Digital Electronics Project

TicTacToe

Saïd Hormat-Allah, Marzouk Ouro-Gomma, Lei Yang, Renaux Ntakirutimana
Group 8

May 11, 2022

1 Introduction

This project consists of in implementation of the popular tic-tac-toe game using a led matrix and buttons on a fgpa.

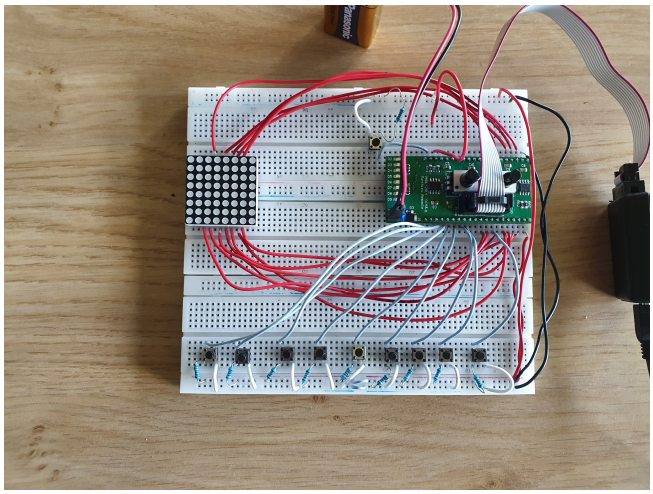


Figure 1: Tic-Tac-Toe Game

2 Rules

The player begins the game by being `X` (or `O`) and players take turns by putting their marks in empty places. The first player to get 3 of their mark in a row (may this be : horizontal, vertical or oblique) is the winner. In the case of a draw, no more marks can be put before the reset button is pressed.

3 Hardware used

For this project, we used :

- a 8x8 LED matrix;
- 10 buttons, 9 for each case of the tic-tac-toe and the last one for the reset;

- a 9v battery;
- Wires;
- resistors

4 I/O description

We use 10 buttons as inputs, where in 9 are used to control the user inputs and place marks at wanted position and the 10th to reset the game.

Concerning the outputs, we use 12 pins to display the LED matrix to show the game's state. Thus, the following signals will be used in the game entity :

- clock1 : input clock 1 signal coming from the 555 timer of the development board, this signal will be used as the main clock of the CPLD

  → std_logic ;
- buttons : input signal of 9 buttons associated with the 9 different LEDs of the game

  → std_logic_vector(0 to 8) ;
- reset : input signal of one buttons associated with the objective to reset the game

  →std_logic ;
- row : output signal of the 8 LEDs to display the `O` or `X`

  → std_logic_vector(0 to 5) ;
- cols : output signal of the 8 LEDs to display the `O` or `X`

  → std_logic_vector(0 to 5) ;
- victory_led : output signal to know the winner of the game.

  → std_logic ;

The code of the entity is represented by :

```
entity tic_tac_toe is port(
    clock1 : in std_logic;
    button : in std_logic_vector(0 to 8);
    row : out std_logic_vector (0 to 5);
    cols : out std_logic_vector (0 to 5);
    reset : in std_logic;
    victory_led : out std_logic
);
end entity tic_tac_toe;
```

