FPGA Tic-Tac-Toe

Riya Shrestha, Jackie Fang, Rayhan Howlader, Nikola Ciric, Alexander Gaskins

Table of Contents

Introduction

The purpose of this project was to give the team exposure and first-hand experience in working with a field programmable gate array (FPGA) board at a higher level. The FPGA board is a reconfigurable integrated circuit (IC) that can implement a wide range of custom digital circuits which can be used to create optimized circuits for several purposes such as digital signal processing (DSP), machine learning, and cryptocurrency mining. The language used to implement the tic tac toe game was VHDL (very-high-speed integrated circuit hardware description language), which is used in electronic design automation to describe digital and mixed-signal systems such as field-programmable gate arrays and integrated circuits.

For this specific project, the team intended to recreate the popular pen-and-paper game tic-tac-toe with the FPGA board. To replicate the game, the team decided to go off the FPGA board by designing the game on bread boards. The idea was to place 9 RGB LED lights on the board and connect each light with a button. The players would use the buttons to control the lights till a winner was found and the 10th LED would go on showing the winning color. However, when this attempt was unsuccessful, the team quickly pivoted to designing the game on an external keyboard and the game being displayed on a monitor connected to the board via VGA cable. The finalized design resulted in the FPGA board being used as the controller instead of a keyboard for simplicity.

Diagrams

This section includes the diagrams made by the team and generated by Vivado

Breadboard

The initial design for this project was meant to be played by two players on a breadboard. All the buttons (inputs) on the left would be connected to the FPGA board which would control the RGB LED lights (outputs). The block diagram for this design can be seen in the figure below and the figure below is the schematic designed by Vivado.

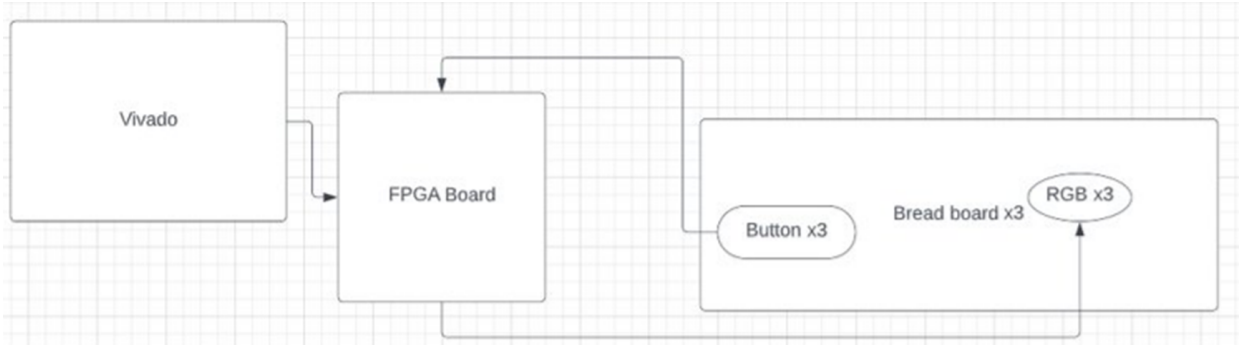


Figure 1: Block diagram of the bread board game design

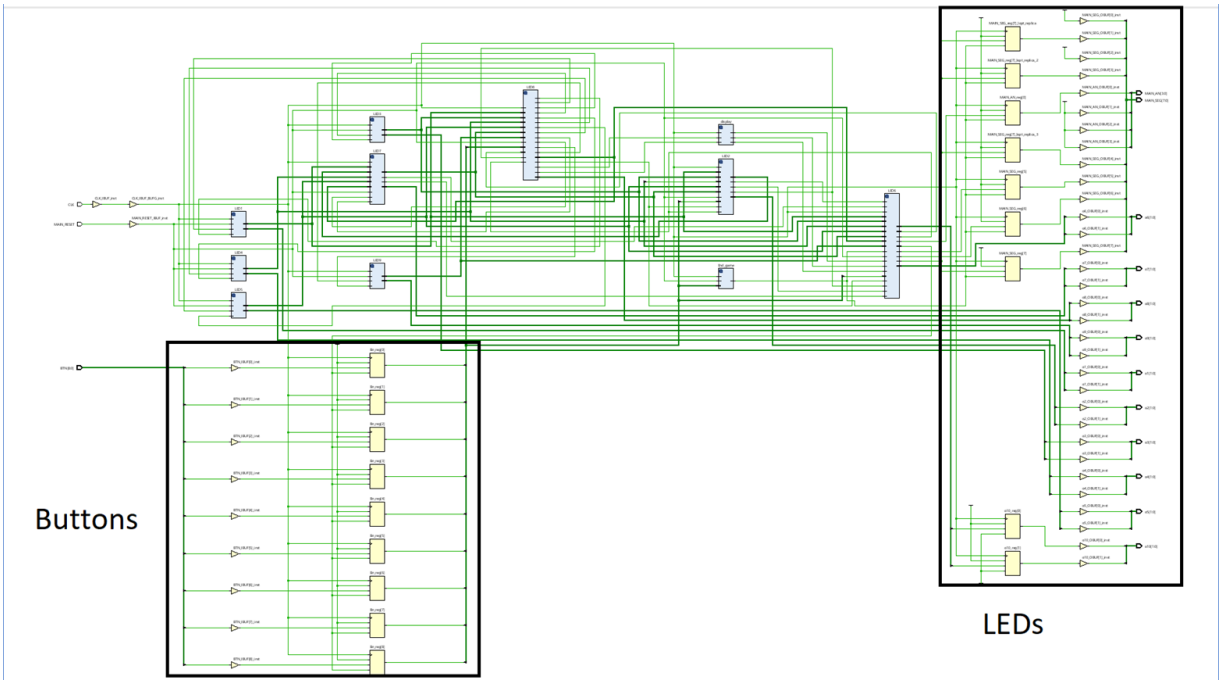


Figure 2: Schematic of the breadboard game design

Monitor display

Even after several attempts, when the team was unable to get the game working on the breadboard design, the team pivoted to this design. The idea here was similar to the previous design as the keys on the keyboard are seen as player inputs that connect to the FPGA board and the monitor display is the output. The figure below shows the block diagram for this game design and the figure after that shows the schematic generated by Vivado.

