EE 314 DIGITAL CIRCUITS LABORATORY 2022-2023 SPRING TERM PROJECT REPORT

FPGA IMPLEMENTATION OF A 2D STRATEGY GAME

Ahmet Caner Akar
Electrical and Electronics Engineering Department
Middle East Technical University
Ankara, Turkey
e244228@metu.edu.tr

Osama Awad
Electrical and Electronics Engineering Department
Middle East Technical University
Ankara, Turkey
e248849@metu.edu.tr

İsmail Enes Bülbül
Electrical and Electronics Engineering Department
Middle East Technical University
Ankara, Turkey
e244263@metu.edu.tr

***Abstract*—This document is about the end-term project of EE314 Digital Circuits Laboratory, implementation of a 2D strategy game by using FPGA.**

***Index Terms*—FPGA, Verilog HDL, VGA driver, button debouncing, state-machine**

I. Introduction

Turn-based strategy games have been popular among people of all ages for decades because they provide a demanding gaming experience while being simple to understand. Tic Tac Toe and Battleships are two examples of such games. In this project, we will implement such a game called ”Triangles vs. Circles” in Verilog HDL by using Altera DE1-SoC FPGA board.

The aim of the project is to design a 2D strategy game by taking the necessary inputs from the user through the pushbuttons on the FPGA. Also, the game features a VGA interface, allowing players to see the board and their moves while the game logic operates in the background.

Our project consists of mainly three part: VGA, button debouncing, and game engine. Also, each main unit includes different sub-units, and they are explained in detail in the following section of the report.

II. Project Overview

A. VGA Module

In this project, we are expected to use the VGA interface of the FPGA to display the game board and some information related to the game. To do this, we decided to display the game with 640x480 resolution at 60 Hz. [1] The timing diagram is shown in Figure 1, below.

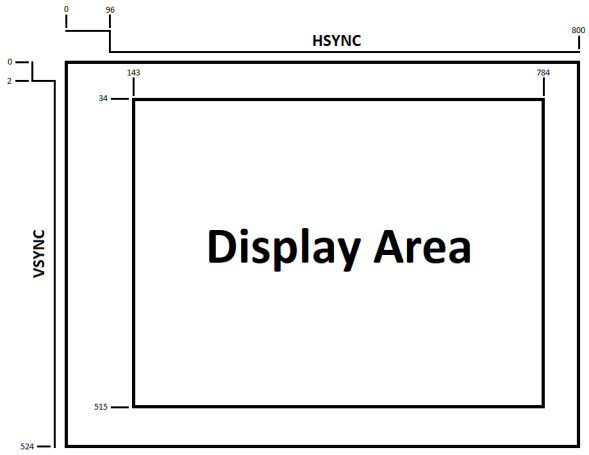

Fig. 1. Timing diagram for the VGA interface

From the timing diagram, we can see that we need two counters namely a horizontal counter and a vertical counter. To drive counters we need a clock with 25 MHz frequency. To obtain a clock signal with that frequency from the internal clock of the FPGA which has a 50 MHz frequency we implemented a clock divider circuit. After we obtained the clock signal with a 25 MHz frequency, we connected it to the horizontal counter. Horizontal counter counts at every clock cycle, and it counts from 0 to 800. Vertical counter counts 1 when the horizontal counter hits 800. We have two synchronization outputs namely HYSNC and VSYNC to tell the monitor beginning and ending of a row and frame respectively. HSYNC is high when the horizontal counter is between 0 and 96, and VSYNC is high when the vertical counter is between 0 and 2. Altera FPGA uses a digital-to-analog converter that takes 8-bit input for each

color channel and converts it to an analog signal that has a voltage between 0 and 5 Volts. [2]

When the horizontal counter is between 143 and 784, and the vertical counter is between 34 and 515, RGB values of the pixel at the point (hcounter - 143, ycounter - 34) are returned to the DAC of the FPGA. We implemented a *rgbSelector* module to get the RGB values for a given coordinate on the display. If the given coordinate is in the board region, the module finds the corresponding block coordinate and reads which piece is placed to that coordinate from the memory. Then it gets the RGB value from the memory module that corresponds to the piece. If the given coordinate is not in the board region but in a region that involves a text or picture, it reads the RGB value from the corresponding memory. If it is neither in the board region nor a picture or text region, then it returns a white color.

