Tic-Tac-Toe using Verilog

Rohan (2020CSB1117)
Raghav Patidar (2020CSB1115)

Teacher - **Dr. Neeraj Goel**

INTRODUCTION

- **Tic-tac-toe** is a paper-and-pencil game for two players who take turns marking the spaces in a 3X3 grid with X or O.
- The player who succeeds in placing three of their marks in a horizontal, vertical, or diagonal row is the winner. It is a solved game, with a forced draw assuming best play from both players.
- In our game, the player plays the Tic Tac Toe game with a computer. When the player/ computer plays the game, a 2-bit value is stored into one of the nine positions in the 3x3 grid. It stores :
- **2'b00** - when neither the player or computer played in that position.
- **2'b01** (X) - when the player played in the position
- **2'b10** (O) - when the computer plays in the position.

The winners is announced as a 2 bit value, where :

- **2'b01** -> Player won
- **2'b10** -> Computer won

For the implementation part, several modules are created which detects illegal moves, decodes position, stores values at positions,

EXAMPLE :

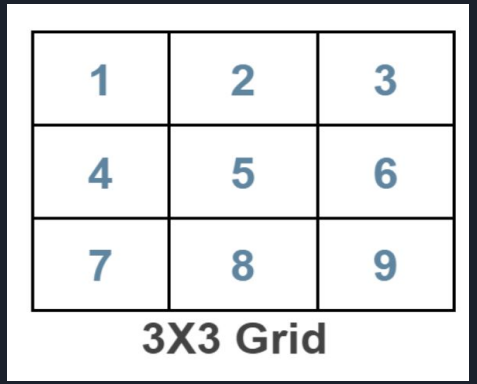
Let us understand it better using an example :

Consider a 3X3 grid as shown in figure 1 :

- When a player plays, a 2 bit value gets stored at the position he wants to mark his symbol. In our implementation, we defined 00 for ideal state, 01 for player and 10 for computer.
- So, initially the 3X3 matrix is initialized with 00 states in all positions.
- The player/ computer wins the game when successfully placing three similar (01-Xs) or (10-Os) values in the following pairs of positions:

  row{(1,2,3), (4,5,6), (7,8,9)} or column{(1,4,7), (2,5,8);(3,6,9)} or diagonal{(1,5,9);(3,5,7)}.

- It is important to keep in mind illegal moves and space left in the 3X3 grid.
- Winner is announced as 2 bit value, which can be : 00 , 01 or 10 .



3X3 Grid

CASE I - Computer (10) Wins

In the first case, we took an example where the computer is winning the match.

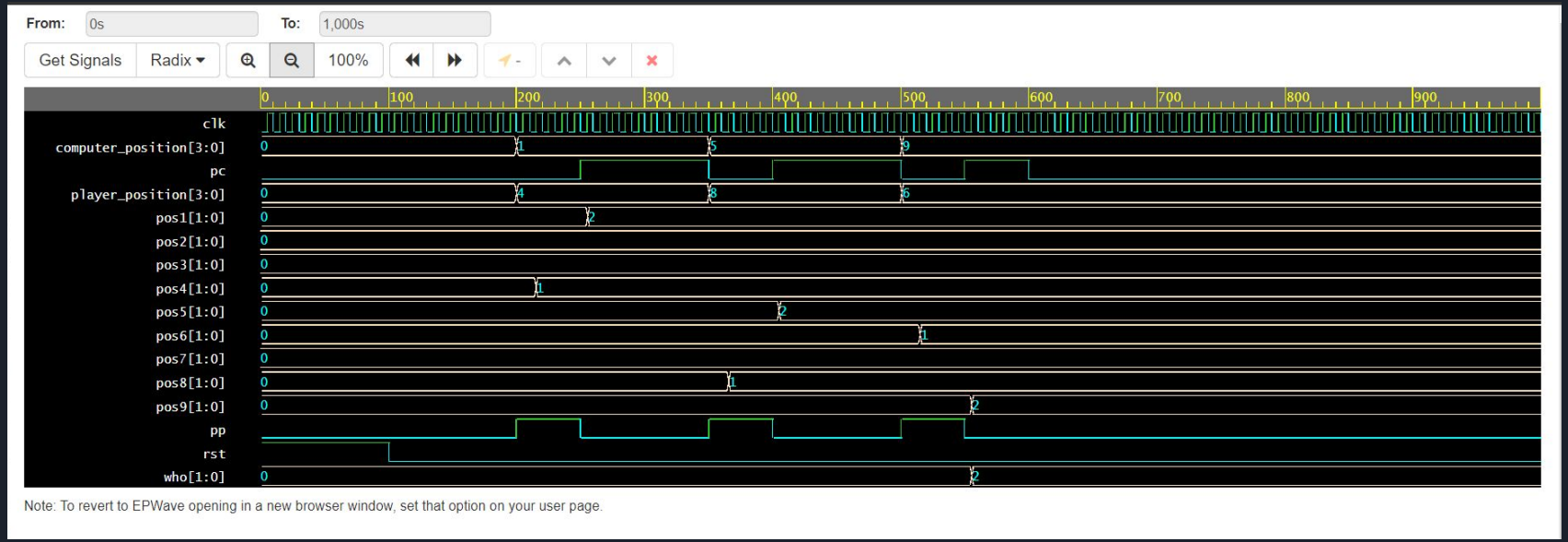
At last, we will be able to see the final 3X3 matrix generated and the output.