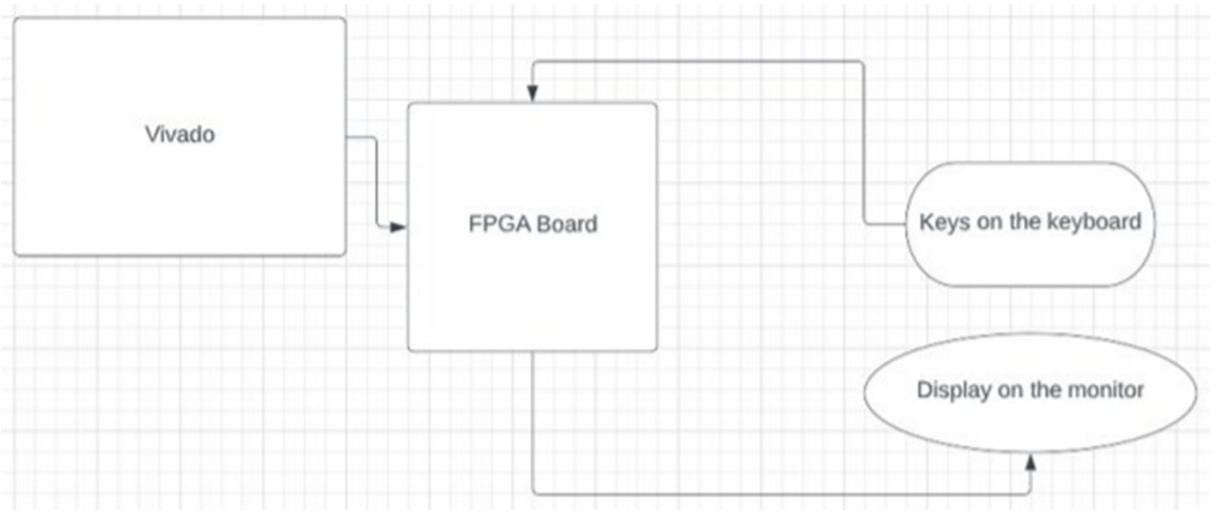


Figure 3: Block diagram of the monitor display game design

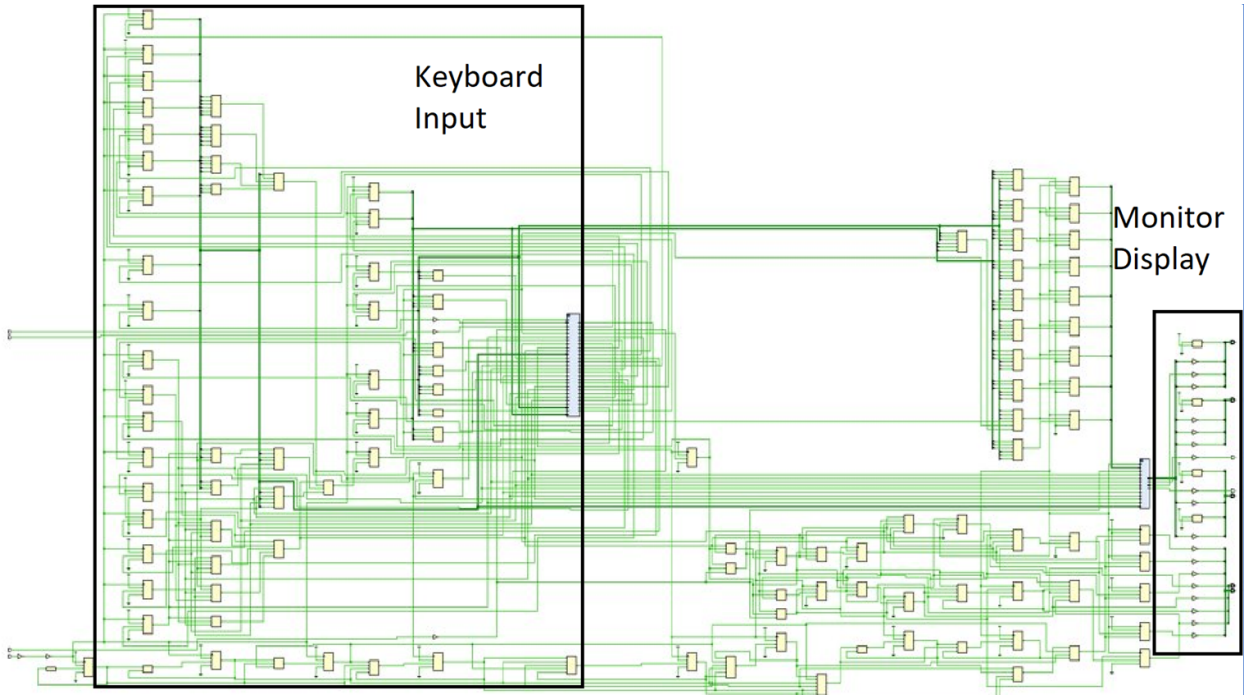
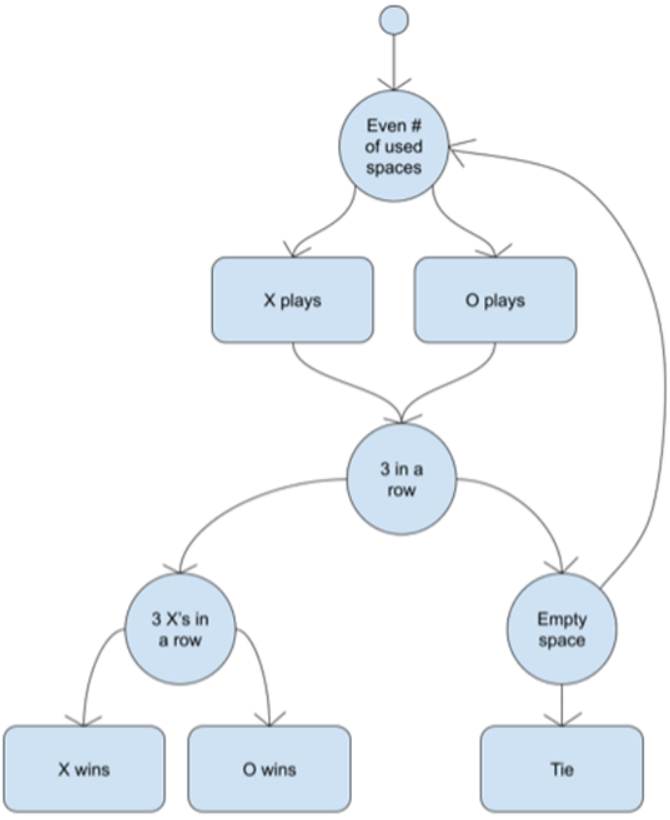


