CS203 - PROJECT
TIC TAC TOE GAME IMPLEMENTATION USING VERILOG

PROJECT BY:
Tanish Goyal (2020CSB1133)
Vijay Dwivedi (2020CSB1140)

INTRODUCTION

We have implemented Tic Tac Toe in verilog using various modules.

We have used computer as the 2nd player in the game.

1	2	3
4	5	6
7	8	9

(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]

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

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]

DESIGN & FEATURES

STATES

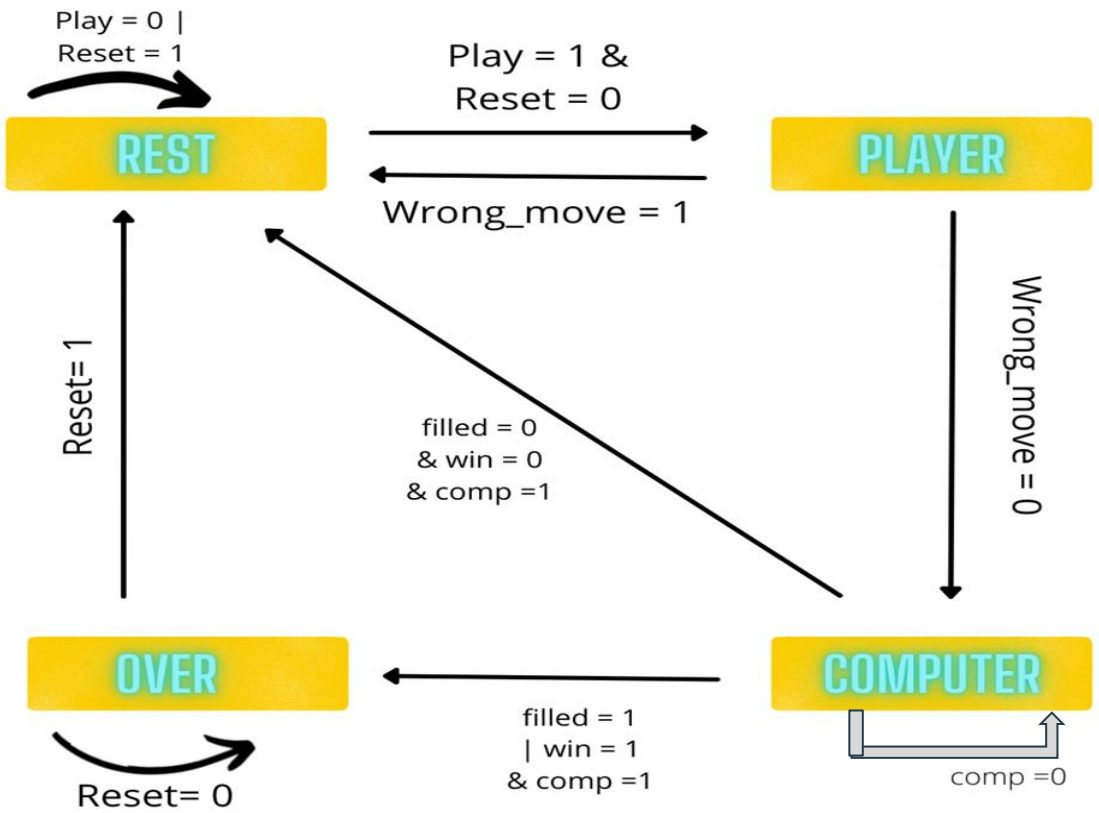
There are 4 states in our project which are as follows :

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

VARIABLES

- **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
- **Pp**=1 → player to play (player's turn)
- **Cp**=1 → computer to play (computer's turn)
- **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).
- 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

State Diagram



MODULES

1. Module Tic_Tac_Toe

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;
```

1. Module Tic_Tac_Toe

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

```
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

This module is used to store the positions of player and computer (2nd player).

```
module position(clock, reset, wrong_move, c_enable, p_enable, pos1, pos2, pos3, pos4, pos5, pos6, pos7, pos8, pos9);
    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
```

Implemented this for all the 9 positions

3. Module FSM

We have implemented state machine in this module.

This is the main module which is used to control the overall gameplay.

```
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
```

Parameter represents constant and are often used to define variable width and delay value

3. Module FSM

We have implemented state machine in this module.

This is the main module which is used to control the overall gameplay.

```
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:
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
```

3. Module FSM

We have implemented state machine in this module.

This is the main module which is used to control the overall gameplay.

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

```
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

This module is used to detect if all 9 positions are filled and no one won the game.

```
module finish(pos1, pos2, pos3, pos4, pos5, pos6, pos7, pos8, pos9, filled);
   input [1:0] pos1, pos2, pos3, pos4, pos5, pos6, pos7, pos8, pos9;
   output wire 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
```

f(i) = 1 → i th position is filled

When all 9 positions are filled

Filled = 1

5. Module Re_block

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

```
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
```

for(i>=1 && i<=9)

If rb(i) = 1 → rb22 = 1
Implies Player played a wrong move

for(i>=11 && i<=19)

If rb(i) = 1 → rb21 = 1
Implies computer played a wrong move

6. Module Pos_decode

It decodes the position of player and computer at all points in a game.it is 4 to 16 bit decoder.