Winner : 10 .

Below is the EP wave generated :

10	00	00
01	10	01
00	01	10

Case I (Winner : 10)



CASE II - No one (00) Wins

In the second case, we took an example where no one is winning the match.

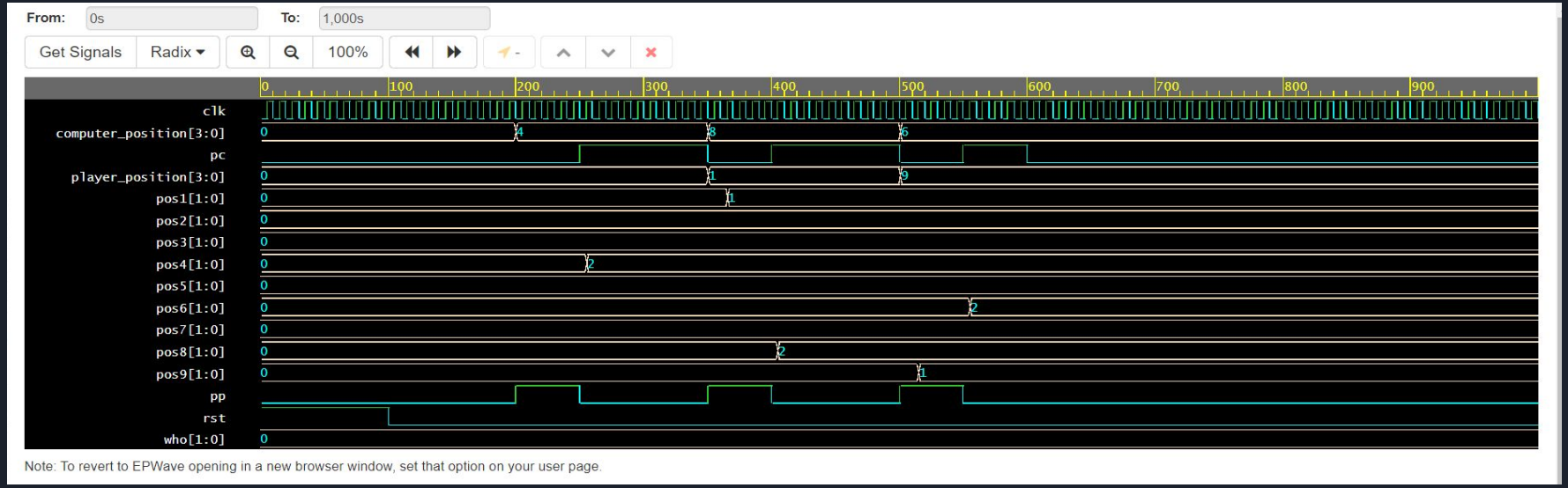
At last, we will be able to see the final 3X3 matrix generated and the output.