Figure 4: Schematic of the monitor display game design

State Machine

The classic TicTacToe game has two main states, one where a player wins and the other where there is a draw. The figure below shows that if either player reaches 3 in a row they win and if no player reaches 3 in a row, it's a tie. In this system, both the input and the state impact the system output which makes this a Mealy machine. In this game, there are 8 different ways to win the game, getting 3 in a row horizontally, vertically, or diagonally. Since there are 3 rows and 3 columns the horizontal and vertical apply each time and the diagonal applies twice going from left to right and right to left.



Input 1	Input 2	Input 3
Input 4	Input 5	Input 6
Input 7	Input 8	Input 9

Figure 5: State machine diagram

Breadboard TicTacToe

VHDL Architecture

The VHDL architecture can best be seen in the main board file which serves to combine all the individual components and modules written before, to work together. There are four main components that will be used to drive the game: the single led controller, multi led controller, victory, and finally tie conditions. Each of these components have their own input and output signals and they are tied together in the main board file. Now that the general architecture is described some of these components are made of other basic digital logic components. In this case we found that the D-flip flop was mostly used to provide the basic function of the single-led controller and executing a tie condition. Finally we have also utilized the exclusive or basic logic gate for processing of the multi led controller. The main game board file can be seen below in figure below.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Main_Game_Board is
  port (CLK     : in std_logic;
        BTN     : in std_logic_vector(8 downto 0);
        MAIN_RESET : in std_logic;
        o1        : out std_logic_vector(1 downto 0);
        o2        : out std_logic_vector(1 downto 0);
        o3        : out std_logic_vector(1 downto 0);
        o4        : out std_logic_vector(1 downto 0);
        o5        : out std_logic_vector(1 downto 0);
        o6        : out std_logic_vector(1 downto 0);
        o7        : out std_logic_vector(1 downto 0);
        o8        : out std_logic_vector(1 downto 0);
        o9        : out std_logic_vector(1 downto 0);
        o10       : out std_logic_vector(1 downto 0);
        MAIN_AN     : out  STD_LOGIC_VECTOR(3 downto 0);
            MAIN_SEG   : out  STD_LOGIC_VECTOR(7 downto 0));
end Main_Game_Board;
```

Figure 6: Main game board ports for the inputs

This declaration of the entity of the main game board is used to combine everything we wrote together. First we declared all of our inputs and outputs. As it can be seen we will have 9 leds that represent the LED outputs and one extra that lights up for when a player wins. Next as mentioned many times before in order to connect all the modules written, this will be done using VHDL components. This will be explained in a later section of this report but can be seen in the figure below.

```
component Single_LED_Controller is
  port (Clock       : in std_logic;
        Button      : in std_logic;
        Reset       : in std_logic;
        Player1     : in std_logic;
        Player2     : in std_logic;
        LED_Output    : out std_logic_vector(1 downto 0);
        Handle_Input  : out std_logic_vector(1 downto 0));
end component;

component Multi_LED_Handle is
  port (Reset   : in std_logic;
        Input1  : in std_logic_vector(1 downto 0);
        Input2  : in std_logic_vector(1 downto 0);
        Input3  : in std_logic_vector(1 downto 0);
        Input4  : in std_logic_vector(1 downto 0);
        Input5  : in std_logic_vector(1 downto 0);
        Input6  : in std_logic_vector(1 downto 0);
        Input7  : in std_logic_vector(1 downto 0);
        Input8  : in std_logic_vector(1 downto 0);
        Input9  : in std_logic_vector(1 downto 0);
        Output  : out std_logic_vector(1 downto 0));
end component;

component Victory is
    Port ( in1 : in  STD_LOGIC_VECTOR(1 downto 0);
           in2 : in  STD_LOGIC_VECTOR(1 downto 0);
           in3 : in  STD_LOGIC_VECTOR(1 downto 0);
           in4 : in  STD_LOGIC_VECTOR(1 downto 0);
           in5 : in  STD_LOGIC_VECTOR(1 downto 0);
           in6 : in  STD_LOGIC_VECTOR(1 downto 0);
           in7 : in  STD_LOGIC_VECTOR(1 downto 0);
           in8 : in  STD_LOGIC_VECTOR(1 downto 0);
           in9 : in  STD_LOGIC_VECTOR(1 downto 0);
          Tie : out STD_LOGIC;
           P1win : out  STD_LOGIC;
           P2win : out  STD_LOGIC);
end component;
```

Figure 7: VHDL components used for the game

The main components used here are the victory conditions which take in nine inputs, and the outputs are a Tie, and two player variables which will be used to provide some signals to be put into separate processes. Next the multi-LED controller has 9 inputs which will be put through an exclusive or gate to determine who goes next. Finally the last component in the main architecture of the game is the single LED controller. This controller will be used to control each individual

LED. It will have five inputs for both players, the clock, button, and reset. It will also have two outputs one being the output for the LED and the other output will be used for the multi LED controller. The single led controller has its own architecture consisting of a D- flip flop and buttons using 9 D-flip flops for processing inputs. Finally to tie everything together we need to create signals and pass them between the inputs and outputs of other modules. After the signals are processed by the other modules they get sent to the outputs of the main game board as seen below.

