CS203 PROJECT

TIC TAC TOE GAME IMPLEMENTATION USING VERILOG

COURSE INSTRUCTOR:

Dr Neeraj Goel

TEAM MEMBERS:

Tanish Goyal (2020CSB1133)

Vijay Dwivedi (2020CSB1140)

OBJECTIVE: To explain the reader the implementation and functionalities of the Project

PROJECT OVERVIEW AND CONCEPT:

We have implemented the tic tac toe in Verilog using various modules. It is a two-player game in which the user plays with the computer. We have made it very much similar to the conventional 3X3 block tic tac toe game. The player who fills a side or a diagonal first will be the winner. Further, to process the turns, a code (2 bit) is assigned to the players and that code fills a position if that position is marked by that player.

The following 2-bit values will be stored in one of the 9 positions. (All are random bit numbers)

00 – if neither player and computer played in that position [REST STATE]

01 – represents **X**. If player played at that position [PLAYER STATE]

10 – represents **0**. If computer played at that position [COMPUTER STATE]

11-game is finished [OVER STATE]

1	2	3
4	5	6
7	8	9

The player or computer wins the game if 3 similar X/0 are put in the following rows:

(1,2,3), (4,5,6), (7,8,9) [HORIZONTAL]

(1,4,7), (2,5,8), (3,6,9) [VERTICAL]

(1,5,9), (3,5,7) [DIAGONAL]

Also, the functionality of wrong move detection is implemented in addition to the normal modules. Testbench is given for a working example.

VERILOG IMPLEMENTATION (MODULES):

1. Module Tic_Tac_Toe:

INPUT:

- clock
- reset
- play
- comp
- 4 bit computer
- 4 bit player

OUTPUT: 2 bit positions that are

- pos1
- pos2
- pos3
- pos4
- pos5
- pos6
- pos7
- pos8
- pos9
- 2 bit winner

It’s the main module that uses all the submodules made in the code. The use of various submodules is ordered.

```
module Tic_Tac_Toe(clock, reset, play, comp, computer, player, pos1, pos2, pos3, pos4, pos5, pos6, pos7, pos8, pos9, winner);
    input clock, reset, play, comp;
    input [3:0] computer , player;
    output wire [1:0] pos1,pos2,pos3,pos4,pos5,pos6,pos7,pos8,pos9;
    output wire[1:0] winner;

  wire [15:0] c_enable,p_enable;// enable signal for computer and the player

 wire wrong_move; //whenever wrong move is played, the program will terminate

 wire cp; //turn of computer
 wire pp; //turn of player
 wire filled;
 wire win;
```

```
position p_reg(clock, reset, wrong_move, c_enable, p_enable, pos1, pos2, pos3, pos4, pos5, pos6, pos7, pos8, pos9);  // function calling 'position'

  who_wins who_pc(pos1,pos2,pos3,pos4,pos5,pos6,pos7,pos8,pos9,win,winner);   //finds out the winner
  pos_decode c_dec(computer,cp,c_enable); //returns the position of computer
  pos_decode p_dec(player,pp,p_enable); //returns the position of player
  re_block error_pc( pos1,pos2,pos3,pos4,pos5,pos6,pos7,pos8,pos9,c_enable[8:0], p_enable[8:0],wrong_move);  //checks for wrong moves

 finish f_no_space(pos1,pos2,pos3,pos4,pos5,pos6,pos7,pos8,pos9,filled);  // whenever all positions are filled
 FSM control(clock,reset,play,comp,wrong_move,filled,win,cp,pp);

endmodule
```

2. Module Position:

INPUT: 16 bits enables i.e.

- c_enable
- p_enable

OUTPUT: 2 bit positions that are

- pos1
- pos2
- pos3
- pos4
- pos5
- pos6

- pos7
- pos8
- pos9

This module is used to store the positions of player and computer (2nd player)
We have done this for all of the 9 positions. It has main use when the FSM uses it. For each of the 9 positions, we made the code such that, when the player's present data or the computer's present data should be stored or the previous data should be kept.

```
module position(clock, reset, wrong_move, c_enable, p_enable, pos1, pos2, pos3, pos4, pos5, pos6, pos7, pos8, pos9);  //positions
    input [15:0] c_enable,p_enable;
    input clock, reset, wrong_move;

     output reg[1:0] pos1, pos2, pos3, pos4, pos5, pos6, pos7, pos8, pos9;

//posedge checks at the edges when clock/reset changes its value
 always @(posedge clock or posedge reset)   //for the 1st position
 begin
  if(reset)
   pos1 <= 2'b00;
  else begin
   if(wrong_move==1'b1)
    pos1<=pos1;                    // stores the old position
   else if(c_enable[0]==1'b1)
    pos1<= 2'b10;                   // stores the computer's position
   else if(p_enable[0]==1'b1)
    pos1<= 2'b01;                     // stores the player's position
   else
    pos1 <= pos1;                     //stores the old position
  end
 end
```

