
computer_play
player_play
fsm_controller
play2
play1
reset
win_detect_unit
p1[1:0]
p2[1:0]
p3[1:0]
p4[1:0]
p5[1:0]
p6[1:0]
p7[1:0]
p8[1:0]
p9[1:0]
who[1:0]
winner
winner_detector
position_reg_unit
PL1_en[15:0]
PL2_en[15:0]
clock
reset
wrong_move
p1[1:0]
p2[1:0]
p3[1:0]
p4[1:0]
p5[1:0]
p6[1:0]
p7[1:0]
p8[1:0]
p9[1:0]
p10[1:0]
p11[1:0]
p12[1:0]
p13[1:0]
p14[1:0]
p15[1:0]
p16[1:0]
position_registers
p12[1:0]
who[1:0]
p1[1:0]
p2[1:0]
p3[1:0]
p4[1:0]
p5[1:0]
p6[1:0]
p7[1:0]
p8[1:0]
p9[1:0]
p10[1:0]
p11[1:0]
p13[1:0]
p14[1:0]
p15[1:0]
p16[1:0]