```
signal PL: std_logic_vector(1 downto 0) := "10";
signal H1, H2, H3, H4, H5, H6, H7, H8, H9 : std_logic_vector(1 downto 0) := "00";
signal T, P1, P2 : std_logic;
signal An : STD_LOGIC_VECTOR (3 downto 0);
signal Seg_temp : STD_LOGIC_VECTOR (7 downto 0);
signal Bn : std_logic_vector (8 downto 0);
signal Judgement_temp : std_logic;

begin
  process(T, BTN, CLK) is
  begin
    if (rising_edge(CLK)) then
    if (T = '0' or Judgement_temp = '1') then
      Bn <= "000000000";
    else
      Bn(0) <= BTN(0);
      Bn(1) <= BTN(1);
      Bn(2) <= BTN(2);
      Bn(3) <= BTN(3);
      Bn(4) <= BTN(4);
      Bn(5) <= BTN(5);
      Bn(6) <= BTN(6);
      Bn(7) <= BTN(7);
      Bn(8) <= BTN(8);
    end if;
    end if;
  end process;

  LED1 : Single_LED_Controller port map (CLK, BN(0), MAIN_RESET, PL(0), PL(1), o1, H1);
  LED2 : Single_LED_Controller port map (CLK, BN(1), MAIN_RESET, PL(0), PL(1), o2, H2);
  LED3 : Single_LED_Controller port map (CLK, BN(2), MAIN_RESET, PL(0), PL(1), o3, H3);
  LED4 : Single_LED_Controller port map (CLK, BN(3), MAIN_RESET, PL(0), PL(1), o4, H4);
  LED5 : Single_LED_Controller port map (CLK, BN(4), MAIN_RESET, PL(0), PL(1), o5, H5);
  LED6 : Single_LED_Controller port map (CLK, BN(5), MAIN_RESET, PL(0), PL(1), o6, H6);
  LED7 : Single_LED_Controller port map (CLK, BN(6), MAIN_RESET, PL(0), PL(1), o7, H7);
  LED8 : Single_LED_Controller port map (CLK, BN(7), MAIN_RESET, PL(0), PL(1), o8, H8);
  LED9 : Single_LED_Controller port map (CLK, BN(8), MAIN_RESET, PL(0), PL(1), o9, H9);

  MLEDH : Multi_LED_Handle port map(MAIN_RESET, H1, H2, H3, H4, H5, H6, H7, H8, H9, PL);

  Victory_module : victory port map(H1, H2, H3, H4, H5, H6, H7, H8, H9, T, P1, P2);

  display : seg port map(PL, CLK, An, Seg_temp);

  End_game : End_tie port map(Bn, MAIN_RESET, CLK, Judgement_temp);
```

Figure 8: Input signals being processed

VHDL Models

Throughout our entire design process we wanted to make it as modular as possible for easy debugging and management. The dataflow for our design all falls onto the basic component called the single LED controller. All of the logic handled by this controller is done using D-Flops. D-flip flops are very useful because they can store data for a certain period of time and then output it. The main goal of the controller is to determine what color the LED should display. In our design we will be red and blue colors to represent the two players. As mentioned in the architecture section the single led controller will have an output going to the physical led and an output going into the multi-LED controller. This output will be used to help decide which player is next seen in the figure below.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Multi_LED_Handle is
  port (Reset   : in std_logic;
        Input1  : in std_logic_vector(1 downto 0);
        Input2  : in std_logic_vector(1 downto 0);
        Input3  : in std_logic_vector(1 downto 0);
        Input4  : in std_logic_vector(1 downto 0);
        Input5  : in std_logic_vector(1 downto 0);
        Input6  : in std_logic_vector(1 downto 0);
        Input7  : in std_logic_vector(1 downto 0);
        Input8  : in std_logic_vector(1 downto 0);
        Input9  : in std_logic_vector(1 downto 0);
        Output  : out std_logic_vector(1 downto 0));
end Multi_LED_Handle;

architecture Dataflow of Multi_LED_Handle is

signal temp : std_logic_vector(9 downto 0);


begin

  -- XOR First level
  temp(0) <= input1(0) XOR input1(1);
  temp(1) <= input2(0) XOR input2(1);
  temp(2) <= input3(0) XOR input3(1);
  temp(3) <= input4(0) XOR input4(1);
  temp(4) <= input5(0) XOR input5(1);
  temp(5) <= input6(0) XOR input6(1);
  temp(6) <= input7(0) XOR input7(1);
  temp(7) <= input8(0) XOR input8(1);
  temp(8) <= input9(0) XOR input9(1);

  -- XOR Second level
  temp(9) <= temp(0) XOR temp(1) XOR temp(2) XOR temp(3) XOR temp(4) XOR temp(5) XOR temp(6) XOR temp(7) XOR temp(8);


  Output(0) <= temp(9);
  Output(1) <= not(temp(9));



end Dataflow;
```

Figure 9: Output being sent to the RGB LEDs

This controller will take in 9 inputs and using the exclusive or gate it can determine who goes next. The exclusive or gates can achieve this because if there is an odd number of 1's it will output 1 and if there is even number of 1's it will output a 0. Using this idea it can be seen that it is fairly simple to then calculate which player's turn is next. Using these two basic components and their output signals they can then be used by the main game board module described in the previous section to drive the whole-Tic-Tac Toe logic.