3. Module FSM:

INPUT:

- Clock
- Reset
- Play
- Comp
- wrong_move
- filled
- win

OUTPUT:

- c
- pp

We have implemented the state machine in this module. This module is used to control the tic tac toe gameplay.
The FSM module contains 4 **parameters** (NET DEFINITION: A parameter is an attribute of a Verilog HDL module that can be altered for each instantiation of the module. These attributes represent constants, and are often used to define variable width and delay value).
The 4 parameters or states are as follows :

1. Rest = 2’b00
2. Computer = 2’b10
3. Player = 2’b01
4. Over = 2’b11

```
module FSM(clock,reset,play,comp,wrong_move,filled,win,cp,pp);    //implementation of state machine, controls the overall gameplay
    input clock,reset,play,comp,wrong_move,filled,win;
    output reg cp,pp;

parameter REST=2'b00;       //rest state
parameter COMPUTER=2'b10;   //computer's turn
parameter PLAYER=2'b01;     //player's turn
parameter OVER=2'b11;       //game is over/finished
reg[1:0] old,new;           //previous and next states
```

Now, we implement each of the states (or parameters):

- REST:

```
REST:
begin
 if(reset==1'b0 && play == 1'b1)
  new<= PLAYER; // player's turn
 else
  new<= REST;
 pp <= 1'b0;
 cp <= 1'b0;
end
```

- COMPUTER:

```
COMPUTER:
begin
 pp<= 1'b0;
 if(comp==1'b0) begin
  new <= COMPUTER;
  cp<= 1'b0;
 end
 else if(filled == 1 || win ==1'b1)
 begin
  new<=OVER;      //game over
  cp<= 1'b1;    //cp=1 gives turn to computer
 end
 else if(win==1'b0 && filled== 1'b0)
 begin
  new <= REST;
  cp <= 1'b1;      //cp=1 gives turn to computer
 end

end
```

- PLAYER:

```
PLAYER:
begin
  pp <= 1'b1;
  cp <= 1'b0;
  if(wrong_move==1'b0)
   new <= COMPUTER; // computer's turn
  else
   new <= REST;
 end
```

- OVER:

```
OVER:
begin      // game over
 pp <= 1'b0;
 cp <= 1'b0;
 if(reset==1'b1)
  new <= REST;  // return to REST
 else
  new <=OVER;
end
default:new <=REST;
```

4. Module finish:

INPUT: 2 bit positions that are

- pos1
- pos2
- pos3
- pos4
- pos5
- pos6
- pos7
- pos8
- pos9

OUTPUT:

- filled

This module is used to detect if all 9 positions of the tic tac toe are filled and no one won the game. In this module we have used f1,f2,……..,f9 To check whether the corresponding spaces as mentioned in their right hand side are filled or not.

Then AND of all f's will return 0 – if any space of the tic tac toe is left unfilled.
Otherwise, it will return 1 – if every space is filled.

```
module finish(pos1, pos2, pos3, pos4, pos5, pos6, pos7, pos8, pos9, filled);    //if no more positions are left
   input [1:0] pos1, pos2, pos3, pos4, pos5, pos6, pos7, pos8, pos9;
   output wire filled;                                                          //all positions filled

wire f1,f2,f3,f4,f5,f6,f7,f8,f9;

//f's check out the spaces mentioned in their RHS are filled or not
assign f1 = (pos1[0] | pos1[1]);
assign f2 = (pos2[0] | pos2[1]);
assign f3 = (pos3[0] | pos3[1]);
assign f4 = (pos4[0] | pos4[1]);
assign f5 = (pos5[0] | pos5[1]);
assign f6 = (pos6[0] | pos6[1]);
assign f7 = (pos7[0] | pos7[1]);
assign f8 = (pos8[0] | pos8[1]);
assign f9 = (pos9[0] | pos9[1]);

assign filled =(((((((f1 & f2) & f3) & f4) & f5) & f6) & f7) & f8) & f9);
endmodule
```

5. Module re_block:

INPUT: 2 bit positions that are

- pos1
- pos2
- pos3
- pos4
- pos5
- pos6
- pos7
- pos8
- pos9
- c_enable
- p_enable

OUTPUT:

- wrong_move

This module is used to check whether the player or computer played a wrong move or not.

