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

Logic Design with HDL ASSIGNMENT

CC01 - GROUP 4

Submitted on: 21^{st} 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

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.

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

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

```
parameter IDLE = 1'b0;  // Idle state (waiting for input)
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 (∼reset).

```
always @(posedge clk or negedge reset) begin
    if (~reset) begin
        lreset <= 1;
        leds <= 9'b000000000;
        state <= IDLE;
        player_turn <= X;
        // Initialize board to empty
        for (i = 0; i <= SIZE - 1; i = i + 1) begin
            for (j = 0; j <= SIZE - 1; j = j + 1) begin
                board[i][j] <= EMPTY;
            end
        end
        // Set Pointer to (0,0)
        p_row <= 2'b00;
        p_col <= 2'b00;
        p1 <= 1'b0;
        p2 <= 1'b0;
        game_over <= 1'b0;
        winner <= 2'b00;
```

```
    end else begin
        lreset <= 0;
    end
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.

```
always @(posedge clk) begin
    if (reset && state == IDLE && game_over == 0) begin
        if (left && (p_col > 0))
            p_col <= p_col - 1;
        else if (right && (p_col < SIZE - 1))
            p_col <= p_col + 1;
        else if (right && (p_col == SIZE - 1)) begin
            p_col <= 0;
            if (p_row < SIZE - 1)
                p_row <= p_row + 1;
            else
                p_row <= 0;
        end else if (left && (p_col == 0)) begin
            p_col <= SIZE - 1;
            if (p_row != 0)
                p_row <= p_row - 1;
            else
                p_row <= SIZE - 1;
        end else if (confirm) begin
            if (board[p_row][p_col] == EMPTY) begin
                if (player_turn == X) begin
                    board[p_row][p_col] <= X_MARK;
                end else begin
                    board[p_row][p_col] <= O_MARK;
                end
                state <= PLACE;
            end
        end
    end
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.

```
always @(posedge clk) begin
    if (state == PLACE) begin
        // Check win conditions after placing mark
        // Check rows
        if (board[p_row][0] == board[p_row][1] && board[p_row][1] == board[p_row][2]) begin
            game_over <= 1;
            if (board[p_row][p_col] == X_MARK && !p1 && !p2) begin
                winner <= 2'b01; // X wins
                p1 <= 1'b1;
            end else if (board[p_row][p_col] == O_MARK && !p1 && !p2) begin
                winner <= 2'b10; // O wins
                p2 <= 1'b1;
            end
            state <= IDLE;
        end
        // Check columns
        else if (board[0][p_col] == board[1][p_col] && board[1][p_col] == board[2][p_col]) begin
            game_over <= 1;
            if (board[p_row][p_col] == X_MARK && !p1 && !p2) begin
                winner <= 2'b01; // X wins
                p1 <= 1'b1;
            end else if (board[p_row][p_col] == O_MARK && !p1 && !p2) begin
                winner <= 2'b10; // O wins
                p2 <= 1'b1;
            end
            state <= IDLE;
```

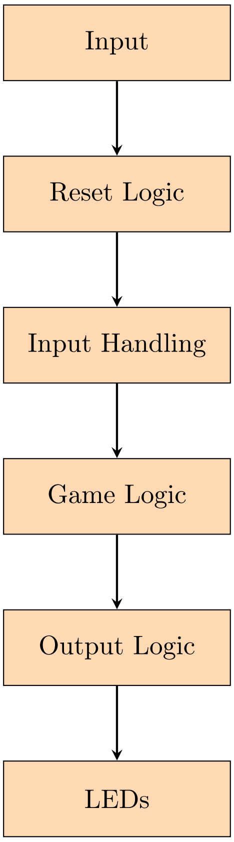
```
        end
        // Check diagonals
        else if ((p_row == p_col && board[0][0] == board[1][1] && board[1][1] == board[2][2]) ||
                 (p_row + p_col == 2 && board[0][2] == board[1][1] && board[1][1] == board[2][0]))
    begin
            game_over <= 1;
            if (board[p_row][p_col] == X_MARK && !p1 && !p2) begin
                winner <= 2'b01; // X wins
                p1 <= 1'b1;
            end else if (board[p_row][p_col] == O_MARK && !p1 && !p2) begin
                winner <= 2'b10; // O wins
                p2 <= 1'b1;
            end
            state <= IDLE;
        end
        // Check for draw
        else if (board[0][0] != EMPTY && board[0][1] != EMPTY && board[0][2] != EMPTY &&
                 board[1][0] != EMPTY && board[1][1] != EMPTY && board[1][2] != EMPTY &&
                 board[2][0] != EMPTY && board[2][1] != EMPTY && board[2][2] != EMPTY) begin
            game_over <= 1;
            winner <= 2'b11; // draw
            p1 <= 1'b1;
            p2 <= 1'b1;
        end
        if (game_over == 0) begin
            if (player_turn == X)
                player_turn <= O;
            else
                player_turn <= X;
            state <= IDLE;
        end
    end
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.