VHDL Component Reuse

In this section we will go more depth into the components used and explain their individual logic and analyze their outputs as well. As previously mentioned before, the single LED controller is used a few times for our design due to the fact that we have 9 LEDs that need to be controlled. The whole code defining this controller can be seen in the figure below.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Single_LED_Controller is
  port (Clock   : in std_logic;
        Button  : in std_logic;
        Reset   : in std_logic;
        Player1 : in std_logic;
        Player2 : in std_logic;
        LED_Output  : out std_logic_vector(1 downto 0);
        Handle_Input : out std_logic_vector(1 downto 0));
end Single_LED_Controller;

architecture Behavioral of Single_LED_Controller is

component D_FlipFlop is
  port (Enable: in std_logic;
        D     : in std_logic;
        Reset : in std_logic;
        Clock : in std_logic;
        Q     : out std_logic);
end component;

signal temp1, temp2 : std_logic;

begin

  D_FF_Main_1 : D_FlipFlop port map(Button, Player1, Reset, Clock, temp1);
  D_FF_Main_2 : D_FlipFlop port map(Button, Player2, Reset, Clock, temp2);

  LED_Output(0) <= not(temp1);
  LED_Output(1) <= not(temp2);

  Handle_Input(0) <= temp1;
  Handle_Input(1) <= temp2;

end Behavioral;
```

Figure 10: Defining a single LED controller

The single LED controller will be used to control one LED. It will be composed of the fundamental component of the D-Flip flip. Since we have decided to use a single RGB LED to

represent the players instead of two single color LEDS we will need two D-Flip Flops for each LED. The D-Flip Flop is used to store data at a predetermined time and hold it until it is needed. Each of the flip flops will share a clock, the input of the button, and a reset button. The press of the button will latch on the D value of each flip flop and output it.The Q inverse output will go directly to the LED while the normal Q output goes directly into the multi-LED controller. We will be using 9 of these flip flops for each button. In addition to the reuse of this component we also reused the D-Flip Flop many times. The definition of this component is shown below.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;


entity D_FlipFlop is
  port (Enable: in std_logic;
        D     : in std_logic;
        Reset : in std_logic;
        Clock : in std_logic;
        Q     : out std_logic);
end D_FlipFlop;

architecture Behavioral of D_FlipFlop is
begin
  process (D, Clock, Reset, Enable) is
  begin
    if (rising_edge(Clock)) then
    if (Enable = '1') then
      Q <= D;
    end if;
    if (reset = '1') then
      Q <= '0';
    else null;
    end if;
    end if;
  end process;
end Behavioral;
```

Figure 11: D-Flip Flops

The D-Flip Flops were the most reused components as they were used to handle the logic in a few other important aspects of our project such as the end tie module in addition to the single-led controller.

VHDL Digital Circuits

Having gone in depth into the basic building blocks of the FPGA TicTacToe implementation we can now use the outputs to tie them together with a signal to perform certain functions. These functions can be then formulated into various circuits that achieve our goal. The majority of the logic that goes into Tic-Tac-Toe is into analyzing the winning condition if there even are any. Victory conditions consist of: vertical, horizontal, and diagonal (total of 8 possible conditions). Using lots of "if " statements inside of processes it is possible to handle all 8 possible winning conditions. In addition to this we must analyze the player 1 and player 2 states to determine if there was a tie or an actual win. This is accomplished by a second process that compares the P1 and P2 win state to determine if there was a tie.This tie condition will be handled by an external module. Finally, we must pass the values stored in the signals to our three outputs being if player 1 or 2 won, or if there was a tie. The main declaration of the victory conditions can be seen in the figure below.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Victory is
    Port ( in1 : in  STD_LOGIC_VECTOR(1 downto 0);
           in2 : in  STD_LOGIC_VECTOR(1 downto 0);
           in3 : in  STD_LOGIC_VECTOR(1 downto 0);
           in4 : in  STD_LOGIC_VECTOR(1 downto 0);
           in5 : in  STD_LOGIC_VECTOR(1 downto 0);
           in6 : in  STD_LOGIC_VECTOR(1 downto 0);
           in7 : in  STD_LOGIC_VECTOR(1 downto 0);
           in8 : in  STD_LOGIC_VECTOR(1 downto 0);
           in9 : in  STD_LOGIC_VECTOR(1 downto 0);
          Tie : out STD_LOGIC;
           P1win : out  STD_LOGIC;
           P2win : out  STD_LOGIC);
end Victory;

architecture Behavioral of Victory is
signal P1, P2 : std_logic;
signal T : std_logic;
signal in1t, in2t, in3t, in4t, in5t, in6t, in7t, in8t, in9t : std_logic_vector(1 downto 0);


begin

  in1t <= in1;
  in2t <= in2;
  in3t <= in3;
  in4t <= in4;
  in5t <= in5;
  in6t <= in6;
  in7t <= in7;
  in8t <= in8;
  in9t <= in9;
```

Figure 12: Victory component for winning the game

As seen there are 9 inputs which will be tied back to the physical buttons. These 9 inputs will then be used by the process seen in figure 13. This process can be analyzed as a state machine as it changes the state of the whole game each time it runs.

```
process (in1t, in2t, in3t, in4t, in5t, in6t, in7t, in8t, in9t) is
begin
  --player 1 conditions

  -- horizontal
  if (in1t(1) = '1' and in2t(1) = '1' and in3t(1) = '1') then
    p1 <= '1';



  elsif (in4t(1) = '1' and in5t(1) = '1' and in6t(1) = '1') then
    p1 <= '1';


  elsif (in7t(1) = '1' and in8t(1) = '1' and in9t(1) = '1') then
    p1 <= '1';


  -- vertical
  elsif (in1t(1) = '1' and in4t(1) = '1' and in7t(1) = '1') then
    p1 <= '1';


  elsif (in2t(1) = '1' and in5t(1) = '1' and in8t(1) = '1') then
    p1 <= '1';


  elsif (in3t(1) = '1' and in6t(1) = '1' and in9t(1) = '1') then
    p1 <= '1';


  -- diagonal
  elsif (in1t(1) = '1' and in5t(1) = '1' and in9t(1) = '1') then
    p1 <= '1';

  elsif (in3t(1) = '1' and in5t(1) = '1' and in7t(1) = '1') then
    p1 <= '1';
  else
    p1 <= '0';
  end if;