Rb(i) (for 1<=i<=9): This is when the player makes a wrong move as we have put AND of p_enable with each variable

Rb(i) (for 11<=i<=19): This is when the computer makes a wrong move as we have put AND of c_enable with each variable

Rb21 is when player makes wrong ,ove in any case and similar case is with rb22 for computer.

```
module re_block (pos1, pos2, pos3, pos4, pos5, pos6, pos7, pos8, pos9, c_enable, p_enable, wrong_move); //ch
    input [1:0] pos1, pos2, pos3, pos4, pos5, pos6, pos7, pos8, pos9;
   input [8:0] c_enable, p_enable;

   output wire wrong_move;

wire rb1,rb2,rb3,rb4,rb5,rb6,rb7,rb8,rb9,rb11,rb12,rb13,rb14,rb15,rb16,rb17,rb18,rb19,rb21,rb22;

assign rb1 = (pos1[0] | pos1[1]) & p_enable[0];  //player makes the wrong move
assign rb2 = (pos2[0] | pos2[1]) & p_enable[1];
assign rb3 = (pos3[0] | pos3[1]) & p_enable[2];
assign rb4 = (pos4[0] | pos4[1]) & p_enable[3];
assign rb5 = (pos5[0] | pos5[1]) & p_enable[4];
assign rb6 = (pos6[0] | pos6[1]) & p_enable[5];
assign rb7 = (pos7[0] | pos7[1]) & p_enable[6];
assign rb8 = (pos8[0] | pos8[1]) & p_enable[7];
assign rb9 = (pos9[0] | pos9[1]) & p_enable[8];

assign rb11 = (pos1[0] | pos1[1]) & c_enable[0];    //computer makes the wrong move
assign rb11 = (pos1[0] | pos1[1]) & c_enable[0];
assign rb12 = (pos2[0] | pos2[1]) & c_enable[1];
assign rb13 = (pos3[0] | pos3[1]) & c_enable[2];
assign rb14 = (pos4[0] | pos4[1]) & c_enable[3];
assign rb15 = (pos5[0] | pos5[1]) & c_enable[4];
assign rb16 = (pos6[0] | pos6[1]) & c_enable[5];
assign rb17 = (pos7[0] | pos7[1]) & c_enable[6];
assign rb18 = (pos8[0] | pos8[1]) & c_enable[7];
assign rb19 = (pos9[0] | pos9[1]) & c_enable[8];

assign rb21 =(((((((rb11|rb12)|rb13)|rb14)|rb15)|rb16)|rb17)|rb18)|rb19);   //computer makes the wrong move
assign rb22 =(((((((rb1|rb2)|rb3)|rb4)|rb5)|rb6)|rb7)|rb8)|rb9);   //player makes the wrong move


assign wrong_move = rb21|rb22 ;  //overall wrong move by any of player

endmodule
```

6. Module pos_decode:

INPUT: 4 bit

- In
- En

OUTPUT:

- out_enable

This module is perfect implementation of decoder that we study in digital logic design. It is a 4 to 16 bit decoder (2^4=16 so decodes a 4 bit number to a 16 bit decoded number). It decodes the position of player and computer at all points in game.

```
module pos_decode(in,en,out_enable);   //decodes the playing position, we have implemented the 4 to 16 DECODER
     input[3:0] in ;
     input en;
     output wire [15:0] out_enable;
     reg[15:0] pd;           //decoder variable

 /* always @(*) begin
 if ((en)==(1'b1))
 assign out_enable=pd;
 else
 assign out_enable=16'd0;
 end*/                     //this didn't work out

 assign out_enable=(en==1'b1)?pd:16'd0;  //enable signal

 always @(*)
 begin

 case(in)                          //switch between cases of input
 4'd0: pd <= 16'b0000000000000001;    //d stands for decimal
 4'd1: pd <= 16'b0000000000000010;
 4'd2: pd <= 16'b0000000000000100;
 4'd3: pd <= 16'b0000000000001000;
 4'd4: pd <= 16'b0000000000010000;
 4'd5: pd <= 16'b0000000000100000;
 4'd6: pd <= 16'b0000000001000000;
 4'd7: pd <= 16'b0000000010000000;
 4'd8: pd <= 16'b0000000100000000;
 4'd9: pd <= 16'b0000001000000000;
 4'd10: pd <= 16'b0000010000000000;
 4'd11: pd <= 16'b0000100000000000;
 4'd12: pd <= 16'b0001000000000000;
 4'd13: pd <= 16'b0010000000000000;
 4'd14: pd <= 16'b0100000000000000;
 4'd15: pd <= 16'b1000000000000000;

 default: pd<=16'b0000000000000001;

 endcase

end

endmodule
```

7. Module who_wins:

INPUT: 2 bit positions that are

- pos1
- pos2
- pos3
- pos4
- pos5
- pos6

- pos7
- pos8
- pos9

OUTPUT:

- won
- 2 bit winner

This module checks the three rows, three columns and two diagonals are filled by the same player. It does so by repeatedly calling “who_wins3”. It keeps a check if any of them has won the match or not.
When the same symbol (0 or X) fills up a vertical side or a horizontal side or a diagonal then this module will return the 01 (if player won) or 10 (if computer won) .means it will return the winner.

If none of the player or computer won the game and all the spaces of the tic tac toe filled then it will return 00 and there is no fluctuation come in the EP wave.

```
module who_wins(input[1:0] pos1,pos2,pos3,pos4,pos5,pos6,pos7,pos8,pos9,output wire won,output wire[1:0]winner); //finding out the winner
wire [1:0] winner1,winner2,winner3,winner4,winner5,winner6,winner7,winner8;
wire win_1,win_2,win_3,win_4,win_5,win_6,win_7,win_8;

//when the same symbol fills up a side or a diagonal
who_wins3 a1(pos1,pos2,pos3,win_1,winner1);
who_wins3 a2(pos4,pos5,pos6,win_2,winner2);
who_wins3 a3(pos7,pos8,pos9,win_3,winner3);
who_wins3 a4(pos1,pos4,pos7,win_4,winner4);
who_wins3 a5(pos2,pos5,pos8,win_5,winner5);
who_wins3 a6(pos3,pos6,pos9,win_6,winner6);
who_wins3 a7(pos1,pos5,pos9,win_7,winner7);
who_wins3 a8(pos3,pos5,pos6,win_8,winner8);

assign won = (((((((win_1|win_2) | win_3) | win_4) | win_5) | win_6) | win_7) | win_8);
assign winner = (((((((winner1 | winner2) | winner3) | winner4) | winner5) | winner6) | winner7) | winner8);

endmodule
```

8. Module who_wins3:

INPUT: 2 bit positions that are

- pos0
- pos1
- pos2

OUTPUT:

- won
- 2 bit winner

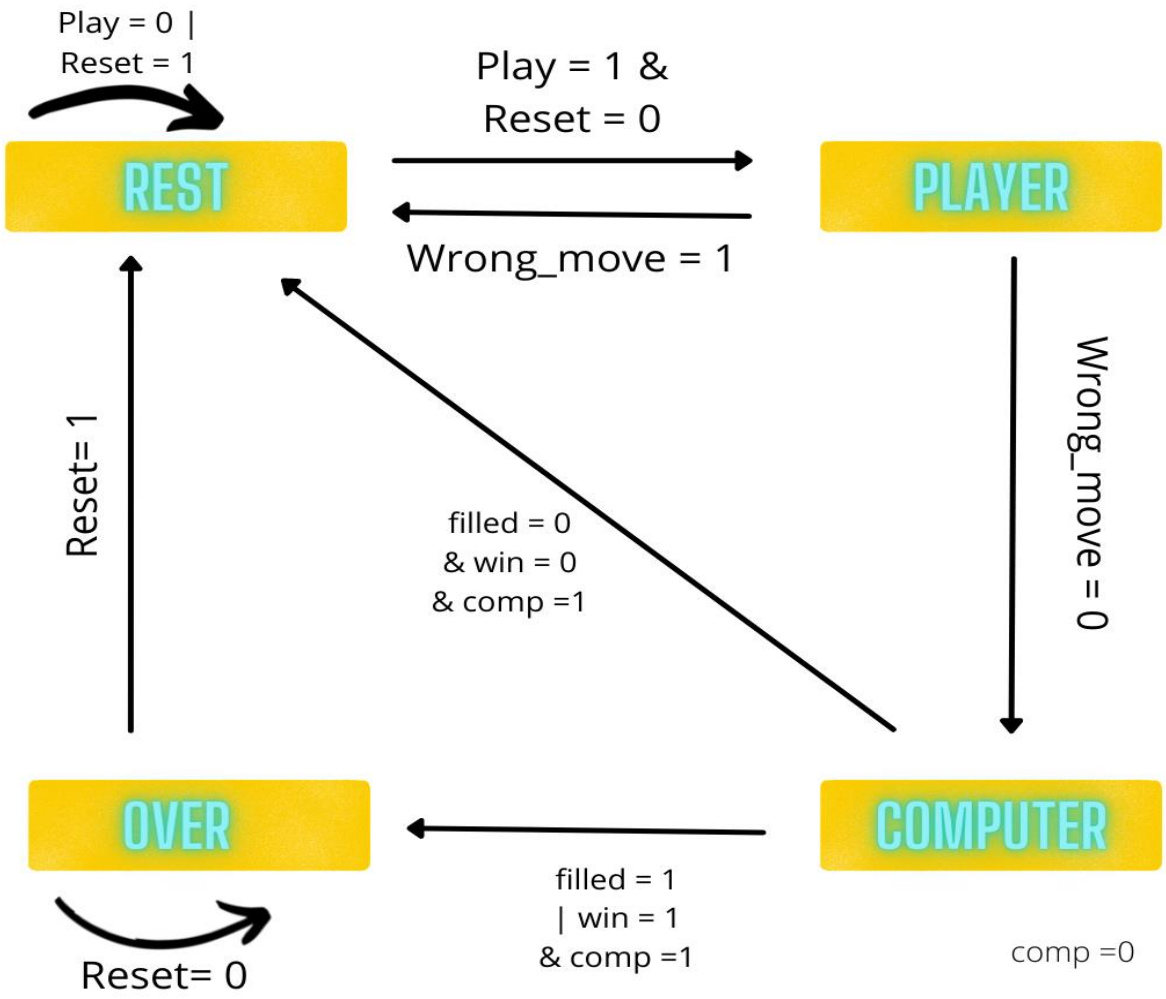
In this module, we check if 3 positions (of a side or diagonal) is filled first by the same player. Output Winner is the winner of game.

```
module who_wins3(input [1:0] pos0,pos1,pos2, output wire won, output wire [1:0]winner);  //used in

wire [1:0] wd0,wd1,wd2;
wire wd3;


assign wd0[0] = !(pos0[0]^pos1[0]);
assign wd0[1] = !(pos0[1]^pos1[1]);
assign wd1[0] = !(pos1[0]^pos2[0]);
assign wd1[1] = !(pos1[1]^pos2[1]);
assign wd2[1] = wd0[1] & wd1[1];
assign wd2[0] = wd0[0] & wd1[0];
assign wd3 = pos0[1] | pos0[0];

assign won = wd2[0]&wd2[1]&wd3 ;  //when the 3 positions in a side/diagonal are by the same player

assign winner[0] = pos0[0]&won;
assign winner[1] = pos0[1]&won;

endmodule
```

DESIGN AND FEATURES:

The main design module of the project is FSM module where we implemented the FSM controller in which all the states are implemented.

The map of the FSM controller is given below where we have explained that how states are changing:



STATE DIAGRAM

The states are as follows:

1. REST state → Rest state comes when player is waiting for the computer or computer is waiting for the player to play.
2. PLAYER state → Player state comes when there is player's turns to play.
3. COMPUTER state → Computer state come when there is computer's turn to play.

4. OVER state → Over state come when the game is finished (one of the 2 player won the match or all spaces of the tic tac toe are filled and no spaces are left to play)

- Play = 0 → Stay in the rest state
- Play = 1 → switches controller to the player state and player's turn comes.

- Reset = 0 → Game starts
- Reset = 1 → It resets the game

- Comp = 0 → stay in the computer state.
- Comp = 1 → switch to the Rest state and computer's turn comes.