```
/* 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
```

7. Module Who_wins3

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

```
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
```

8. Module Who_wins

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.

```
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
```

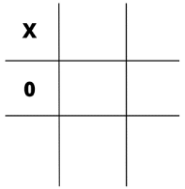
RESULTS

Test_Benc

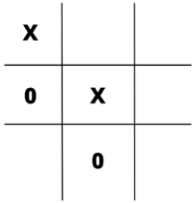
```
initial
 begin

 play = 0;
 reset = 1;
 computer = 0;          //reset
 player = 0;
 comp= 0;
 #10;
 reset = 0;             //game
 #10;
 play = 1;
 comp= 0;
 computer= 3;           // comp
 player= 0;             // play
 #5;
 comp= 1;
 play = 0;
 #10;
 reset = 0;
 play = 1;
 comp= 0;
 computer= 7;           //comp
 player= 4;             // pla
 #5;
 comp= 1;
 play = 0;
 #10;
 reset = 0;
 play = 1;
 comp= 0;
 computer= 5;       //computer
 player= 8;         //player_at
 #5;
 comp= 1;
 play = 0;
 #5
 comp= 0;
 play = 0;
 end

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

endmodule
```

Game starts

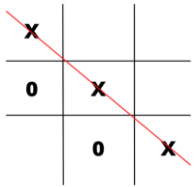
Player → 1
Computer → 4



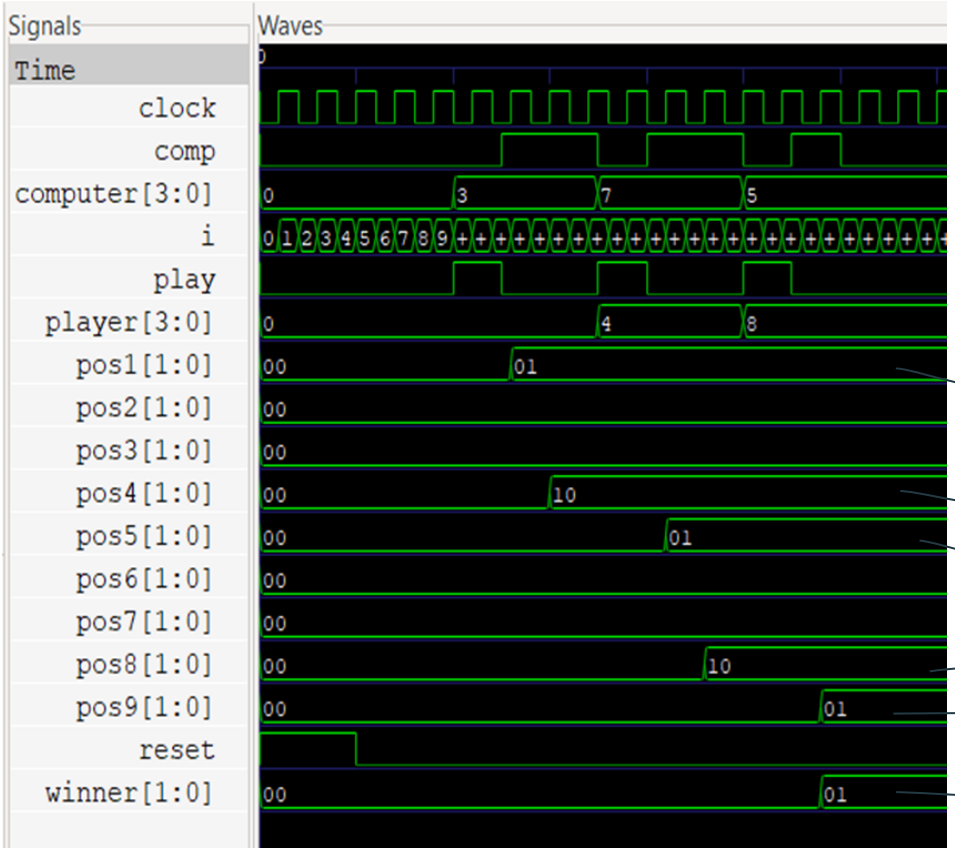
Player → 5
Computer → 8

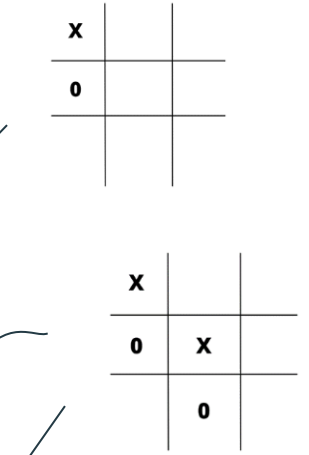


Player → 9
Computer → 6



EP_Wave


Signals
Waves
Time
clock
comp
computer[3:0]
i
play
player[3:0]
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]
reset
winner[1:0]


X
0
X
0
X
0
X
0
X

WE THANK NEERAJ SIR FOR GIVING US THIS OPPORTUNITY TO WORK AND COLLABORATE ON PROJECT.