  -- player 2 conditions

  -- horizontal
  if (in1t(0) = '1' and in2t(0) = '1' and in3t(0) = '1') then
    p2 <= '1';


  elsif (in4t(0) = '1' and in5t(0) = '1' and in6t(0) = '1') then
    p2 <= '1';
```

Figure 13: Victory conditions for winning the game

Finally to handle the tie condition when no player has won and the game will not accept any more inputs we had to create a new module. To prevent the user from changing their input we will use 9 D flip flops. D set to ‘1’ automatically to determine if the buttons have been pressed or not. The module output will get the result of all the outputs of the D flip flops AND gated

together. If the output of this module equals one, then the user won't be able to press any button and have it register.

On-Screen TicTacToe

VHDL Architecture

The functionality behind our on-screen design is driven by similar logic derived from the breadboard-LED design. In this design, nine separate components are handled together in one driver file conveniently titled *TicTacToe_shell.vhd*, which instantiates the behavior of individual components that allow the game to run when combined. The key component of the on-screen design is the graphics driver, as without this, we wouldn't be able to display anything on the screen in the first place. All of the code references a 25 MHz clock that coordinates actions made in intervals. Below is a screenshot of the VGA driver code:

```
component VGA_driver is
    port (  vclk_port       :   in  STD_LOGIC;
            Vsync_port      :   out STD_LOGIC;
            Hsync_port      :   out STD_LOGIC;
            video_on_port   :   out STD_LOGIC;
            pixel_x_port    :   out STD_LOGIC_VECTOR(9 downto 0);
            pixel_y_port    :   out STD_LOGIC_VECTOR(9 downto 0));
end component;
```

Figure 14: VGA Display Driver Code

The aforementioned clock variable (vclk_port) is initialized to run at 25 MHz, and the vertical and horizontal setup on the monitor is predefined to sync with the connected monitor, with the default resolution being 640x480 pixels. To control the display area, *video_on_port* is defined to be set to 1 when the cursor is in the display area, and 0 when outside. Finally, the x and y coordinates of each pixel are handled from the *pixel_x_port* and *pixel_y_port* vectors.

Following the display driver code, the main game logic (Figure 15) is handled via the logic handler component that is run through the TicTacToe_shell with a focus on the same general characteristics as the breadboard-based functionality. There are two players, defined as p1 and p2. Player one is set to the color red via applying constraints, and player two is green. The benefit of using a monitor instead of an array of LED lights is the obvious increase in aesthetic, where the corresponding player selection is signified by an 'X' or an 'O' as opposed to a differently colored LED light blocked by a monstrosity of wires. In addition to handling the properties of player one and player two, the game logic also handles the movement of the cursor, as well as selection of a panel and winning conditions. The output ports are what handle the on-screen process. They include two 8-bit vectors for the blocks that each player has marked (p1_port, p2_port), a 4-bit vector for the location of the selecting frame (sf_port), a signal for the color of the player dropping next (sf_color_port), two 8-bit vectors for the blocks that make each player win (p1_win_port, p2_win_port).

```
component game_logic is
    port (  clk_port     :   in std_logic;
            start_port   :   in std_logic;
            reset_port   :   in std_logic;
            drop_port    :   in std_logic;
            up_port      :   in std_logic;
            down_port    :   in std_logic;
            left_port    :   in std_logic;
            right_port   :   in std_logic;
            p1_port      :   out std_logic_vector(8 downto 0);
            p2_port      :   out std_logic_vector(8 downto 0);
            sf_port      :   out std_logic_vector(3 downto 0);
            sf_color_port:   out std_logic;
            p1_win_port  :   out std_logic_vector(8 downto 0);
            p2_win_port  :   out std_logic_vector(8 downto 0));
end component;
```

Figure 15: On-Screen Panel Selection and Logic

The "pixel_generation" component is then defined. This component is used to generate the pixels that make up the images displayed on the screen. The component has a number of input and output ports. The input ports include the clock signal (clk_port), the horizontal sync signal (Hsync_port), the vertical sync signal (Vsync_port), the pixel x coordinate (pixel_x_port), the pixel y coordinate (pixel_y_port), and the signals for the blocks that each player has dropped a chess piece (p1_port, p2_port). The output ports include the pixel color (color_port) and the transparency of the pixel (alpha_port).

```
pixel_generation_datapath: pixel_generation
port map(
    pixel_x_port    => x,
    pixel_y_port    => y,
    video_on_port   => video_on,
    p1_port         => p1,
    p2_port         => p2,
    sf_port         => sf,
    sf_color_port   => sf_color,
    p1_win_port     => p1_win,
    p2_win_port     => p2_win,
    color_port      => color_ext_port);

end behavior;
```

Figure 16: Pixel Generation Process

VHDL Models

The "game_logic" component implements the game logic using a state machine. The state machine has a number of states that correspond to different stages of the game. For example, there may be a state for when the game is waiting for a player to make a move, a state for when a player is selecting a tile, a state for when the game is checking for a win, etc. The state machine transitions between these states based on the input signals and the current state of the game. Consider, for instance, when the start signal is asserted (i.e., goes from low to high), the state machine may transition from the "waiting" state to the "player 1 turn" state. When the drop signal is asserted, the state machine may transition from the "player 1 turn" state to the "checking for win" state. If the game is in the "checking for win" state and it is determined that player 1 has won, the state machine may transition to the "player 1 wins" state. The output signals of the "game_logic" component are generated based on the current state of the state machine.

To verify the functionality of the pixel generation, a testbench was set up. The testbench includes the "pixel_generation" component and defines a number of signals that are used to provide input to the component and check its output. The signals include *pixel_x*, *pixel_y*, *video_on*, *p1*, *p2*, and *color*. The *pixel_x* and *pixel_y* signals represent the x and y coordinates of the pixel being generated, the *video_on* signal indicates whether the video is currently on, the p1 and p2 signals represent the blocks that each player has dropped a chess piece, and the color signal represents the color of the pixel being generated.The testbench also includes a clock signal (clk) and a constant (clk_period) that define the period of the clock. The *clk_period* constant is set to 40 nanoseconds, which means that the clock frequency is 25 MHz.The testbench includes a process called clk_process that generates the clock signal. The process waits for half of the clock period, then sets the clock signal to '1', waits for the other half of the clock period, then sets the clock signal back to '0'. This creates a clock signal with a frequency of 25 MHz.The testbench also includes a process called *stim_proc* that provides input stimuli to the "pixel_generation" component. The process waits for two clock periods, then sets the *video_on* signal to '1'. This tells the "pixel_generation" component to start generating pixels. The process then goes into an infinite loop.