- Wrong_move = 0 → Player's state will switch to the Computer's state (if game is in player state). Otherwise computer's state will switch to the player's state (if game is in Computer state)
- Wrong_move = 1 → It means the player or computer played a wrong move which is not valid in the game this will switch to the rest state once again.

- Filled = 0 → if the tic tac toe have enough space to play the next chance.
- Filled = 1 → if tic tac toe does not have enough space to play (all the 9 spaces are filled and no one won the game)

- Win = 0 → No one won the game till now.
- Win = 1 → any one player won the game and game is over and the game resets.

These are all design inputs and outputs used in various modules, the same is explained in state diagram wherever they are used.

VALIDATION APPROACH:

To verify our code, we have added the testbench that can be manipulated by the user. The testbench that we provided is valid and is giving correct results. The winner is declared on the basis of moves the players are playing immediately after the last winning move is played. EP wave displays the results correctly. This indicates that our code is implemented correctly.

We have also added clock: the pulsating wave so periodic changes are visible.

```
initial
 begin
clock=0;
for(integer i=0;i<500;i=i+1)
 #2 clock= ~clock;
end
```

```
initial
 begin

 play = 0;
 reset = 1;
 computer = 0;          //reset=1 so game hasn't started
 player = 0;
 comp= 0;
 #10;
 reset = 0;             //game starts
 #10;
 play = 1;
 comp= 0;
 computer= 3;           // computer at 4
 player= 0;             // player at 1
 #5;
 comp= 1;
 play = 0;
 #10;
 reset = 0;
 play = 1;
 comp= 0;
 computer= 7;            //computer at 8
 player= 4;              // player at 1
 #5;
 comp= 1;
 play = 0;
 #10;
 reset = 0;
 play = 1;
 comp= 0;
 computer= 5;       //computer at 6
 player= 8;         //player at 9 and wins so game ends and winner is declared
 #5;
 comp= 1;
 play = 0;
 #5
 comp= 0;
 play = 0;
 end

 initial
 begin
     $dumpfile("design.vcd");
     $dumpvars;
 end

endmodule
```

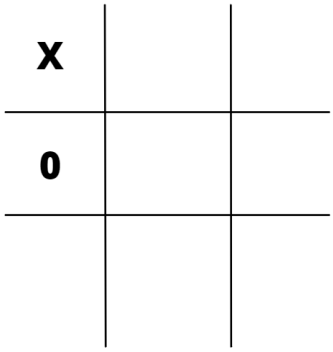
This is the testbench made and the user can edit the ‘player’ and ‘computer’ to edit the positions. The results are displayed on EPWave or gtkwave.

RESULTS:

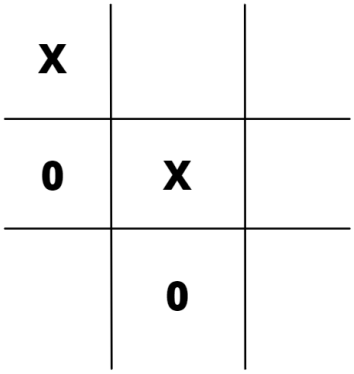
We have successfully implemented, designed the tictactoe in Verilog. The results displayed match with the expected ones.
When reset=0, our game starts. Also, play=1 gives player the chance to play at first. Now the player takes a turn and then the computer (2nd player).

HERE IS THE SEQUENCE OF THE TURNS THAT IS FOLLOWED IN THE TESTBENCH: [X: PLAYER, 0:COMPUTER (AS WRITTEN IN THE CONVENTION IN THE OVERVIEW)]

