

HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING



Logic Design with HDL ASSIGNMENT

CC01 - GROUP 4

Submitted on: 21st June, 2024
Supported by: Mr. Nguyen The Binh

Full name	Student's ID	Contribution
Thai Hoang Gia Thoai	2353146	Implementation
Tran Thai Tai	2353065	Write Code
Huynh Van Thanh	2353083	Gather Information
Ngo Vinh Quang	2352968	Write Report

Contents

Logic Design with HDL ASSIGNMENT	1
1 Introduction	3
1.1 Introduction to Tic Tac Toe with Logic Design Using HDL	3
1.2 Background of Tic Tac Toe in Logic Design	3
1.3 Key Components	3
2 Design Summary	3
2.1 State Definitions:	3
3 Methodology	4
3.1 Parameter Definition	4
3.2 Registers and Arrays for State and Data Storage	4
3.3 State Machine Implementation	4
3.4 Synchronous and Asynchronous Reset	4
3.5 Button Debouncing and Input Handling	5
3.6 Win Condition Checking	5
3.7 LED Output Control	6
4 Block Diagram	7
5 Verilog Code	7
6 Testbench	10
7 Waveform Analysis	12
7.1 Clock Signal (clk)	12
7.2 Reset Signal (reset)	12
7.3 Movement and Confirmation Signals (left, right, confirm)	13
7.4 Game State Outputs	13
7.5 Test Sequence Analysis	13
7.5.1 Initial Reset and Move to Position (0, 0)	13
7.5.2 Confirming Moves Across the Board	13
7.5.3 Additional Scenarios with Different Players	13
7.5.4 End of Simulation	14
7.6 Expected Waveform Highlights	14
7.7 Received Waveform	14
7.7.1 Initial Conditions and Clock Generation	14
7.8 Detailed Waveform Analysis	15
7.9 Summary	16
8 Timing Analysis	16
8.1 Clock Generation	16
8.2 Reset Initialization	16
8.3 Player Moves and Confirmations	16
8.3.1 Game Session 1	17
8.3.2 Game Session 2	18
8.3.3 Game Session 3	18
8.4 Summary	19
9 Conclusion	20
10 References	20



1 Introduction

1.1 Introduction to Tic Tac Toe with Logic Design Using HDL

Tic-Tac-Toe, a quintessential game of strategy and foresight, serves as an ideal pedagogical tool in the realms of game theory, artificial intelligence, and digital logic design. This seemingly simple game, where two players alternately mark spaces in a 3x3 grid with the objective of aligning three consecutive marks horizontally, vertically, or diagonally, offers profound insights into complex theoretical and practical concepts.

The interplay of strategic thinking and technical execution in Tic-Tac-Toe makes it an exemplary framework for fostering a deep understanding of HDLs and their applications in contemporary logic design.

1.2 Background of Tic Tac Toe in Logic Design

1. **Educational Tool:** Tic Tac Toe serves as an excellent project for learning and applying logic design principles. Designing a Tic Tac Toe game in HDL requires understanding of combinational and sequential logic, state machines, and basic digital design principles.
2. **Simple yet Comprehensive:** The simplicity of Tic Tac Toe makes it a perfect candidate for a beginner's project in HDL. Despite its simplicity, the game encapsulates a wide range of logic design challenges, including input handling, state management, and output display.
3. **Foundation for Advanced Concepts:** Starting with a simple game like Tic Tac Toe provides a foundation for understanding more complex digital design projects. The concepts learned can be scaled and adapted to larger, more complex systems.

1.3 Key Components

1. **State Machine:** The core of the Tic Tac Toe game logic can be implemented using a finite state machine (FSM). The FSM will handle the various states of the game, including:
 - Initial state (reset)
 - Player turns
 - Win detection
 - Draw detection
 - Resetting the game
2. **Input Handling:** The design needs to manage inputs from two players. Each player's move corresponds to placing a mark on the grid, which can be mapped to specific inputs in the HDL design.
3. **Grid Representation:** The 3x3 grid can be represented using a 2D array or a similar data structure. Each cell in the array can hold values to indicate whether it is empty, occupied by 'X', or occupied by 'O'.
4. **Win/Draw Detection:** The logic for detecting a win or a draw involves checking the rows, columns, and diagonals for three consecutive 'X's or 'O's.
5. **Output Handling:** The design should include outputs to indicate the current state of the game, such as displaying the grid, showing whose turn it is, and declaring the winner or a draw.

2 Design Summary

2.1 State Definitions:

1. **Module Declaration:** Defines the module, inputs, and outputs.
2. **Parameters and Constants:** Specifies grid size, player states, cell values, and state machine states.
3. **Internal Registers and Variables:** Declares the game board, state register, and loop variables.
4. **Reset Logic:** Initializes game variables, sets the board to empty, and resets LEDs.
5. **State Machine:**

- Handles cursor movement within the grid.
- Places marks on the board based on player input.

6. Win Condition Checks:

- Verifies if a player has won or if the game is a draw.
- Updates game status and LEDs accordingly.