```
stim_proc: process
begin
    wait for 2*clk_sys_period;
    video_on <= '1';

    wait;
end process stim_proc;
```

Figure 17: Testbench Stimulus Process

VHDL Component Reuse

In the TicTacToe_shell architecture, several components are used to implement the TicTacToe game. These components are designed and verified separately, and are then "reused" in the TicTacToe_shell design.For example, the system_clock_generator component is used to generate a 25 MHz clock signal from the input clock signal. This component has been designed and verified independently, and can be used in other designs as well.Similarly, the button_interface component is used to debounce the button signals and generate debounce button signals. This component has also been designed and verified independently, and can be reused in other designs.The VGA_driver, game_logic, and pixel_generation components are also examples of components that have been designed and verified independently and are being reused in the TicTacToe_shell design.By using these pre-designed and pre-verified components, the TicTacToe_shell design can be implemented more efficiently and with higher reliability.

VHDL Digital Circuits

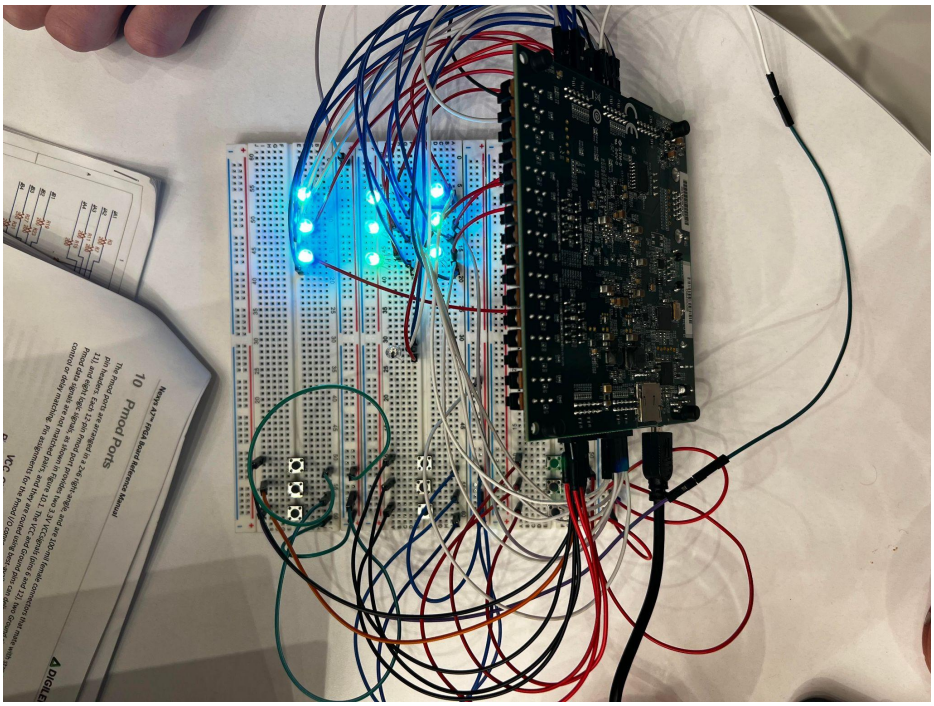
The FPGA TicTacToe game consists of several components that are interconnected to form the TicTacToe game. These components include a clock generator circuit, a button interface circuit, a VGA driver circuit, a game logic circuit, and a pixel generation circuit.The clock generator circuit generates a 25 MHz clock signal from the input clock signal. The button interface circuit denounces the button signals and generates debounce button signals. The VGA driver circuit generates the horizontal and vertical sync signals and the video data for the VGA display. The game logic circuit implements the game logic for the TicTacToe game, including keeping track of the game state, handling player input, and determining the winner. The pixel generation circuit generates the pixel data for the VGA display based on the game state and the x and y coordinates

of the pixel.The FPGA TicTacToe game operates as follows: The input clock signal is passed through the clock generator circuit to generate the 25 MHz clock signal. The button interface circuit denounces the button signals and generates debounce button signals. The debounce button signals are passed to the game logic circuit, which updates the game state based on the player input. The game logic circuit also generates the data for the selecting frame, which indicates the location on the game board where the next mark will be placed. The game logic circuit also generates the data for the player marks, which indicates the locations on the game board where each player has placed their pieces. The game logic circuit also generates the data for the winning marks, which indicates the locations on the game board that form a winning combination for either player. The pixel generation circuit generates the pixel data for the VGA display based on the game state, the x and y coordinates of the pixel, and the data from the game logic circuit. The VGA driver circuit generates the horizontal and vertical sync signals and the video data for the VGA display, which are then transmitted to the monitor through the VGA cable.

Testing

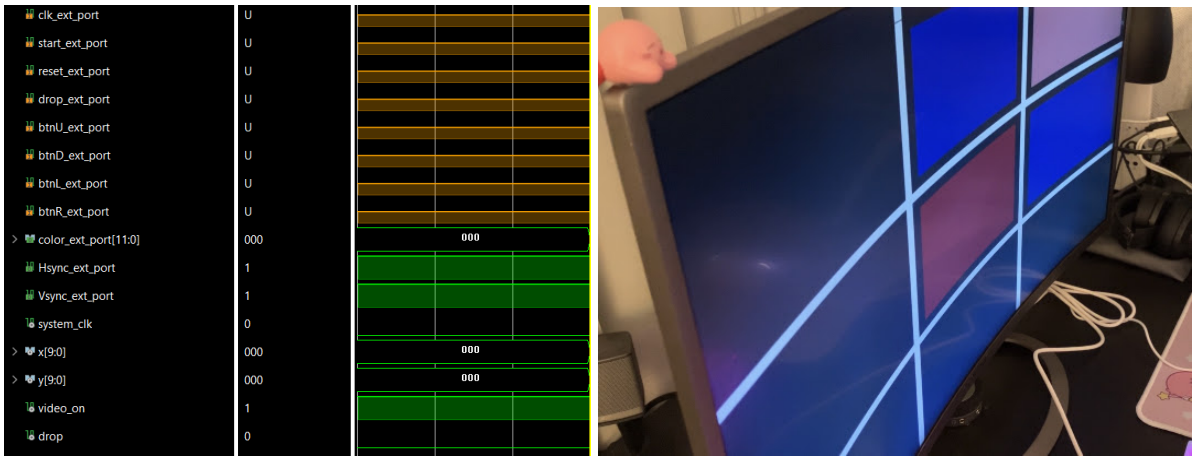
Breadboard

For testing, we first made the breadboard implementation for tic tac toe board. Although we had tried multiple attempts at rewiring and coding, it has not been successful.