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

4 Block Diagram



5 Verilog Code

```
`timescale 1ns / 1ps

module tic_tac_toe (
    input wire clk,          // System clock input
    input wire reset,        // Reset input (active low)
    input wire left,       // Button for left (move cursor left)
    input wire right,      // Button for right (move cursor right)
    input wire confirm,    // Button to confirm selection
    output reg [8:0] leds, // to show if enough led implemented
    output reg lreset,       // LED for reset
    output reg game_over,        // State machine state register
    output reg p1,     // Player 1 == Player X
    output reg p2,   // Player 2 == Player O
    output reg player_turn,                  // Current player's turn (X or O)
    output reg [1:0] winner,        // Winner (11 for draw, 01 for X, 10 for O)
    // Additional variables for cursor position
output reg [1:0] p_row,           // Cursor row position (0 to SIZE-1)
output reg [1:0] p_col            // Cursor column position (0 to SIZE-1)
);

// Define states and constants
parameter SIZE = 3;          // Size of the tic tac toe grid (3x3)

parameter X = 1'b0;          // X player state
parameter O = 1'b1;          // O player state


parameter EMPTY = 2'b00;        // Empty cell value
parameter X_MARK = 2'b01;       // X player's mark
parameter O_MARK = 2'b10;       // O player's mark
reg [1:0] board [2:0][2:0];      // Game board (3x3 grid) (0, 1, 2 for naming)

// State machine states
parameter IDLE = 1'b0;          // Idle state (waiting for input)
```

```
parameter PLACE = 1'b1;   // Place mark state
reg state;

integer i;
integer j;
// Reset all game variables
always @(posedge clk or negedge reset) begin
    if (~reset) begin
    lreset <= 1;
    leds <= 9'b000000000;
    state <= IDLE;
        player_turn <= X;  // X starts first
        // Initialize board to empty
         for ( i = 0; i <= SIZE - 1; i = i + 1) begin
            for ( j = 0; j <= SIZE - 1; j = j + 1) begin
                board[i][j] <= EMPTY;
            end
        end
        // Set Pointer to (0,0)
        p_row <= 2'b00;
        p_col <= 2'b00;
        // Initialize LEDs to all off
        p1 <= 1'b0;
        p2 <= 1'b0;
        game_over <= 1'b0;
        winner <= 2'b00;
    end
    else begin
    lreset <= 0;
    i = 0;
    j = 0;
    end
end

// State machine for controlling the game flow
always @(posedge clk) begin
if (reset && state == IDLE && game_over == 0) begin
    if (left && (p_col > 0)) // move to the left col if at col 2 or 3
        p_col <= p_col - 1;
    else if (right && (p_col < SIZE - 1)) // move to right col if at col 1 or 2
        p_col <= p_col + 1;
    else if (right && (p_col == SIZE - 1)) begin // move to right at col 3
        p_col <= 0;
        if (p_row < SIZE - 1) // move to next row if available
           p_row <= p_row + 1;
           else
           p_row <= 0; // Wrap around to the first row if not
        end
    else if (left && (p_col == 0)) begin // move to left at col 1
        p_col <= SIZE - 1;
           if (p_row != 0) // move to previous row if available
               p_row <= p_row - 1;
           else
               p_row <= SIZE - 1; // Wrap around to the last row if not
        end

    else if (confirm) begin // Press select button
         if (board[p_row][p_col] == EMPTY) begin
            if (player_turn == X) begin
               board[p_row][p_col] <= X_MARK;
            $display("Player 1 place mark X at ( %d, %d )", p_row, p_col);
            end
            else begin
               board[p_row][p_col] <= O_MARK;
            $display("Player 2 place mark O at ( %d, %d )", p_row, p_col);
            end
            state <= PLACE;
            end
         end
     end
 end
// Output LEDs based on game board and game status
always @(posedge clk) begin
if (state == PLACE) begin
// Check win conditions after placing mark