Winner : 00

Below is the EP wave generated :

01	00	00
10	00	10
00	10	01

Case II (Winner : 00)



CASE III - Player (01) Wins

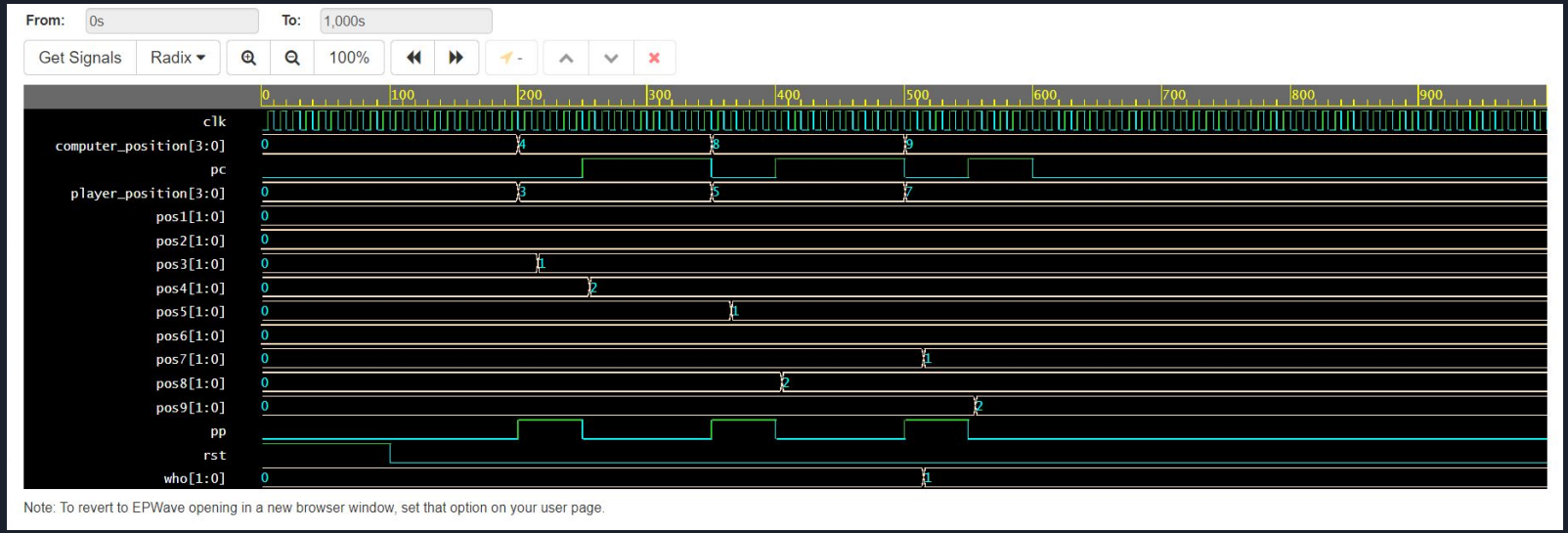
In the third case, we took an example where the player is winning the match.

At last, we will be able to see the final 3X3 matrix generated and the output **Winner : 01**

Below is the EP Wave generated :

00	00	01
10	01	00
01	10	10

Case III (Winner : 01)



CODE ANALYSIS FOR DESIGN

```
// winner detection for 3 positions and determine who the winner is
// Player: 01
// Computer: 10

module winner_detect_across_line(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];
endmodule
```

Input : [1:0] pos1, pos2, pos3
Output : winner, [1:0]who
Variables : temp3, [1:0]temp0, temp1, temp2

winner_detect_across_line

This module is for checking if winner is present in the three input positions.
At first we store XOR of all positions in temp2.
temp2 = pos1^pos2^pos3
If all three positions are equal, then temp2 will be 11.
We created another variable temp3 equal to OR of pos1[0] and pos[1]
temp3 = pos1[0] | pos1[1]
Finally winner = 1 if temp2 is 11 and temp3 = 1 , else 0.
winner = temp3 & temp2
So, temp3 was to make winner = 0 if all three positions are 00.
Finally who is calculated by AND of winner with pos.
Who[0] = winner & pos1[0]
Who[1] = winner & pos1[1]