Since we need only 5 colors (white, black, red, green, and blue), we encoded and stored pictures that we used with 1 bit for each color channel. After the *rgbSelector* module returns RGB values with 1 bit for each channel, another module we called *vgaDecoder* converts it to output with 8 bits for each color channel and returns it to the DAC of the FPGA.

When the counters are not in the display area, 0 is returned to the DAC of the FPGA. An example of the display is shown in Figure 2, below.

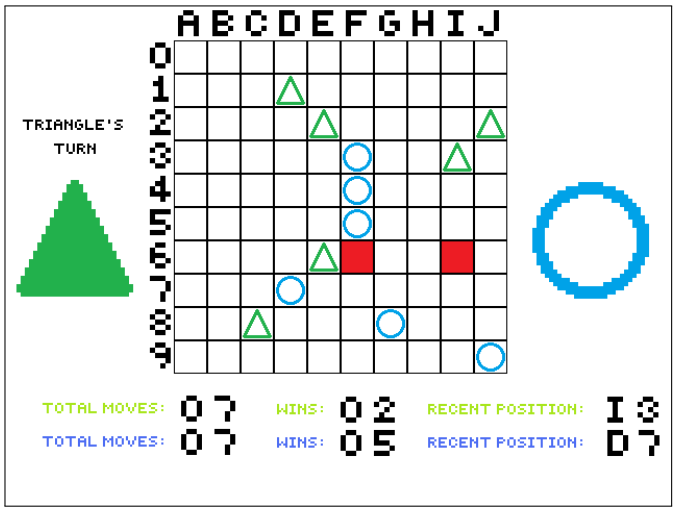

Fig. 2. Example of the display

B. Button Debouncing and Edge Detector

In this project, we need to get three inputs, **logic 1**, **logic 0**, and **activity**, from the user by using push buttons on the FPGA. However, due to mechanical and physical issues, pushbuttons often generate noisy signals called dirty bounces, and these bounces prevent us from properly triggering the program. Thus, to eliminate these undesirable effects, we used the *debouncer_delayed* module that makes a noisy pushbutton input signal to the ideal input case.

The working principle of the *debouncer_delayed* module is quite simple. When a button is pressed, the timer is going to count the elapsed time up to a predefined threshold parameter. If the timer hits the threshold value, the program concludes that the button has reached its steady state and has been pressed. Similarly, when the button is released and the steady state is reached, the program concludes that the button has been released.

After debouncing the button input, we designed another module called *edge_detector* to detect the negative and positive edges of the debounced input so that we will use the edge signals directly as button signals in the game controller.

The *button* module contains both *debouncer_delayed* and *edge_detector* modules so that the hierarchical design principle is followed throughout the project. Also, both of these modules, are written like a state machine to make it easier to implement condition-based and flexible code. The waveform simulation result of the *button* module in Quartus II is given in Figure 3.

Besides these, in the *button_top* module each button is defined separately in a hierarchical manner by using the *button* module. Also, we implemented another algorithm so that player can inform the game using the activity button that the player specified a wrong coordinate. Then, the game will allow the player to re-enter a coordinate. To do this, we used the negative and positive edge output of the activity button. One clock cycle after the negative edge is high, a counter starts counting the time until the positive edge is high. Thus, in this interval, if the counter hits a pre-defined value (3 seconds), the coordinate inputs will reset and the user must enter new coordinates. Otherwise, the activity button operates as usual.

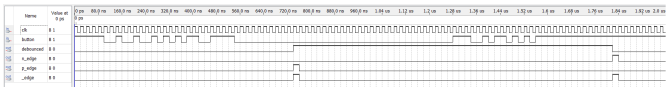

Fig. 3. Button module, waveform simulation result

C. Game Engine

As shown in Figure 4, the game engine can be modeled as a state machine that consists of 5 separate states: an input processing state (parse_inp), an invalid move checking state (invld_mv), a bookkeeping state, a state that checks if a win condition is satisfied, and finally, a state that resets the game board with appropriate modifications to the scores. Before discussing the states any further, it is helpful to understand the general encoding scheme of the board and shapes.