5 State diagram

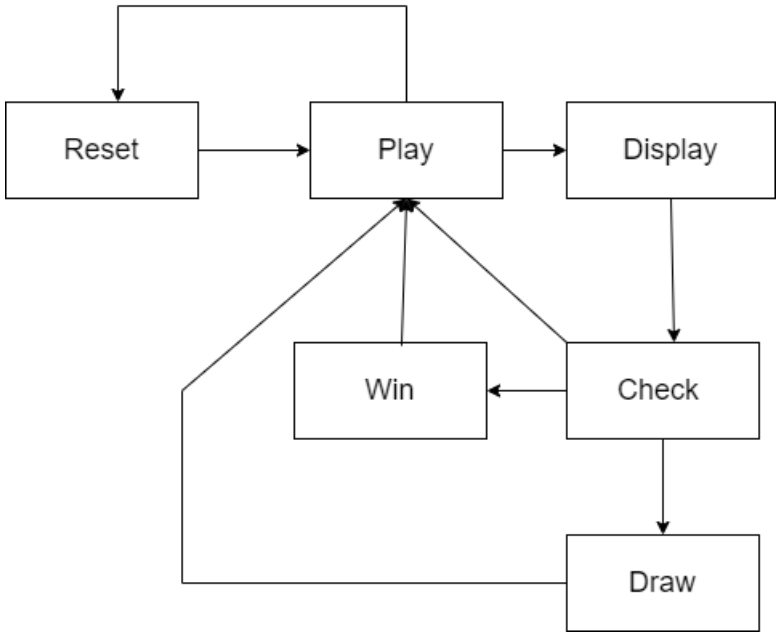


Figure 2: Tic-Tac-Toe Game

At any given point of the game, the reset button can be pressed and the game restarted. Otherwise the basic process. of the game is that the win condition is checked after each player input. the game can either end with a draw or with a victory.

6 Code Description

Before all, we can't enable all the led at the same time so we need to refresh the led matrix fast enough for it to seem that all the leds are on at the same time.

For this , we need :

```
signal   clk_counter : integer := 0;
signal row_counter : unsigned (5 downto 0);
COUNTER_PROC : process(clk)
begin
    if rising_edge(clk) then
                row_counter <= row_counter+1;
        end if;
     end if;
end process;

rows <= matrix(to_integer(row_counter));
cols <= (others => '0');
cols(to_integer(row_counter)) <= '1';
```

Now,we have the code to refresh the led matrix. We can start to implement tic tac toe logic.

We need for the game a matrix to have a map of the led matrix to display O or X.We need to know the current player if it is the player one or player 2. The variable x,y to know where to

display in the led matrix and finally the signal state. The vector tab to check the victory of a player.

```
    -- Used to refresh rows at a given time
    signal row_counter : unsigned (5 downto 0);

    -- Matrix storing the state of the LED matrix at a given time
    type matrix_type  is array (0 to 5) of std_logic_vector(0 to 5);
    signal matrix : matrix_type ;

    -- States initialised to init at the beginning
    type states is (reset_st, init, play, display, check);
    signal state : states := init;


    signal player : integer range 0 to 1 := 0;
    type grid_type is array(0 to 8) of integer range 0 to 2;
    signal grid : grid_type := (2,2,2,2,2,2,2,2,2);
```

The code for the reset we are going to start :

6.1 Reset

```
case state is
    when reset_st =>
        matrix <=
        (
            ('1', '1', '1', '1', '1', '1'),
            ('1', '1', '1', '1', '1', '1'),
            ('1', '1', '1', '1', '1', '1'),
            ('1', '1', '1', '1', '1', '1'),
            ('1', '1', '1', '1', '1', '1'),
            ('1', '1', '1', '1', '1', '1')
        );
        victory_led <= '0';
        state <= play;
        grid <=  (2, 2, 2, 2, 2, 2, 2, 2, 2);
        when init =>
            if(reset = '0') then
                victory_led <= '0';
                state <= play;

            end if;
```

For the reset , we need to initialize correctly the matrix of led and give a value for the current player and reset the vector too.

6.2 Play

Now, we can go to the play zone

```
when play =>
        if(buttons(0) = '0' and matrix(0)(0) = '1') then
            x <= 0;
            y <= 0;
```

```
            grid(0) <= player;
            state <= display;
        elsif (buttons(1) = '0' and matrix(2)(0) = '1') then
            x <= 2;
            y <= 0;
            grid(1) <= player;
            state <= display;
        elsif (buttons(2)  = '0' and matrix(4)(0) = '1') then
            x <= 4;
            y <= 0;
            grid(2) <= player;
            state <= display;
        elsif (buttons(3)  = '0'and matrix(0)(2) = '1') then
            x <= 0;
            y <= 2;
            grid(3) <= player;
            state <= display;
        elsif (buttons(4) = '0'and matrix(2)(2) = '1') then
            x <= 2;
            y <= 2;
            grid(4) <= player;
            state <= display;
        elsif (buttons(5)  = '0' and matrix(4)(2) = '1') then
            x <= 4;
            y <= 2;
            grid(5) <= player;
            state <= display;
        elsif (buttons(6)  = '0' and  matrix(0)(4) = '1') then
            x <= 0;
            y <= 4;
            grid(6) <= player;
            state <= display;
        elsif (buttons(7) = '0' and  matrix(2)(4) = '1') then
            x <= 2;
            y <= 4;
            grid(7) <= player;
            state <= display;
        elsif (buttons(8) = '0' and matrix(4)(4) = '1') then
            x <= 4;
            y <= 4;
            state <= display;
            grid(8) <= player;
        elsif (reset = '0') then

            state <= reset_st;
        end if;
```