winner_detector

```
// winner detector circuit
// to detect who the winner is
// We will win when we have 3 similar (x) or (0) in the following pairs:
// (1,2,3); (4,5,6);(7,8,9); (1,4,7); (2,5,8);(3,6,9); (1,5,9);(3,5,7);

module winner_detector(input [1:0] pos1,pos2,pos3,pos4,pos5,pos6,pos7,pos8,pos9, output
wire winner, output wire [1:0]who);
wire win1,win2,win3,win4,win5,win6,win7,win8;
wire [1:0] who1,who2,who3,who4,who5,who6,who7,who8;

winner_detect_across_line u1(pos1,pos2,pos3,win1,who1);// (1,2,3);
winner_detect_across_line u2(pos4,pos5,pos6,win2,who2);// (4,5,6);
winner_detect_across_line u3(pos7,pos8,pos9,win3,who3);// (7,8,9);
winner_detect_across_line u4(pos1,pos4,pos7,win4,who4);// (1,4,7);
winner_detect_across_line u5(pos2,pos5,pos8,win5,who5);// (2,5,8);
winner_detect_across_line u6(pos3,pos6,pos9,win6,who6);// (3,6,9);
winner_detect_across_line u7(pos1,pos5,pos9,win7,who7);// (1,5,9);
winner_detect_across_line u8(pos3,pos5,pos7,win8,who8);// (3,5,7);

assign winner = (((((((win1 | win2) | win3) | win4) | win5) | win6) | win7) | win8);
assign who = (((((((who1 | who2) | who3) | who4) | who5) | who6) | who7) | who8);
endmodule
```

Input : [1:0] pos1, pos2, pos3
Output : winner, [1:0]who

This module is for deciding the winner of game. It is implemented by instantiating module **winner_detector_across_line .**

We check for winner in all possible rows, columns and diagonals.
win1, win2 ... win8 represents if there is a winner in a particular row, column or diagonal.
Final winner will be **OR** of win1 to win8.

who(i), where 1<=i<=8, can be 00 or 10 or 01, depending on winner found or not.
Winner, if present will be in one of who(i), so final winner is **OR** from who1 to who8

nospace_detector

```
// NO SPACE detector
// to detect if no more spaces to play
module nospace_detector(
   input [1:0] pos1,pos2,pos3,pos4,pos5,pos6,pos7,pos8,pos9,
   output wire no_space
    );
wire t1,t2,t3,t4,t5,t6,t7,t8,t9;
// detect no more space
assign t1 = pos1[1] | pos1[0];
assign t2 = pos2[1] | pos2[0];
assign t3 = pos3[1] | pos3[0];
assign t4 = pos4[1] | pos4[0];
assign t5 = pos5[1] | pos5[0];
assign t6 = pos6[1] | pos6[0];
assign t7 = pos7[1] | pos7[0];
assign t8 = pos8[1] | pos8[0];
assign t9 = pos9[1] | pos9[0];
// output
assign no_space =(((((((t1 & t2) & t3) & t4) & t5) & t6) & t7) & t8) & t9);
endmodule
```

Input: [1:0] pos1, pos2 .. pos9
Output: no_space

LOGIC:
Position is stored as a 2 bit value , where :
10 is for computer,
01 is for player,
00 is for empty.
If we take **OR** of this two bits if it comes out to be 0 it means both the bits are 0 . Hence the position is empty.
Therefore space is available.
The output in this case no_space is 0.
Else position are occupied , and output is equals to 1.

position_registers

```
module position_registers(
      input clk, // clock of the game
      input rst, // reset the game
      input illegal_move, // disable writing when an illegal move is detected
  input [9:0] PC_en_signal, // Computer enable signals
  input [9:0] PL_en_signal, // Player enable signals
      output reg[1:0] pos1,pos2,pos3,pos4,pos5,pos6,pos7,pos8,pos9// positions stored
      );

 // Position 1
  always @(posedge clk or posedge rst)
 begin
  if(rst)
   pos1 <= 2'b00;
  else begin
   if(illegal_move==1'b1)
    pos1 <= pos1;// keep previous position
    else if(PC_en_signal[1]==1'b1)
    pos1 <= 2'b10; // store computer data
    else if (PL_en_signal[1]==1'b1)
    pos1 <= 2'b01;// store player data
   else
    pos1 <= pos1;// keep previous position
  end
 end
```

Logic:
At first we check if the reset is equals to 1. If yes, then we initialise position to 00 and else we check for the illegal move. **If** it equals to 1 then it keeps the previous position, **else if** it will check the PC enable signal at index 1 and if that equals to 1, then we store the computer data or player data, **else** it will keep the previous position.

