Faculty of Engineering
Cairo University

Cairo University

Electronics II

Project 2

Tic Tac Toe Game

Team 21

Team Members:
Khaled Mohammed Badr
Yassmeen Tarek Attia
Nourhan Ahmed
Carole Emad

Submitted to: Eng. Samar Taher

Table of contents:

Our Objective:

The objective of our code is to implement a Tic Tac Toe game using Verilog. The code describes the hardware modules necessary to create the game, including position registers, a winner detector, a position decoder for the computer and player, an illegal move detector, a no space detector, and an FSM (Finite State Machine) controller.
The code aims to simulate the functionality of a Tic Tac Toe game on a digital hardware platform. It allows players to make moves by pressing buttons (play and pc) and enables the computer to play as an opponent. The position registers store the positions played by the players and the computer.

We used both sequential and combinational circuits. The sequential circuit is required to handle the game's state and determine the next move (state). The combinational is used to handle various logical operations such as win, tie, illegal and valid move checking. Overall, the objective for the players is to position their marks so that they make a continuous line of three cells vertically, horizontally or diagonally.

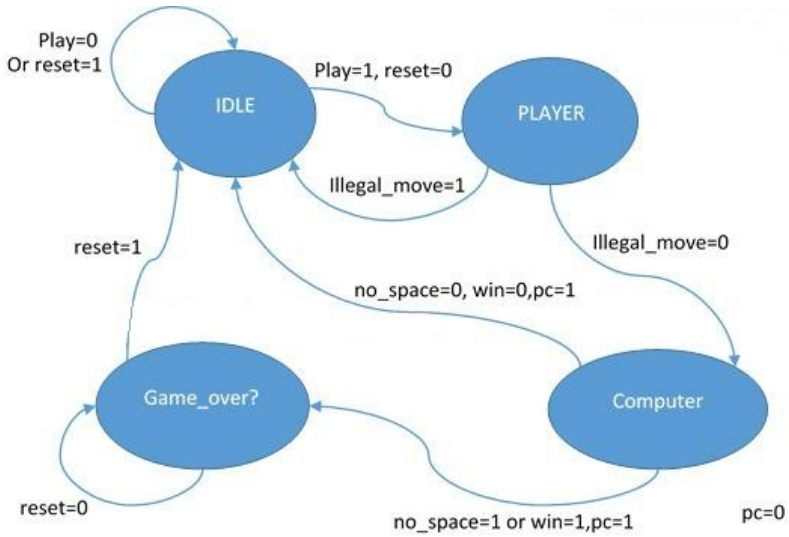
Overview:

Our project consists of multiple modules that work together to control the game logic, store player and computer positions, detect illegal moves, detect a win or no more space, and determine which player's turn it is.

The following 3x3 grid table represents the order of the positions being played:

1	2	3
4	5	6
7	8	9

The FSM controller is as follows:



The modules of the project are as follows:

1. `tic_tac_toe_game`: This is the top-level module that connects all the other modules. It takes input signals like clock, reset, play, pc (computer play), and player_position/computer_position to determine the positions played by the player and computer. It also has output signals for LED display positions and the winner.

2. `position_registers`: This module is responsible for storing the player and computer positions. It has registers for each position (pos1-pos9), and based on the inputs, it updates the positions accordingly.

3. `fsm_controller`: This module implements the FSM controller that determines the game state and controls the flow of the game. It takes inputs such as clock, reset, play, pc, illegal_move, no_space, and win to transition between different states (IDLE, PLAYER, COMPUTER, and GAME_DONE). It enables the player and computer to play based on the current state.

4. `nospace_detector`: This module detects if there are no more spaces available to play. It checks the positions (pos1-pos9) and sets the `no_space` signal accordingly.

5. `illegal_move_detector`: This module detects if a player makes an illegal move by playing on an already occupied position. It checks the positions (pos1-pos9) and the enable signals (PC_en, PL_en) to determine if an illegal move has occurred.

6. `position_decoder`: This module is responsible for decoding the computer_position input and activating the corresponding PC_en (computer enable) signal. The position_decoder module takes a 4-bit input called computer_position, which represents the position where the computer should play. The position_decoder uses combinational logic to decode the computer_position input and activate the appropriate PC_en signal. Let's take an example to illustrate the decoding process. Suppose the computer_position input is 0100, which corresponds to pos4. Here's how the position_decoder would behave:

- The computer_position input is compared with each possible position using logic gates.
- The comparison for pos4 (0100) is successful, indicating a match.
- The PC_en signal for pos4 is activated, allowing the computer to play at that position.