Play respects the diagram he can go to display or go the reset. We need to place the value for the display and with this code we can just touch one time the button.

```
when display =>
    case player  is
        when '0' =>
            player <= '1';
            matrix(x)(y) <= '0';
```

```
            matrix(x+1)(y+1) <= '0';
            state <= check;
        when others =>
            player <= '0';
            matrix(x)(y) <= '0';
            matrix(x+1)(y) <= '0';
            matrix(x)(y+1) <= '0';
            matrix(x+1)(y+1) <= '0';
            state <= check;
        end case;
```

With this code, we display **O** or **X** and go to check to verify if there is a win or not.

6.3 Check

Now it's the time to check the vector we verify all the case and go the play again.

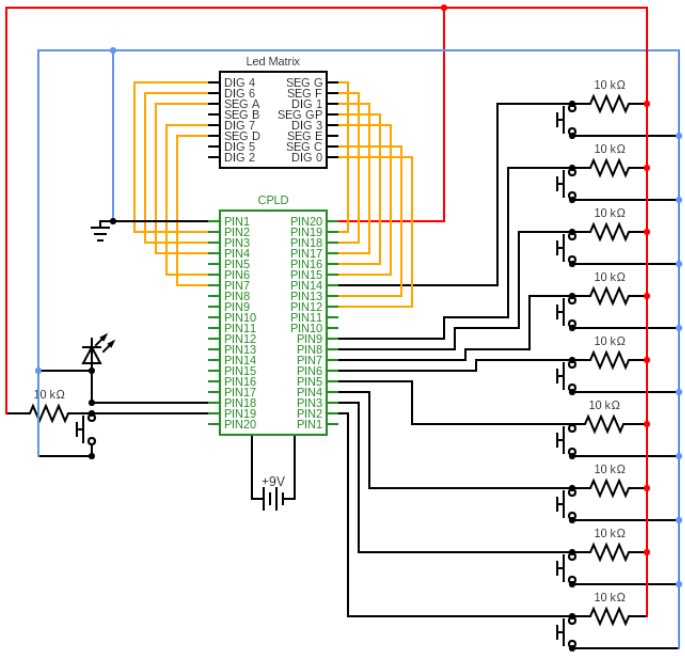
```
when check =>
     -- row conditions
    if ((grid(0) /= 2) and (((grid(0) = grid(1)) and (grid(1) = grid(2))) or
                            ((grid(0) = grid(4)) and (grid(4) = grid(8))) or
                            ((grid(0) = grid(3)) and (grid(3) = grid(6))))) then
         victory_led <= '1';

    elsif ((grid(3) /= 2) and ((grid(3) = grid(4)) and (grid(4) = grid(5)))) then
         victory_led <= '1';

    elsif ((grid(6) /= 2) and ((grid(6) = grid(7)) and (grid(7) = grid(8)))) then
         victory_led <= '1';


     -- col condition
    elsif ((grid(1) /= 2) and ((grid(1) = grid(4)) and (grid(4) = grid(7)))) then
         victory_led <= '1';


    elsif ((grid(2) /= 2) and ((grid(2) = grid(5)) and (grid(5) = grid(8)))) then
         victory_led <= '1';


     -- left diagonal cond

    elsif ((grid(2) /= 2) and ((grid(2) = grid(4)) and (grid(4) = grid(6)))) then
         victory_led <= '1';

    end if;
            state <= play;
```

We always return to play so we can reset the game at any given time.

7 Noticable choices of implementation

Uppon clicking on a button we verify thou the matrix if the button was clicked before. Even thou this operation is costly in terms of logical components. It's usage permits us to save in terms of hardware. It prevents us from using capacitors since bounce issues won't be issues anymore event uppon button clicks.

8 Schematics



9 Conclusion

In conclusion, connecting hardware and software to implement a tic-tac-toe is a process that takes a lot of thinking and planning, but with good reflexion and effort it's possible in spite of implementation issues and hardware imperfection.