Similarly we do it for all other positions too.

Input: clk , rst , illegal_move, pc _en_signal, pl_en_signal
Output: pos1 , pos2……pos3

illegal_move_detector

```
module illegal_move_detector(
    input [1:0] pos1,pos2,pos3,pos4,pos5,pos6,pos7,pos8,pos9,
   input [9:0] PC_en_signal, PL_en_signal,
    output wire illegal_move
    );
wire t1,t2,t3,t4,t5,t6,t7,t8,t9,t11,t12,t13,t14,t15,t16,t17,t18,t19,t21,t22;

// player : illegal moving

   assign t1 = (pos1[1] | pos1[0]) & PL_en_signal[1];
   assign t2 = (pos2[1] | pos2[0]) & PL_en_signal[2];
   assign t3 = (pos3[1] | pos3[0]) & PL_en_signal[3];
   assign t4 = (pos4[1] | pos4[0]) & PL_en_signal[4];
   assign t5 = (pos5[1] | pos5[0]) & PL_en_signal[5];
   assign t6 = (pos6[1] | pos6[0]) & PL_en_signal[6];
   assign t7 = (pos7[1] | pos7[0]) & PL_en_signal[7];
   assign t8 = (pos8[1] | pos8[0]) & PL_en_signal[8];
   assign t9 = (pos9[1] | pos9[0]) & PL_en_signal[9];

// computer : illegal moving
   assign t11 = (pos1[1] | pos1[0]) & PC_en_signal[1];
   assign t12 = (pos2[1] | pos2[0]) & PC_en_signal[2];
   assign t13 = (pos3[1] | pos3[0]) & PC_en_signal[3];
   assign t14 = (pos4[1] | pos4[0]) & PC_en_signal[4];
   assign t15 = (pos5[1] | pos5[0]) & PC_en_signal[5];
   assign t16 = (pos6[1] | pos6[0]) & PC_en_signal[6];
   assign t17 = (pos7[1] | pos7[0]) & PC_en_signal[7];
   assign t18 = (pos8[1] | pos8[0]) & PC_en_signal[8];
   assign t19 = (pos9[1] | pos9[0]) & PC_en_signal[9];
// intermediate signals
assign t21 =(((((((t1 | t2) | t3) | t4) | t5) | t6) | t7) | t8) | t9);
assign t22 =(((((((t11 | t12) | t13) | t14) | t15) | t16) | t17) | t18) | t19);

// output illegal move
assign illegal_move = t21 | t22 ;
endmodule
```

Input: clk , rst , illegal_move, pc _en_signal, pl_en_signal
Output: pos1 , pos2…...pos3

LOGIC:

If Player enable signal or computer enable signal for that particular position is equals to 1 and any of two bits of that particular position is 1 then it simply means it is trying to overwrite the existing status and hence it is illegal move.
So we declared some variables and used appropriate AND and OR gates to define the logic
Similarly we check for all positions , separately for computer illegal move and player illegal move.