7. LED Output Logic: Controls LED patterns to indicate game state and cursor position.

3 Methodology

3.1 Parameter Definition

Parameters are used to define constants for the tic-tac-toe grid size and player states, making the code more readable and maintainable.

```
1 parameter SIZE = 3;           // Size of the tic tac toe grid (3x3)
2 parameter X = 1'b0;           // X player state
3 parameter O = 1'b1;           // O player state
4 parameter EMPTY = 2'b00;      // Empty cell value
5 parameter X_MARK = 2'b01;     // X player's mark
6 parameter O_MARK = 2'b10;     // O player's mark
```

3.2 Registers and Arrays for State and Data Storage

Registers and arrays are used to store the game state, the board state, and the cursor position. This allows the state machine to maintain and update the game's status.

```
1 reg [1:0] board [2:0][2:0]; // Game board (3x3 grid)
2 reg state;
3 reg [1:0] p_row;           // Cursor row position (0 to SIZE-1)
4 reg [1:0] p_col;           // Cursor column position (0 to SIZE-1)
```

3.3 State Machine Implementation

A finite state machine (FSM) is used to control the game flow. The states include **IDLE** for waiting for input and **PLACE** for placing a mark on the board. The FSM handles input from buttons to move the cursor and confirm selections.

```
1 parameter IDLE = 1'b0; // Idle state (waiting for input)
2 parameter PLACE = 1'b1; // Place mark state
```

3.4 Synchronous and Asynchronous Reset

An asynchronous reset is used to initialize the game state and board. The reset signal is active low, meaning it initializes the system when it is low (\sim reset).

```
1 always @(posedge clk or negedge reset) begin
2     if (~reset) begin
3         lreset <= 1;
4         leds <= 9'b000000000;
5         state <= IDLE;
6         player_turn <= X;
7         // Initialize board to empty
8         for (i = 0; i <= SIZE - 1; i = i + 1) begin
9             for (j = 0; j <= SIZE - 1; j = j + 1) begin
10                board[i][j] <= EMPTY;
11            end
12        end
13        // Set Pointer to (0,0)
14        p_row <= 2'b00;
15        p_col <= 2'b00;
16        p1 <= 1'b0;
17        p2 <= 1'b0;
18        game_over <= 1'b0;
19        winner <= 2'b00;
```

```

20     end else begin
21         lreset <= 0;
22     end
23 end

```

3.5 Button Debouncing and Input Handling

The code handles button inputs (`left`, `right`, `confirm`) to move the cursor and place marks. It ensures the cursor wraps around the grid correctly and places marks only in empty cells.

```

1  always @(posedge clk) begin
2      if (reset && state == IDLE && game_over == 0) begin
3          if (left && (p_col > 0))
4              p_col <= p_col - 1;
5          else if (right && (p_col < SIZE - 1))
6              p_col <= p_col + 1;
7          else if (right && (p_col == SIZE - 1)) begin
8              p_col <= 0;
9              if (p_row < SIZE - 1)
10                 p_row <= p_row + 1;
11             else
12                 p_row <= 0;
13         end else if (left && (p_col == 0)) begin
14             p_col <= SIZE - 1;
15             if (p_row != 0)
16                 p_row <= p_row - 1;
17             else
18                 p_row <= SIZE - 1;
19         end else if (confirm) begin
20             if (board[p_row][p_col] == EMPTY) begin
21                 if (player_turn == X) begin
22                     board[p_row][p_col] <= X_MARK;
23                 end else begin
24                     board[p_row][p_col] <= O_MARK;
25                 end
26                 state <= PLACE;
27             end
28         end
29     end
30 end

```

3.6 Win Condition Checking

The code checks for win conditions after a mark is placed, including rows, columns, and diagonals. It updates the game state accordingly.

```

1  always @(posedge clk) begin
2      if (state == PLACE) begin
3          // Check win conditions after placing mark
4          // Check rows
5          if (board[p_row][0] == board[p_row][1] && board[p_row][1] == board[p_row][2]) begin
6              game_over <= 1;
7              if (board[p_row][p_col] == X_MARK && !p1 && !p2) begin
8                  winner <= 2'b01; // X wins
9                  p1 <= 1'b1;
10             end else if (board[p_row][p_col] == O_MARK && !p1 && !p2) begin
11                 winner <= 2'b10; // O wins
12                 p2 <= 1'b1;
13             end
14             state <= IDLE;
15         end
16         // Check columns
17         else if (board[0][p_col] == board[1][p_col] && board[1][p_col] == board[2][p_col]) begin
18             game_over <= 1;
19             if (board[p_row][p_col] == X_MARK && !p1 && !p2) begin
20                 winner <= 2'b01; // X wins
21                 p1 <= 1'b1;
22             end else if (board[p_row][p_col] == O_MARK && !p1 && !p2) begin
23                 winner <= 2'b10; // O wins
24                 p2 <= 1'b1;
25             end
26             state <= IDLE;