1) Shape Encoding and Board Layout: The shapes are encoded using 2 bits as follows:

- Empty (no shape): 00

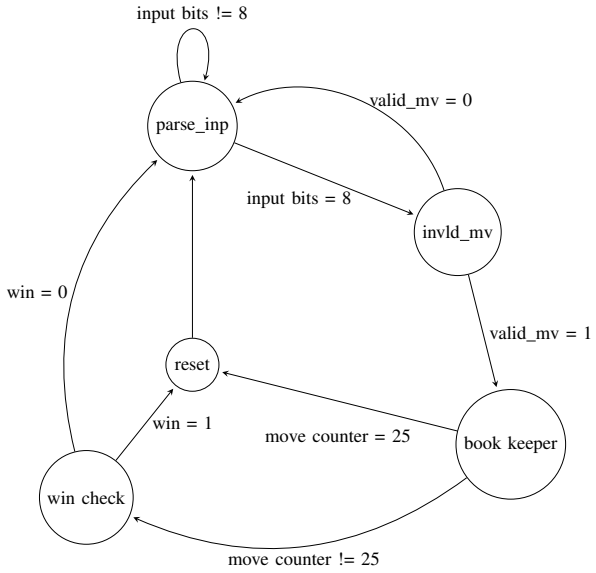

Fig. 4. State machine of the game engine

- Triangle: 01
- Circle: 10
- Square: 11

Encoding the shapes in this way has several advantages. The first and most obvious is the ability to keep track of triangle and circle turns using a single variable -say currTurn- and if a player's turn is over, the bits can simply be inverted to give the other player the ability to place their shape. The other advantages will be discussed in the relevant states in the report.

2) Input Parsing: The input parsing state has the sole role of converting the binary input into x and y coordinates. This is done through the use of two counters – one for the y and the other for the x. The counters allow for an easy inversion of the order of the input bits and for checking whether the user has provided 8 bits or not.

3) Invalid Move Checking: If the activity button is pressed after the user has provided 8 bits, the state machine moves on to invld_mv. In this state two main checks are performed:

- The x and y coordinates specified by the user are within range, i.e., the coordinates are greater than or equal to zero and less than 10. This is done using simple if statements.
- The specified coordinate is empty. This can be done by taking the OR operator of the two bits of the encoding scheme specified above. Only an empty cell will give zero, if the state is occupied the result of the operator will be one.

4) Bookkeping: The game board can be modeled as a 2-bit 100-cell wide memory block. A position is stored in the board as follows: $y \cdot 10 + x$. In this encoding scheme, the y position is stored in the tens place while the x position is stored in the ones place. Furthermore, another array called bookkeeper is used to (as the name suggests) keep track of all player movements in sequential order.

Since the bookkeeper keeps track of movement coordinates sequentially, erasing every sixth move becomes just a trivial task of accessing the bookkeeper array at the appropriate index, using the info retrieved from that array to write 11 to the board.

5) Win Checking Algorithm: A win condition can be only created after the last move that is placed on the board. That is why, instead of checking the whole board we just check the area around the last move. To do that we define several offsets: dx, dy, positive diagonal offset, and negative diagonal offset used to check rows, columns, diagonals with positive slopes, and finally diagonals with negative slopes respectively. The slopes are defined with respect to the coordinate system shown in Figure 5.

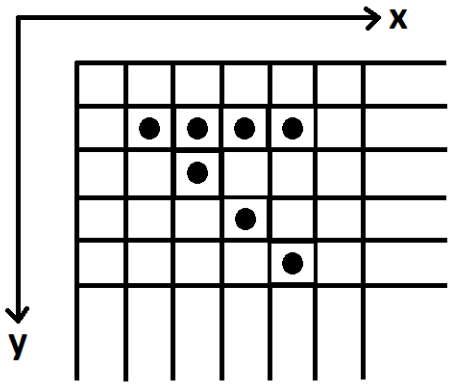

Fig. 5. Win checking algorithm

The win checking state is itself a state machine with two distinct states:

- Looping limit calculations
- Consecutive shape counting

To understand the algorithm better consider Figure 5, above.

a) Consecutive Shape Counting: To count consecutive shapes one can add the previously defined offsets to the x and y coordinates and keep track of similar shapes using a counter. Depending on the slope one can add the offsets as follows:

- Rows: $y \cdot 10 + x + dx$
- Columns: $(y + dy) \cdot 10 + x$
- Positive Diagonals: $(y + \text{diagoffset}) \cdot 10 + x + \text{diagoffset}$
- Negative Diagonals: $(y - \text{diagoffset}) \cdot 10 + x + \text{diagoffset}$

Note that the offsets can be negative and that the offsets count from the minimum possible coordinate up to the maximum possible coordinate. Thus, an implementation detail that should not be forgotten is to reset the consecutive shape counter if the sequence is broken, otherwise the

algorithm might give a false positive.

b) Row and Column Limit Calculations: The minimum possible limit for looping is -3 regardless of the slope type. However, this value cannot be hardcoded due to the edge cases. Consider what happens if the player selects the coordinate B1 as the last move as in Figure 5. Then the minimum offset that should be added to the row cannot exceed -1, otherwise, the array index bounds will be exceeded. The same applies to the maximum offset. Thus the looping limits for row checking can be defined as follows:

$$\text{dxmin} = \begin{cases} -3 & \text{if x} \geq \text{D} \\ -x & \text{, otherwise} \end{cases} \qquad \text{dxmax} = \begin{cases} 3 & \text{if x} \leq \text{G} \\ 10 - x & \text{, otherwise} \end{cases}$$

c) Positive and Negative Diagonal Limit Calculations: There is a little more subtlety involved in calculating the diagonal looping limits because both x and y coordinates should be taken into account. Again, consider input B1. In this case, the minimum offset is -1 = dxmin = dymin = posmin.

Now let's consider the input C1. In this case, the minimum diagonal looping limit is bounded by the y coordinate since it is smaller. Thus, it can be concluded that the looping limits of the diagonals depend on the looping limits of rows and columns. The same logic can be repeated for several test cases to obtain the following formulas:

- Posdiagmin = min{dxmin, dymin}
- Posdiagmax = min{dxmax, dymax}
- Negdiagmin = min{dxmax, dymin}
- Negdiagmax = min{dxmin, dymax}

As a result of win cheking algorithm, if one of the side wins the game, an indicating message is displayed at the bottom of the screen and the red line is drawn across four boxes indicating the winning placement as it can be seen in Figure 6. Also, the board is wiped after 10 seconds for a new round and the score of the winner side is increased by one.

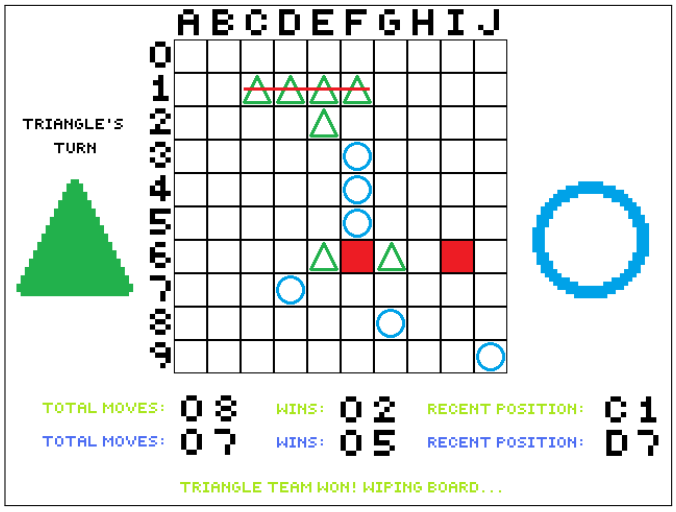

Fig. 6. Example of the win state

6) Resetting: Resetting is triggered if either a draw or win condition is created. In either case, the board game is wiped by writing 00 to all cells. The bookkeeper array is also wiped, and the appropriate counters are reset. The game state is set to parse input in preparation for a new game.

III. Conclusion

As a result, in this project, we implemented a 2D strategy game using a modular and flexible program structure so that the game is both efficient and adaptable to potential future modifications. As a result of the research we did during the project, we also had the opportunity to better understand and learn the Verilog syntax. In addition, we gained experience in debugging as a result of the problems we encountered during the project. Finally, thanks to this enjoyable and instructive project, we learned about collaboration and time management.

References

[1] "Vga signal 640 x 480 @ 60 hz industry standard timing." http://tinyvga.com/vga-timing/640x480@60Hz.

[2] terasIC, *DE1-SoC*, 2014. https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=165&No=836&PartNo=4#contents.