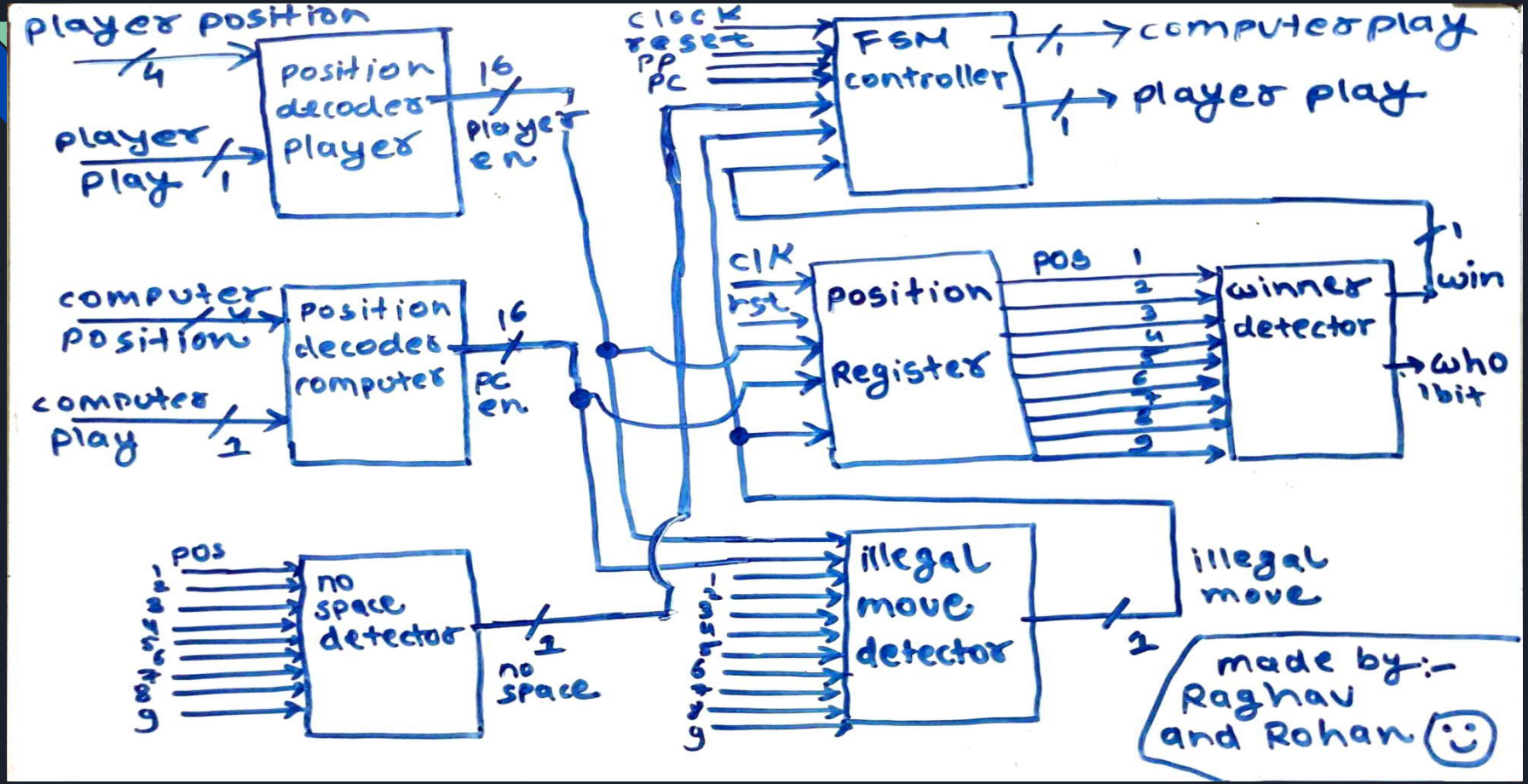
position_decoder

```
module position_decoder(input[3:0] in, input enable, output wire [15:0] out_en);

 reg[15:0] t1;
 assign out_en = (enable==1'b1)?t1:16'd0;
 always @(*)
 begin
 case(in)
 4'd0: t1 <= 16'b0000000000000001;
 4'd1: t1 <= 16'b0000000000000010;
 4'd2: t1 <= 16'b0000000000000100;
 4'd3: t1 <= 16'b0000000000001000;
 4'd4: t1 <= 16'b0000000000010000;
 4'd5: t1 <= 16'b0000000000100000;
 4'd6: t1 <= 16'b0000000001000000;
 4'd7: t1 <= 16'b0000000010000000;
 4'd8: t1 <= 16'b0000000100000000;
 4'd9: t1 <= 16'b0000001000000000;
 4'd10: t1 <= 16'b0000010000000000;
 4'd11: t1 <= 16'b0000100000000000;
 4'd12: t1 <= 16'b0001000000000000;
 4'd13: t1 <= 16'b0010000000000000;
 4'd14: t1 <= 16'b0100000000000000;
 4'd15: t1 <= 16'b1000000000000000;
 default: t1 <= 16'b0000000000000001;
 endcase
end
endmodule
```

To decode the position being played, a 4-to-16 decoder needed.
When a button is pressed, a player will play and the position at IN [3:0] will be decoded to enable writing to the corresponding registers.
It helps to create player and computer enable signals.
For every decimal position there is unique 16 bit signal which helps to check the position to be played.