```

```

27     end
28     // Check diagonals
29     else if ((p_row == p_col && board[0][0] == board[1][1] && board[1][1] == board[2][2]) ||
30         (p_row + p_col == 2 && board[0][2] == board[1][1] && board[1][1] == board[2][0]))
31     begin
32         game_over <= 1;
33         if (board[p_row][p_col] == X_MARK && !p1 && !p2) begin
34             winner <= 2'b01; // X wins
35             p1 <= 1'b1;
36         end else if (board[p_row][p_col] == O_MARK && !p1 && !p2) begin
37             winner <= 2'b10; // O wins
38             p2 <= 1'b1;
39         end
40         state <= IDLE;
41     end
42     // Check for draw
43     else if (board[0][0] != EMPTY && board[0][1] != EMPTY && board[0][2] != EMPTY &&
44         board[1][0] != EMPTY && board[1][1] != EMPTY && board[1][2] != EMPTY &&
45         board[2][0] != EMPTY && board[2][1] != EMPTY && board[2][2] != EMPTY) begin
46         game_over <= 1;
47         winner <= 2'b11; // draw
48         p1 <= 1'b1;
49         p2 <= 1'b1;
50     end
51     if (game_over == 0) begin
52         if (player_turn == X)
53             player_turn <= 0;
54         else
55             player_turn <= X;
56         state <= IDLE;
57     end
58 end

```

3.7 LED Output Control

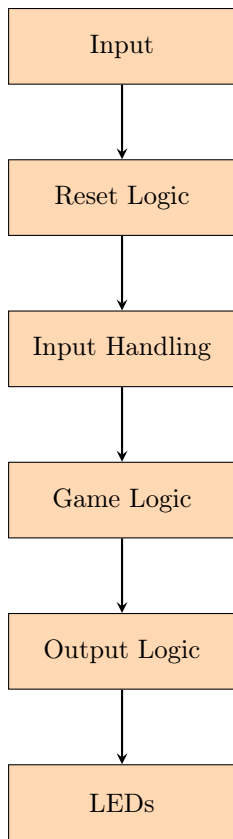
The code updates the LEDs to reflect the current game state, highlighting the current cursor position and indicating the winner or a draw when the game ends.

```

1 always @(posedge clk) begin
2     if (game_over == 1) begin
3         // Game over: indicate winner or draw
4         case (winner)
5             2'b01: leds <= 9'b101_010_101; // LEDs for X wins
6             2'b10: leds <= 9'b111_101_111; // LEDs for O wins
7             2'b11: leds <= 9'b111_111_111; // LEDs for draw
8         endcase
9     end else begin
10        // Game ongoing: update LEDs to show board state
11        leds <= 9'b000000000;
12        // Highlight current cursor position
13        leds[p_row * 3 + p_col] <= 1'b1;
14    end
15 end

```

4 Block Diagram



5 Verilog Code

```

1 `timescale 1ns / 1ps
2
3 module tic_tac_toe (
4     input wire clk,           // System clock input
5     input wire reset,        // Reset input (active low)
6     input wire left,         // Button for left (move cursor left)
7     input wire right,        // Button for right (move cursor right)
8     input wire confirm,      // Button to confirm selection
9     output reg [8:0] leds,    // to show if enough led implemented
10    output reg lreset,        // LED for reset
11    output reg game_over,     // State machine state register
12    output reg p1,            // Player 1 == Player X
13    output reg p2,            // Player 2 == Player O
14    output reg player_turn,    // Current player's turn (X or O)
15    output reg [1:0] winner,   // Winner (11 for draw, 01 for X, 10 for O)
16    // Additional variables for cursor position
17    output reg [1:0] p_row,     // Cursor row position (0 to SIZE-1)
18    output reg [1:0] p_col     // Cursor column position (0 to SIZE-1)
19 );
20
21 // Define states and constants
22 parameter SIZE = 3;          // Size of the tic tac toe grid (3x3)
23
24 parameter X = 1'b0;          // X player state
25 parameter O = 1'b1;          // O player state
26
27
28 parameter EMPTY = 2'b00;     // Empty cell value
29 parameter X_MARK = 2'b01;    // X player's mark
30 parameter O_MARK = 2'b10;    // O player's mark
31 reg [1:0] board [2:0][2:0];  // Game board (3x3 grid) (0, 1, 2 for naming)
32
33 // State machine states
34 parameter IDLE = 1'b0;       // Idle state (waiting for input)