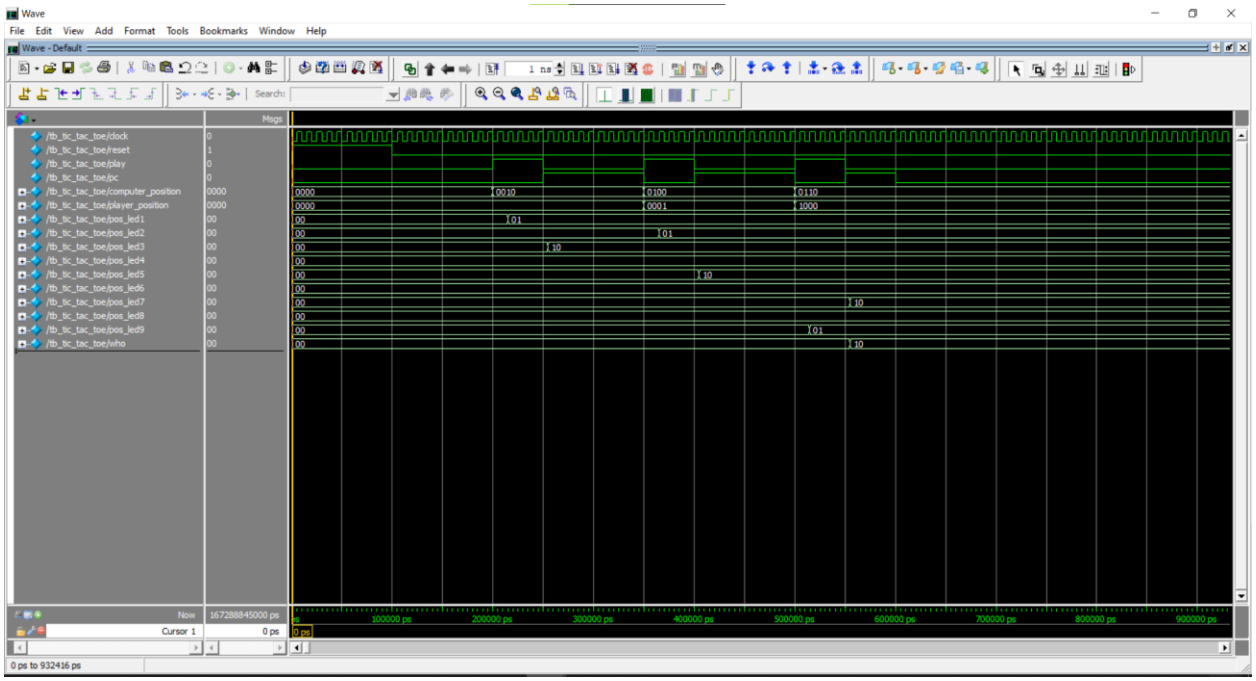
At the same time, all other PC_en signals for positions pos1, pos2, pos3, pos5, pos6, pos7, pos8, and pos9 would be deactivated. By using a decoder, the code can easily control and manipulate individual positions on the board based on the provided input signal.

7. `winner_detector`: This module is responsible for detecting the winner of the Tic Tac Toe game based on the current positions on the game board. The module takes the current positions of the Tic Tac Toe game board (pos1 to pos9) as inputs. It compares these positions to determine if any winning condition has been met. It does this by checking for three positions in a row, either horizontally, vertically, or diagonally. The module uses several intermediate wire signals to represent each possible winning line on the game board. If any of these lines are true (indicating a winning condition), the win output signal is set to 1. Additionally, the who output signal indicates which player has

won. If there is a winner, it checks which line is true and assigns the corresponding player's position (pos1 to pos9) to the who signal.

8. `winner_detect_3`: This module is designed to detect the winner in the tic-tac-toe game. It takes three positions on the game board as inputs and determines if there is a winning combination. The module checks for various winning combinations based on the values of the input positions ((1, 2, 3), (4, 5, 6), (7, 8, 9), (1, 4, 7), (2, 5, 8), (3, 6, 9), (1, 5, 9) and (3, 5, 7)). If a winning combination is found, the win signal is set to 1, and the who signal indicates the winner. The who signal is set to 2'b01 if the player wins and 2'b10 if the computer wins.

Picture of the simulation:



Problems we faced and how we overcame them:

- Our initial problem was keeping track of all possible game-breaking scenarios (illegal moves, determining the winner, no available places to play, etc.). It was challenging for us to ensure that all the rules of the game were applied.
- We also spent a lot of time discussing the game's states (the player’s state, the computer’s state, the idle state and the game over) and what happens when the states are a changing from one to another.
- To overcome this problem, we started thinking as if we were playing the game, taking the time to consider every possible move the player may make and all the rules that must be applied to the game, to ensure that we covered each case and every condition.
- Additionally, we spent a lot of effort in creating the test bench to make sure that we considered every scenario and that our code worked well and covered all the scenarios necessary.