```

```
            // Check rows
            if (board[p_row][0] == board[p_row][1] &&
                board[p_row][1] == board[p_row][2]) begin
                $display("ROW WIN");
                game_over <= 1;
                // Game over: indicate winner or draw
         if (board[p_row][p_col] == X_MARK && !p1 && !p2) begin
                winner <= 2'b01; // X wins
                p1 <= 1'b1;
                end
            else if (board[p_row][p_col] == O_MARK && !p1 && !p2) begin
                winner <= 2'b10; // O wins
                p2 <= 1'b1;
                end
                    $display("Game state: %d, over? : %d", state, game_over);
                    state <= IDLE;
                end
            // Check columns
            else if (board[0][p_col] == board[1][p_col] &&
                     board[1][p_col] == board[2][p_col]) begin
                $display("COL WIN");
                game_over <= 1;
                // Game over: indicate winner or draw
         if (board[p_row][p_col] == X_MARK && !p1 && !p2) begin
                winner <= 2'b01; // X wins
                p1 <= 1'b1;
                end
            else if (board[p_row][p_col] == O_MARK && !p1 && !p2) begin
                winner <= 2'b10; // O wins
                p2 <= 1'b1;
                end
                    $display("Game state: %d, over? : %d", state, game_over);
                    state <= IDLE;
                end

            // Check diagonals
            else if ((p_row == p_col &&
                      board[0][0] == board[1][1] &&
                      board[1][1] == board[2][2]) ||
                     (p_row + p_col == 2 &&
                      board[0][2] == board[1][1] &&
                      board[1][1] == board[2][0])) begin
                $display("DIAGONAL WIN");
                game_over <= 1;
                // Game over: indicate winner or draw
         if (board[p_row][p_col] == X_MARK && !p1 && !p2) begin
                winner <= 2'b01; // X wins
                p1 <= 1'b1;
                end
            else if (board[p_row][p_col] == O_MARK && !p1 && !p2) begin
                winner <= 2'b10; // O wins
                p2 <= 1'b1;
                end
                    $display("Game state: %d, over? : %d", state, game_over);
                    state <= IDLE;
                end

            // Check for draw
            else if (board[0][0] != EMPTY && board[0][1] != EMPTY && board[0][2] != EMPTY &&
                 board[1][0] != EMPTY && board[1][1] != EMPTY && board[1][2] != EMPTY &&
                 board[2][0] != EMPTY && board[2][1] != EMPTY && board[2][2] != EMPTY) begin
               game_over <= 1;
               // Game over: indicate winner or draw
         winner <= 2'b11; // draw
                    $display("Game state: %d, over? : %d", state, game_over);
                    state <= IDLE;
               $display("DRAW");
               p1 <= 1'b1;
               p2 <= 1'b1;
            end
    if (game_over == 0) begin  ////
    if (player_turn == X)
                    player_turn <= O;
                else
                    player_turn <= X;
```

```
                    state <= IDLE;
                end
                    $display("Game state: %d, over? : %d", state, game_over);
         end
    end
// Output LEDs based on game board and game status
always @(posedge clk) begin
    if (game_over == 1) begin
        // Game over: indicate winner or draw
        case (winner)
            2'b01: leds <= 9'b101_010_101; // LEDs for X wins
            2'b10: leds <= 9'b111_101_111; // LEDs for O wins
            2'b11: leds <= 9'b111_111_111; // LEDs for draw
        endcase
    end
    else begin
        // Game ongoing: update LEDs to show board state
        leds <= 9'b000000000;
        // Highlight current cursor position
        leds[p_row * 3 + p_col] <= 1'b1;
    end
end

endmodule
```