```

```

35 parameter PLACE = 1'b1;    // Place mark state
36 reg state;
37
38 integer i;
39 integer j;
40 // Reset all game variables
41 always @(posedge clk or negedge reset) begin
42     if (~reset) begin
43         lreset <= 1;
44         leds <= 9'b000000000;
45         state <= IDLE;
46         player_turn <= X; // X starts first
47         // Initialize board to empty
48         for ( i = 0; i <= SIZE - 1; i = i + 1) begin
49             for ( j = 0; j <= SIZE - 1; j = j + 1) begin
50                 board[i][j] <= EMPTY;
51             end
52         end
53         // Set Pointer to (0,0)
54         p_row <= 2'b00;
55         p_col <= 2'b00;
56         // Initialize LEDs to all off
57         p1 <= 1'b0;
58         p2 <= 1'b0;
59         game_over <= 1'b0;
60         winner <= 2'b00;
61     end
62     else begin
63         lreset <= 0;
64         i = 0;
65         j = 0;
66     end
67 end
68
69 // State machine for controlling the game flow
70 always @(posedge clk) begin
71     if (reset && state == IDLE && game_over == 0) begin
72         if (left && (p_col > 0)) // move to the left col if at col 2 or 3
73             p_col <= p_col - 1;
74         else if (right && (p_col < SIZE - 1)) // move to right col if at col 1 or 2
75             p_col <= p_col + 1;
76         else if (right && (p_col == SIZE - 1)) begin // move to right at col 3
77             p_col <= 0;
78             if (p_row < SIZE - 1) // move to next row if available
79                 p_row <= p_row + 1;
80             else
81                 p_row <= 0; // Wrap around to the first row if not
82         end
83         else if (left && (p_col == 0)) begin // move to left at col 1
84             p_col <= SIZE - 1;
85             if (p_row != 0) // move to previous row if available
86                 p_row <= p_row - 1;
87             else
88                 p_row <= SIZE - 1; // Wrap around to the last row if not
89         end
90     end
91     else if (confirm) begin // Press select button
92         if (board[p_row][p_col] == EMPTY) begin
93             if (player_turn == X) begin
94                 board[p_row][p_col] <= X_MARK;
95                 $display("Player 1 place mark X at ( %d, %d )", p_row, p_col);
96             end
97             else begin
98                 board[p_row][p_col] <= O_MARK;
99                 $display("Player 2 place mark O at ( %d, %d )", p_row, p_col);
100             end
101             state <= PLACE;
102         end
103     end
104 end
105 end
106 // Output LEDs based on game board and game status
107 always @(posedge clk) begin
108     if (state == PLACE) begin
109         // Check win conditions after placing mark

```



```

110 // Check rows
111 if (board[p_row][0] == board[p_row][1] &&
112     board[p_row][1] == board[p_row][2]) begin
113     $display("ROW WIN");
114     game_over <= 1;
115     // Game over: indicate winner or draw
116 if (board[p_row][p_col] == X_MARK && !p1 && !p2) begin
117     winner <= 2'b01; // X wins
118     p1 <= 1'b1;
119     end
120 else if (board[p_row][p_col] == O_MARK && !p1 && !p2) begin
121     winner <= 2'b10; // O wins
122     p2 <= 1'b1;
123     end
124     $display("Game state: %d, over? : %d", state, game_over);
125     state <= IDLE;
126     end
127 // Check columns
128 else if (board[0][p_col] == board[1][p_col] &&
129     board[1][p_col] == board[2][p_col]) begin
130     $display("COL WIN");
131     game_over <= 1;
132     // Game over: indicate winner or draw
133 if (board[p_row][p_col] == X_MARK && !p1 && !p2) begin
134     winner <= 2'b01; // X wins
135     p1 <= 1'b1;
136     end
137 else if (board[p_row][p_col] == O_MARK && !p1 && !p2) begin
138     winner <= 2'b10; // O wins
139     p2 <= 1'b1;
140     end
141     $display("Game state: %d, over? : %d", state, game_over);
142     state <= IDLE;
143     end
144
145 // Check diagonals
146 else if ((p_row == p_col &&
147     board[0][0] == board[1][1] &&
148     board[1][1] == board[2][2]) ||
149     (p_row + p_col == 2 &&
150     board[0][2] == board[1][1] &&
151     board[1][1] == board[2][0])) begin
152     $display("DIAGONAL WIN");
153     game_over <= 1;
154     // Game over: indicate winner or draw
155 if (board[p_row][p_col] == X_MARK && !p1 && !p2) begin
156     winner <= 2'b01; // X wins
157     p1 <= 1'b1;
158     end
159 else if (board[p_row][p_col] == O_MARK && !p1 && !p2) begin
160     winner <= 2'b10; // O wins
161     p2 <= 1'b1;
162     end
163     $display("Game state: %d, over? : %d", state, game_over);
164     state <= IDLE;
165     end
166
167 // Check for draw
168 else if (board[0][0] != EMPTY && board[0][1] != EMPTY && board[0][2] != EMPTY &&
169     board[1][0] != EMPTY && board[1][1] != EMPTY && board[1][2] != EMPTY &&
170     board[2][0] != EMPTY && board[2][1] != EMPTY && board[2][2] != EMPTY) begin
171     game_over <= 1;
172     // Game over: indicate winner or draw
173 winner <= 2'b11; // draw
174     $display("Game state: %d, over? : %d", state, game_over);
175     state <= IDLE;
176     $display("DRAW");
177     p1 <= 1'b1;
178     p2 <= 1'b1;
179     end
180 if (game_over == 0) begin ///
181 if (player_turn == X)
182     player_turn <= 0;
183 else
184     player_turn <= X;