- Player:1 And Computer:4

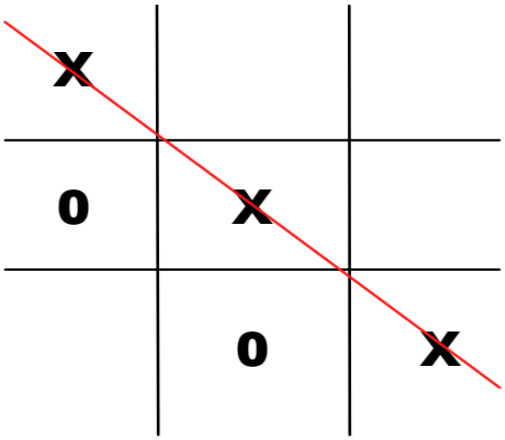


- Player:5 And Computer:8

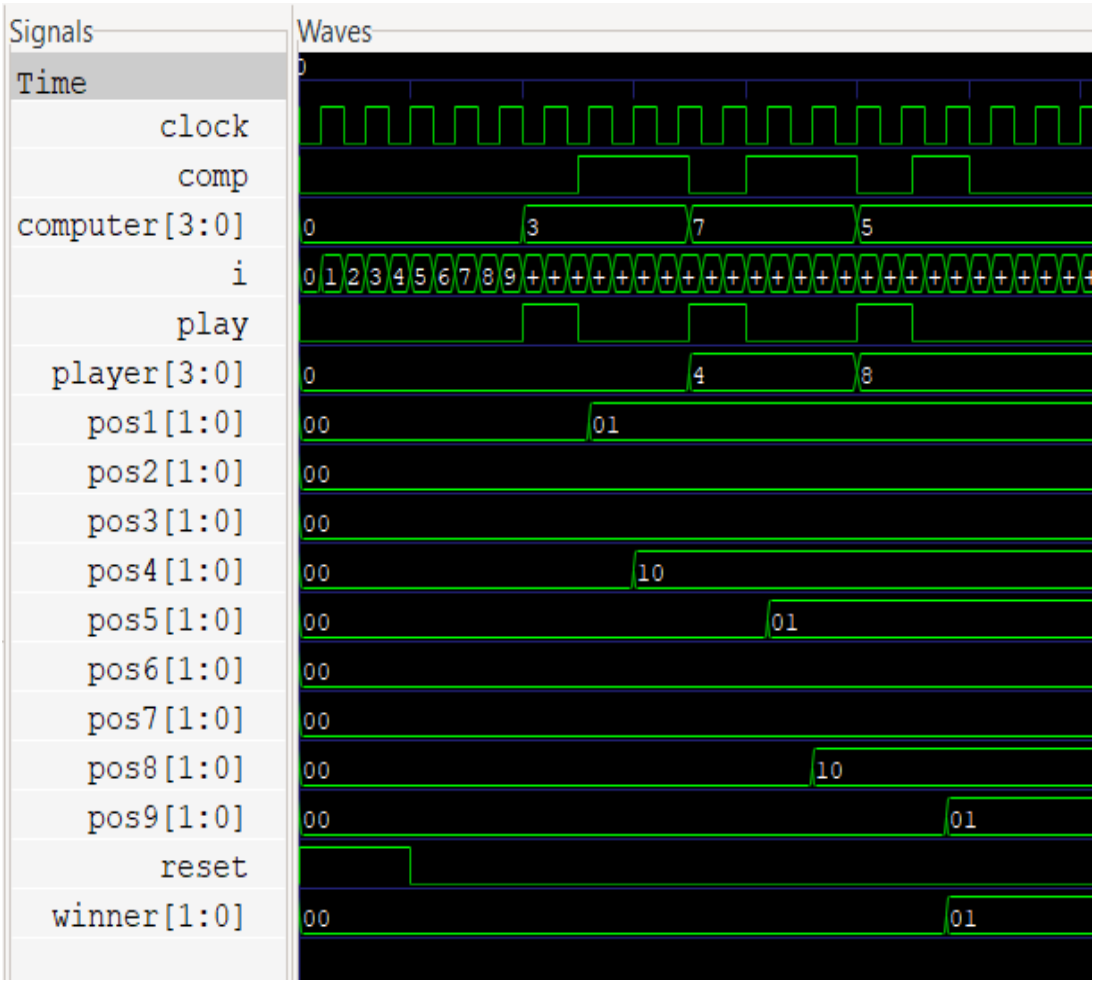


Now as we can see, as soon as the player would play at 9, the player will win and computer won't be able to play the next turn. That's exactly what our code does. The program terminates just after the player plays at 9.

- Player:9



The given below is EPWave for testbench that we provided above:



GITHUB LINK TO PROJECT:

https://github.com/Vijaydwivedi10/CS203_project

ACKNOWLEDGEMENTS:

We would like to thank to our professor Dr. Neeraj Goel who gave us the opportunity to work and collaborate upon the project “TIC TAC TOE using Verilog”. While working upon this project, we did lots of research about this topic and learnt many new things while writing the code as we searched about our problems. Finally. we were able to implement the same.