ILLUSTRATIONS



CODE ANALYSIS FOR TESTBENCH

- Here, we took 3 possible cases through which we can see that our design tells whether the player wins, the computer wins or it’s an incomplete match.
- First of all we instantiated the **test_bench** module with the main module of design. Then, we took three possible matches between players.
- Initially the clock cycle is defined using the forever statement.
- Player enable signals (denoted by pp) and Computer enable signals (denoted by pc) are enabled/disabled based on whose turn is going on.
- Positions of computer and player are registered by computer_position and player_position respectively.
- We have nine 2 bit positions represented by pos1, pos2, ... pos9, which are updated whenever any of the players register their positions. Initially all the positions are 00.
- Example : If any player or computer registers position 6, then pos6 is updated to 01(for player) or 10(for computer).

CODE ANALYSIS FOR TESTBENCH Continue....

- Reset button (denoted by rst) is for re-initializing the 3X3 grid to make all positions equal to 00. So, we made rst=1 at starting and rst is maintained 0 throughout a particular match.

```
// Initialize Inputs
pp = 0;
rst = 1;
computer_position = 0;
player_position = 0;
pc = 0;
```

Here we initialize all variables to zero except reset i. Rst =1 . It means game is not started yet.

```
rst = 0;
#100;
pp = 1;
pc = 0;
computer_position = 1;
player_position = 4;
#50;
pc = 1;
pp = 0;
```

Now we make rst=0 . Here the game begins. pp=1 means player chance is enabled and pc=0 means computer chance is disabled. Player plays on position 4. Here computer play (pc) plays on position 1 but it does not get registered in position because computer chance to play is disabled. After this when pc =1 and pp =0 , at this point computer position is registered .

CODE ANALYSIS FOR TESTBENCH Continue....

```
rst = 0;
pp = 1;
pc = 0;
computer_position = 5;
player_position = 8;
#50;
pc = 1;
pp = 0;
```

Now we make rst=0 , here game begins. pp=1 means player chance is enabled and pc=0 means computer chance is disabled. Player plays on position 8. Here computer play (pc) plays on position 5, but at this point of time it does not get registered in position because computer chance to play is disabled. After this when pc =1 and pp =0 , at this point computer position is registered .

```
rst = 0;
pp = 1;
pc = 0;
computer_position = 9;
player_position = 6;
#50;
pc = 1;
pp = 0;
```

Now we make rst=0 , here game begins. pp=1 means player chance is enabled and pc=0 means computer chance is disabled. Player plays on position 6. Here computer play (pc) plays on position 9, but at this point of time it does not get registered in position because computer chance to play is disabled. After this when pc =1 and pp =0 , at this point computer position is registered .

CODE ANALYSIS FOR TESTBENCH continue....

```
  pc = 0;
  pp = 0;
   $display("If winner is 01, it means player won, else if winner is
10, it means computer won , else Winner is not decided yet. \n");
   $display("Below is 3X3 matrix generated after first match :\n");
   $display(" %b %b %b \n",pos1,pos2,pos3);
   $display(" %b %b %b \n",pos4,pos5,pos6);
   $display(" %b %b %b \n",pos7,pos8,pos9);
   $display("Winner Decided is : %b \n",who);
```

10	00	00
01	10	01
00	01	10

Case I (Winner : 10)

At this point of time for checking result we makes player chance pp and computer chance pc to zero.
And after that we display grid and the winner.
This is the brief analysis of test bench of one case , similarly we explain for other tests cases too.

APPLICATIONS

- Because of the simplicity of tic-tac-toe, it is often used as a pedagogical tool for teaching the concepts of good sportsmanship and the branch of artificial intelligence that deals with the searching of game trees.
- It contributes to children's developmental growth in numerous ways including their understanding of predictability, problem solving, spatial reasoning, hand-eye coordination, turn taking, and strategizing.

CONCLUSIONS

- Tic-Tac-Toe is a 2 player game in which both players mark their symbols over a 3X3 grid turn by turn.
- The player who makes the first three of their marks in a diagonal, vertical, or horizontal row wins the game.
- Because of the simplicity of tic-tac-toe, it is often used as a pedagogical tool for teaching the concepts of good sportsmanship.
- This game contributes to children's developmental growth in numerous ways including their understanding of predictability, problem solving, spatial reasoning, hand-eye coordination, turn taking, and strategizing.

THANK YOU

Created by :
Rohan 2020CSB1117
Raghav Patidar 2020CSB1115