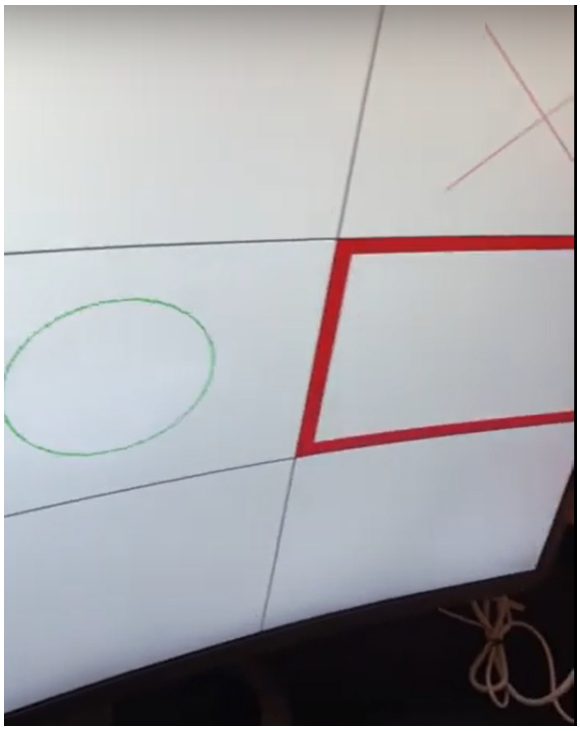
Monitor

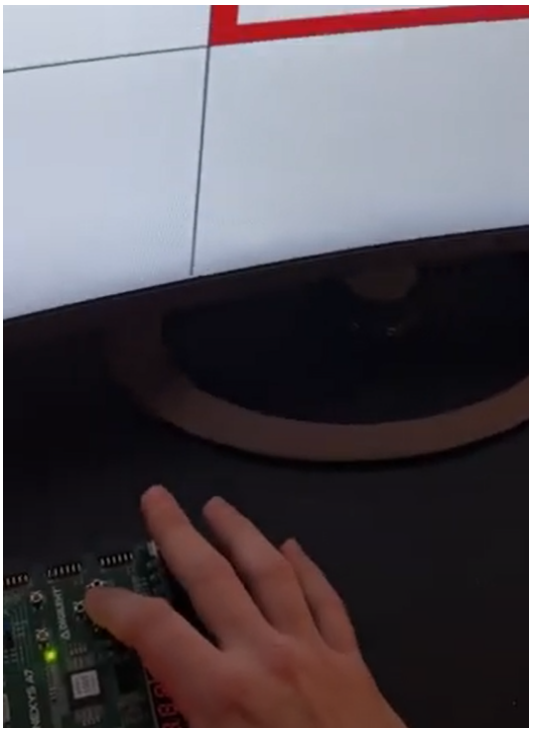
After successful testing using a VGA driven program, the project was shifted to on-screen objectives.



Final Result

We combined the button manipulation of the initial design with the on-screen UI in our second design to create our final working TicTacToe game:





Results

In the Bread board game design, the main flaw was that when the FPGA board was turned on, all the LEDs would light up with both player colors. These lights could not be controlled with the buttons we had connected or the switches. It seems as if the VHDL code was doing nothing and the FPGA was only supplying power to the LED lights. Even after several attempts, when this issue could not be fixed the group had to pivot keeping the due date of the project in mind.

The new idea for the game to work was to have it displayed on a monitor and take in key entries. The arrows on the keyboard allowed the players to move left, right, up, and down. To mark their spot on the tic tac toe board the player had to press enter. Instead of drawing out the normal Xs and Os the team decided to keep similar formatting to the breadboard and have the given box fill with the players colors. In this game player 1 was blue and player 2 was green and the system would automatically change the colors after each turn. Once the game ends with a win or a tie, the players are no longer allowed to move around the board and have to restart the game. If there is victory the blue LED light turns on on the FPGA board for the green winner and the green LED light for the blue winner. The final product is a slightly revised version of this that does not require a keyboard and a PS/2 adapter, but instead is more simplistic, using only the arrow buttons on the FPGA board, along with slight UI enhancements.

Conclusion

The original intention of our design was to create a two-player tic-tac-toe game on an FPGA board. It would use 9 buttons on the breadboard as inputs and 9 RGB LEDs as outputs. In addition to those, another LED was added to represent the winner as outputs. However, the team was not successful in completing this. The team was adamant on producing a game that could be played by 2 players, so the project was transitioned into a keyboard and monitor setup. The keyboard acted as the input rather than buttons and the game was displayed on a monitor rather than the LEDs. The game would be played by using the arrows to move around the board and enter a button to make a play on the selected space on the board.

There were many possibilities that can be recognized that can explain why the breadboard approach had not been implemented successfully. One is the wiring, as there may have been issues with the connections established within the VHDL code and the wiring on the breadboard that had prevented the VHDL code's logic from being recognized when running. Another can be derived from the code mainly within the part that recognizes the switches. As this was the main issue when testing. The logic was not being recognized, causing the lights to turn on at all times. Which meant that the value of the variables we assigned stayed as true the entire time and no input would change it. Although this way of designing had failed, the team was able to quickly pivot and create another design that was successful.