```

```

185         state <= IDLE;
186     end
187     $display("Game state: %d, over? : %d", state, game_over);
188 end
189 end
190 // Output LEDs based on game board and game status
191 always @(posedge clk) begin
192     if (game_over == 1) begin
193         // Game over: indicate winner or draw
194         case (winner)
195             2'b01: leds <= 9'b101_010_101; // LEDs for X wins
196             2'b10: leds <= 9'b111_101_111; // LEDs for O wins
197             2'b11: leds <= 9'b111_111_111; // LEDs for draw
198         endcase
199     end
200     else begin
201         // Game ongoing: update LEDs to show board state
202         leds <= 9'b000000000;
203         // Highlight current cursor position
204         leds[p_row * 3 + p_col] <= 1'b1;
205     end
206 end
207
208 endmodule

```

6 Testbench

```

1  `timescale 1ns / 1ps
2
3  module tb_tic_tac_toe();
4
5      // Signals
6      reg clk;
7      reg reset;
8      reg left, right, confirm;
9      wire [8:0] leds;
10     wire lreset;
11     wire game_over;
12     wire p1, p2;
13     wire player_turn;
14     wire [1:0] winner;
15     wire [1:0] p_row;
16     wire [1:0] p_col;
17
18     // Instantiate the tic_tac_toe module
19     tic_tac_toe dut (
20         .clk(clk),
21         .reset(reset),
22         .left(left),
23         .right(right),
24         .confirm(confirm),
25         .leds(leds),
26         .lreset(lreset),
27         .game_over(game_over),
28         .p1(p1),
29         .p2(p2),
30         .player_turn(player_turn),
31         .winner(winner),
32         .p_row(p_row),
33         .p_col(p_col)
34     );
35
36     // Clock generation
37     always begin
38         clk = 0;
39         #5;
40         clk = 1;
41         #5;
42     end
43
44     // Reset initialization
45     initial begin
46         reset = 1;

```

```

47     #10;
48     reset = 0;
49     left = 0;
50     right = 0;
51     confirm = 0;
52     #10;
53     left = 1;
54     #10;
55     left = 0;
56     #3;
57     reset = 1;
58     #27;
59
60     #3;
61     confirm = 1; #3; confirm = 0; #17;
62     // Move to position (0, 0) and confirm
63     right = 1; #3; right = 0; #7;
64     confirm = 1; #3; confirm = 0; #17;
65
66     // Move to position (0, 1) and confirm
67     right = 1; #3; right = 0; #7;
68     confirm = 1; #3; confirm = 0; #17;
69
70     // Move to position (0, 2) and confirm
71     right = 1; #3; right = 0; #7;
72     confirm = 1; #3; confirm = 0; #17;
73
74     // Move to position (1, 0) and confirm
75     right = 1; #3; right = 0; #7;
76     confirm = 1; #3; confirm = 0; #17;
77
78     // Move to position (1, 1) and confirm
79     right = 1; #3; right = 0; #7;
80     confirm = 1; #3; confirm = 0; #17;
81
82     // Move to position (1, 2) and confirm
83     right = 1; #3; right = 0; #7;
84     confirm = 1; #3; confirm = 0; #17;
85
86     // Move to position (2, 0) and confirm
87     right = 1; #3; right = 0; #7;
88     confirm = 1; #3; confirm = 0; #17;
89
90     // Move to position (2, 1) and confirm
91     right = 1; #3; right = 0; #7;
92     confirm = 1; #3; confirm = 0; #17;
93
94     // Move to position (2, 2) and confirm
95     // End simulation after sufficient time
96     #30;
97
98     //////////////////////////////////////
99     reset = 0;
100    #3;
101    reset = 1;
102    #27;
103    confirm = 1; #3; confirm = 0; #17;
104    // Move to position (0, 0) and confirm of player 1
105    right = 1; #3; right = 0; #7;
106    confirm = 1; #3; confirm = 0; #17;
107
108    // Move to position (0, 1) and confirm of player 2
109    right = 1; #3; right = 0; #7;
110    right = 1; #3; right = 0; #7;
111    confirm = 1; #3; confirm = 0; #17;
112
113    // Move to position (1,0) and confirm of player 1
114    right = 1; #3; right = 0; #7;
115    confirm = 1; #3; confirm = 0; #17;
116
117    // Move to position (1, 1) and confirm of player 2
118    right = 1; #3; right = 0; #7;
119    right = 1; #3; right = 0; #7;
120    right = 1; #3; right = 0; #7;
121    right = 1; #3; right = 0; #7;