6 Testbench

```
`timescale 1ns / 1ps

module tb_tic_tac_toe();

    // Signals
    reg clk;
    reg reset;
    reg left, right, confirm;
    wire [8:0] leds;
    wire lreset;
    wire game_over;
    wire p1, p2;
    wire player_turn;
    wire [1:0] winner;
    wire [1:0] p_row;
    wire [1:0] p_col;

    // Instantiate the tic_tac_toe module
    tic_tac_toe dut (
        .clk(clk),
        .reset(reset),
        .left(left),
        .right(right),
        .confirm(confirm),
        .leds(leds),
        .lreset(lreset),
        .game_over(game_over),
        .p1(p1),
        .p2(p2),
        .player_turn(player_turn),
        .winner(winner),
        .p_row(p_row),
        .p_col(p_col)
    );

    // Clock generation
    always begin
        clk = 0;
        #5;
        clk = 1;
        #5;
    end

    // Reset initialization
    initial begin
        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;
        // Move to position (0, 0) and confirm
        right = 1; #3; right = 0; #7;
        confirm = 1; #3; confirm = 0; #17;

        // Move to position (0, 1) and confirm
        right = 1; #3; right = 0; #7;
        confirm = 1; #3; confirm = 0; #17;

        // Move to position (0, 2) and confirm
        right = 1; #3; right = 0; #7;
        confirm = 1; #3; confirm = 0; #17;

        // Move to position (1, 0) and confirm
        right = 1; #3; right = 0; #7;
        confirm = 1; #3; confirm = 0; #17;

        // Move to position (1, 1) and confirm
        right = 1; #3; right = 0; #7;
        confirm = 1; #3; confirm = 0; #17;

        // Move to position (1, 2) and confirm
        right = 1; #3; right = 0; #7;
        confirm = 1; #3; confirm = 0; #17;

        // Move to position (2, 0) and confirm
        right = 1; #3; right = 0; #7;
        confirm = 1; #3; confirm = 0; #17;

        // Move to position (2, 1) and confirm
        right = 1; #3; right = 0; #7;
        confirm = 1; #3; confirm = 0; #17;

        // Move to position (2, 2) and confirm
        // End simulation after sufficient time
        #30;

        ////////////////////////////////////////////////
        reset = 0;
        #3;
        reset = 1;
        #27;
        confirm = 1; #3; confirm = 0; #17;
        // Move to position (0, 0) and confirm of player 1
        right = 1; #3; right = 0; #7;
        confirm = 1; #3; confirm = 0; #17;

        // Move to position (0, 1) and confirm of player 2
        right = 1; #3; right = 0; #7;
        right = 1; #3; right = 0; #7;
        confirm = 1; #3; confirm = 0; #17;

        // Move to position (1,0) and confirm of player 1
        right = 1; #3; right = 0; #7;
        confirm = 1; #3; confirm = 0; #17;

        // Move to position (1, 1) and confirm of player 2
        right = 1; #3; right = 0; #7;
        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 to position (2, 2) and confirm of player 1
        left = 1; #3; left = 0; #7;
        confirm = 1; #3; confirm = 0; #17;
        // Move to position (2, 1) and confirm of player 2

        #30;
        ////////////////////////////////////////////////

        reset = 0;
        #3;
        reset = 1;
        #27;
        confirm = 1; #3; confirm = 0; #17;
        // Move to position (0, 0) and confirm
        right = 1; #3; right = 0; #7;
        confirm = 1; #3; confirm = 0; #17;

        // Move to position (0, 1) and confirm
        right = 1; #3; right = 0; #7;
        confirm = 1; #3; confirm = 0; #17;

        // Move to position (0, 2) and confirm
        right = 1; #3; right = 0; #7;
        confirm = 1; #3; confirm = 0; #17;

        // Move to position (1, 0) and confirm
        right = 1; #3; right = 0; #7;
        confirm = 1; #3; confirm = 0; #17;

        // Move to position (1, 1) and confirm
        right = 1; #3; right = 0; #7;
        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 to position (2, 2) and confirm of p2
        left = 1; #3; left = 0; #7;
        confirm = 1; #3; confirm = 0; #17;

        // Move to position (2, 1) and confirm of p1
        left = 1; #3; left = 0; #7;
        confirm = 1; #3; confirm = 0; #17;

        // Move to position (2, 0) and confirm of p2
        left = 1; #3; left = 0; #7;
        confirm = 1; #3; confirm = 0; #17;

        // Move to position (1, 2) and confirm of p1
        // End simulation after sufficient time
        #30;

        #20;
        $finish; // Finish simulation
    end

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 $T = 0$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

$T = 0$**ns to** $T = 10$**ns** • `reset` = 1

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

$T = 20$**ns** • `left` = 1

- The cursor moves to the left.

$T = 30$**ns** • `left` = 0

- The cursor stops moving.

$T = 33$**ns** • `reset` = 1

- The game state is reset.

$T = 60$**ns** • `reset` = 0

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

$T = 63$**ns** • `confirm` = 1

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

$T = 80$**ns to** $T = 97$**ns** • `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.

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

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

$T = 117$**ns to** $T = 397$**ns** • 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.

$T = 430$**ns** • `reset` = 0

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

7.9 Summary

1. **Initial Setup**:
   - System reset at $T = 0$ns.
   - Clock starts toggling every 5ns.
2. **First Reset**:
   - Reset deasserted at $T = 10$ns.
   - Initial move sequence begins at $T = 20$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 $T = 33$ns.
   - New series of moves start.
5. **Final Reset**:
   - Final game reset at $T = 430$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/