```

```

122     confirm = 1; #3; confirm = 0; #17;
123
124     // Move to position (2, 2) and confirm of player 1
125     left = 1; #3; left = 0; #7;
126     confirm = 1; #3; confirm = 0; #17;
127     // Move to position (2, 1) and confirm of player 2
128
129     #30;
130     //////////////////////////////////////////
131
132     reset = 0;
133     #3;
134     reset = 1;
135     #27;
136     confirm = 1; #3; confirm = 0; #17;
137     // Move to position (0, 0) and confirm
138     right = 1; #3; right = 0; #7;
139     confirm = 1; #3; confirm = 0; #17;
140
141     // Move to position (0, 1) and confirm
142     right = 1; #3; right = 0; #7;
143     confirm = 1; #3; confirm = 0; #17;
144
145     // Move to position (0, 2) and confirm
146     right = 1; #3; right = 0; #7;
147     confirm = 1; #3; confirm = 0; #17;
148
149     // Move to position (1, 0) and confirm
150     right = 1; #3; right = 0; #7;
151     confirm = 1; #3; confirm = 0; #17;
152
153     // Move to position (1, 1) and confirm
154     right = 1; #3; right = 0; #7;
155     right = 1; #3; right = 0; #7;
156     right = 1; #3; right = 0; #7;
157     right = 1; #3; right = 0; #7;
158     confirm = 1; #3; confirm = 0; #17;
159
160     // Move to position (2, 2) and confirm of p2
161     left = 1; #3; left = 0; #7;
162     confirm = 1; #3; confirm = 0; #17;
163
164     // Move to position (2, 1) and confirm of p1
165     left = 1; #3; left = 0; #7;
166     confirm = 1; #3; confirm = 0; #17;
167
168     // Move to position (2, 0) and confirm of p2
169     left = 1; #3; left = 0; #7;
170     confirm = 1; #3; confirm = 0; #17;
171
172     // Move to position (1, 2) and confirm of p1
173     // End simulation after sufficient time
174     #30;
175
176     #20;
177     $finish; // Finish simulation
178 end
179
180 endmodule

```

7 Waveform Analysis

7.1 Clock Signal (clk)

The clk signal is a continuous square wave with a period of 10 ns (5 ns high, 5 ns low).

7.2 Reset Signal (reset)

Initially, **reset** is asserted high for 10 ns to initialize the system. After 10 ns, **reset** is deasserted low to start the game. The reset signal is manipulated at various points to reinitialize the game state and test different scenarios.

7.3 Movement and Confirmation Signals (left, right, confirm)

The **left**, **right**, and **confirm** signals control the movement and confirmation of the player's selection on the board. These signals are pulsed at specific times to simulate the player's actions.

7.4 Game State Outputs

- **leds**: Represents the current state of the game board.
- **lreset**: Local reset for the game.
- **game_over**: Indicates if the game is over.
- **p1, p2**: Player 1 and Player 2 indicators.
- **player_turn**: Indicates whose turn it is to play.
- **winner**: Indicates the winner of the game (00 for no winner, 01 for player 1, 10 for player 2).
- **p_row, p_col**: Current row and column position selected.

7.5 Test Sequence Analysis

7.5.1 Initial Reset and Move to Position (0, 0)

1. **Time 0-10 ns**: **reset** is high to initialize.
2. **Time 10-20 ns**: **reset** goes low, starting the game. **left**, **right**, and **confirm** are low.
3. **Time 20-30 ns**: **left** is pulsed high to move left (though initially not needed since it starts at (0, 0)).
4. **Time 30-40 ns**: **left** goes low, no change.
5. **Time 40-70 ns**: **reset** is toggled to ensure proper initialization.

7.5.2 Confirming Moves Across the Board

The **right** and **confirm** signals are pulsed to move the cursor and confirm positions:

- **Position (0, 0)**: **confirm** is pulsed at 70 ns.
- **Position (0, 1)**: **right** at 90 ns, **confirm** at 120 ns.
- **Position (0, 2)**: **right** at 140 ns, **confirm** at 170 ns.
- **Position (1, 0)**: **right** at 190 ns, **confirm** at 220 ns.
- **Position (1, 1)**: **right** at 240 ns, **confirm** at 270 ns.
- **Position (1, 2)**: **right** at 290 ns, **confirm** at 320 ns.
- **Position (2, 0)**: **right** at 340 ns, **confirm** at 370 ns.
- **Position (2, 1)**: **right** at 390 ns, **confirm** at 420 ns.
- **Position (2, 2)**: **right** at 440 ns, **confirm** at 470 ns.

7.5.3 Additional Scenarios with Different Players

1. **Second Game Sequence**:
 - The **reset** signal is manipulated again to start a new game.
 - Similar moves are made with alternating player turns.
 - Example: Player 1 moves to (0, 0), confirmed at 500 ns. Player 2 moves to (0, 1), confirmed at 540 ns after moving the cursor.

7.5.4 End of Simulation

The simulation ends after sufficient moves and reset sequences to ensure all scenarios are tested. Each move and confirmation changes the game state and updates the `leds`, `player_turn`, and potentially the `game_over` and `winner` signals.

7.6 Expected Waveform Highlights

- `clk`: Continuous square wave.
- `reset`: Pulsed high initially, then manipulated for reinitializations.
- `left/right/confirm`: Pulsed to simulate player actions.
- `leds`: Update reflecting the game board state.
- `game_over`: High when the game ends.
- `player_turn`: Alternates between players.
- `winner`: Indicates the winning player at game end.
- `p_row/p_col`: Reflects the current cursor position.

7.7 Received Waveform

7.7.1 Initial Conditions and Clock Generation

- The clock signal (`clk`) is generated with a period of 10ns.
- At $T = 0\text{ns}$, the `reset` signal is initially set to 1, ensuring the system starts in a known state.

T = 10ns

- The `reset` signal is deasserted (set to 0), allowing the system to begin normal operation.
- The `left`, `right`, and `confirm` signals are all 0 initially.

***T = 20ns**

- The `left` signal is asserted (set to 1).

T = 30ns

- The `left` signal is deasserted (set to 0).

T = 33ns

- The `reset` signal is asserted (set to 1) again to reset the game state.

T = 60ns

- The `reset` signal is deasserted (set to 0).

T = 63ns

- The `confirm` signal is asserted (set to 1) for a brief period to confirm the current position (0,0) for player 1.
- The `confirm` signal is deasserted (set to 0).

T = 80ns to T = 97ns

- The sequence of moving right and confirming the position is repeated to place moves on the tic-tac-toe board.

T = 100ns

- The **right** signal is asserted and deasserted to move the cursor to the next position.
- The **confirm** signal is asserted and deasserted to confirm the position (0, 1).

T = 117ns to T = 397ns

- This sequence continues with the cursor moving and confirming positions for players 1 and 2.
- Moves are made sequentially as described in the testbench code.

T = 430ns

- The reset signal is deasserted (set to 0), resetting the game state.
- Another series of moves is initiated.

7.8 Detailed Waveform Analysis

T = 0ns to T = 10ns • **reset = 1**

- The tic-tac-toe module is reset, all signals are at their initial state.

T = 20ns • **left = 1**

- The cursor moves to the left.

T = 30ns • **left = 0**

- The cursor stops moving.

T = 33ns • **reset = 1**

- The game state is reset.

T = 60ns • **reset = 0**

- The game state is now active, ready for player input.

T = 63ns • **confirm = 1**

- Player 1 confirms the move at position (0,0).
- The position is updated, and the game state reflects this move.

T = 80ns to T = 97ns • **right = 1**, then **right = 0**

- **confirm = 1**, then **confirm = 0**
- The cursor moves to the next position, and the move is confirmed by player 1.

T = 100ns • **right = 1**, then **right = 0**

- **confirm = 1**, then **confirm = 0**
- The cursor moves to position (0,1), confirmed by player 2.

T = 117ns to T = 397ns • The cursor moves and confirms the positions as described.

- Player turns are alternated.
- The game state (**leds**, **player_turn**, **p_row**, **p_col**, etc.) is updated accordingly.

T = 430ns • **reset = 0**

- The game state is reset, and a new game can begin.

7.9 Summary

1. Initial Setup:

- System reset at $T = 0\text{ns}$.
- Clock starts toggling every 5ns.

2. First Reset:

- Reset deasserted at $T = 10\text{ns}$.
- Initial move sequence begins at $T = 20\text{ns}$.

3. Game Moves:

- Series of moves made by players with positions confirmed.
- Alternating between player 1 and player 2.
- Cursor moves indicated by `left` and `right` signals.
- Moves confirmed by `confirm` signal.

4. Game Reset and New Moves:

- Game state reset at $T = 33\text{ns}$.
- New series of moves start.

5. Final Reset:

- Final game reset at $T = 430\text{ns}$.

8 Timing Analysis

8.1 Clock Generation

The clock (`clk`) is generated with a period of 10 ns:

```
always begin
    clk = 0;
    #5;
    clk = 1;
    #5;
end
```

- `clk` toggles every 5 ns, creating a clock period of 10 ns.

8.2 Reset Initialization

The reset (`reset`) signal is initially set high for 10 ns to initialize the system:

```
reset = 1;
#10;
reset = 0;
```

8.3 Player Moves and Confirmations

The testbench simulates multiple game sessions, each with specific player moves and confirmations. Here is the breakdown of the timing and events:



8.3.1 Game Session 1

Initial Reset and Move to Position (0, 0)

```
reset = 1; #10;
reset = 0;
left = 0;
right = 0;
confirm = 0; #10;
left = 1; #10;
left = 0; #3;
reset = 1; #27;
#3;
confirm = 1; #3; confirm = 0; #17;
```

- Reset for 10 ns, set control signals to 0, move left for 10 ns, release left for 3 ns. - Another reset for 27 ns, followed by a confirmation pulse of 3 ns.

Move and Confirm Position (0, 1)

```
right = 1; #3; right = 0; #7;
confirm = 1; #3; confirm = 0; #17;
```

- Move right for 3 ns, wait for 7 ns, confirm for 3 ns, and wait for 17 ns.

Move and Confirm Position (0, 2)

```
right = 1; #3; right = 0; #7;
confirm = 1; #3; confirm = 0; #17;
```

Move and Confirm Position (1, 0)

```
right = 1; #3; right = 0; #7;
confirm = 1; #3; confirm = 0; #17;
```

Move and Confirm Position (1, 1)

```
right = 1; #3; right = 0; #7;
confirm = 1; #3; confirm = 0; #17;
```

Move and Confirm Position (1, 2)

```
right = 1; #3; right = 0; #7;
confirm = 1; #3; confirm = 0; #17;
```

Move and Confirm Position (2, 0)

```
right = 1; #3; right = 0; #7;
confirm = 1; #3; confirm = 0; #17;
```

Move and Confirm Position (2, 1)

```
right = 1; #3; right = 0; #7;
confirm = 1; #3; confirm = 0; #17;
```

Move and Confirm Position (2, 2)

```
#30;
```

8.3.2 Game Session 2

Initial Reset and Move to Position (0, 0)

```
reset = 0; #3; reset = 1; #27;  
confirm = 1; #3; confirm = 0; #17;
```

Move and Confirm Position (0, 1) for Player 2

```
right = 1; #3; right = 0; #7;  
confirm = 1; #3; confirm = 0; #17;
```

Move and Confirm Position (1, 0) for Player 1

```
right = 1; #3; right = 0; #7;  
confirm = 1; #3; confirm = 0; #17;
```

Move and Confirm Position (1, 1) for Player 2

```
right = 1; #3; right = 0; #7;  
right = 1; #3; right = 0; #7;  
confirm = 1; #3; confirm = 0; #17;
```

Move and Confirm Position (2, 2) for Player 1

```
left = 1; #3; left = 0; #7;  
confirm = 1; #3; confirm = 0; #17;
```

8.3.3 Game Session 3

Initial Reset and Move to Position (0, 0)

```
reset = 0; #3; reset = 1; #27;  
confirm = 1; #3; confirm = 0; #17;
```

Move and Confirm Position (0, 1)

```
right = 1; #3; right = 0; #7;  
confirm = 1; #3; confirm = 0; #17;
```

Move and Confirm Position (0, 2)

```
right = 1; #3; right = 0; #7;  
confirm = 1; #3; confirm = 0; #17;
```

Move and Confirm Position (1, 0)

```
right = 1; #3; right = 0; #7;  
confirm = 1; #3; confirm = 0; #17;
```

Move and Confirm Position (1, 1)

```
right = 1; #3; right = 0; #7;  
right = 1; #3; right = 0; #7;  
right = 1; #3; right = 0; #7;  
confirm = 1; #3; confirm = 0; #17;
```



Move and Confirm Position (2, 2) for Player 2

```
left = 1; #3; left = 0; #7;  
confirm = 1; #3; confirm = 0; #17;
```

Move and Confirm Position (2, 1) for Player 1

```
left = 1; #3; left = 0; #7;  
confirm = 1; #3; confirm = 0; #17;
```

Move and Confirm Position (2, 0) for Player 2

```
left = 1; #3; left = 0; #7;  
confirm = 1; #3; confirm = 0; #17;
```

Move and Confirm Position (1, 2) for Player 1

```
#30;
```

8.4 Summary

- **Clock period:** 10 ns - **Reset pulse:** Initially 10 ns, later sessions 30 ns - **Control signals (left, right, confirm):** 3 ns active, 7 ns inactive before confirmation - **Confirmation:** 3 ns active, followed by 17 ns wait before next action

Move and Confirm Position (0, 1)

```
right = 1; #3; right = 0; #7;  
confirm = 1; #3; confirm = 0; #17;
```

Move and Confirm Position (0, 2)

```
right = 1; #3; right = 0; #7;  
confirm = 1; #3; confirm = 0; #17;
```

Move and Confirm Position (1, 0)

```
right = 1; #3; right = 0; #7;  
confirm = 1; #3; confirm = 0; #17;
```

Move and Confirm Position (1, 1)

```
right = 1; #3; right = 0; #7;  
confirm = 1; #3; confirm = 0; #17;
```

Move and Confirm Position (1, 2)

```
right = 1; #3; right = 0; #7;  
confirm = 1; #3; confirm = 0; #17;
```

Move and Confirm Position (2, 0)

```
right = 1; #3; right = 0; #7;  
confirm = 1; #3; confirm = 0; #17;
```

Move and Confirm Position (2, 1)

```
right = 1; #3; right = 0; #7;  
confirm = 1; #3; confirm = 0; #17;
```



Move and Confirm Position (2, 2)

#30;

9 Conclusion

Implementing Tic Tac Toe using Hardware Description Language (HDL) provides a comprehensive learning experience in digital logic design, encompassing both combinational and sequential logic. It involves programming in Verilog or VHDL, using modular design for clarity and reusability, and verifying functionality through simulation and testbenches. The project highlights the importance of state machine design and efficient resource utilization, especially in FPGA implementation. It addresses challenges like concurrency and optimization, enhancing problem-solving skills and understanding of system-level design. Overall, this project bridges theoretical knowledge with practical application, laying a strong foundation for more complex digital design endeavors.

10 References

- Dr. Pham Quoc Cuong, Logic Design with HDL lectures (2023-2024)
- <https://www.fpga4student.com/>
- <https://www.fpga4fun.com/>
- <https://www.element14.com/>
- <https://www.hackster.io/>
- <https://www.instructables.com/circuits/howto/fpga/>