FPGA Maze Game

David Robert Smith

Computer Science Department  
San Jose State University  
San Jose, CA 95192  
408-924-1000

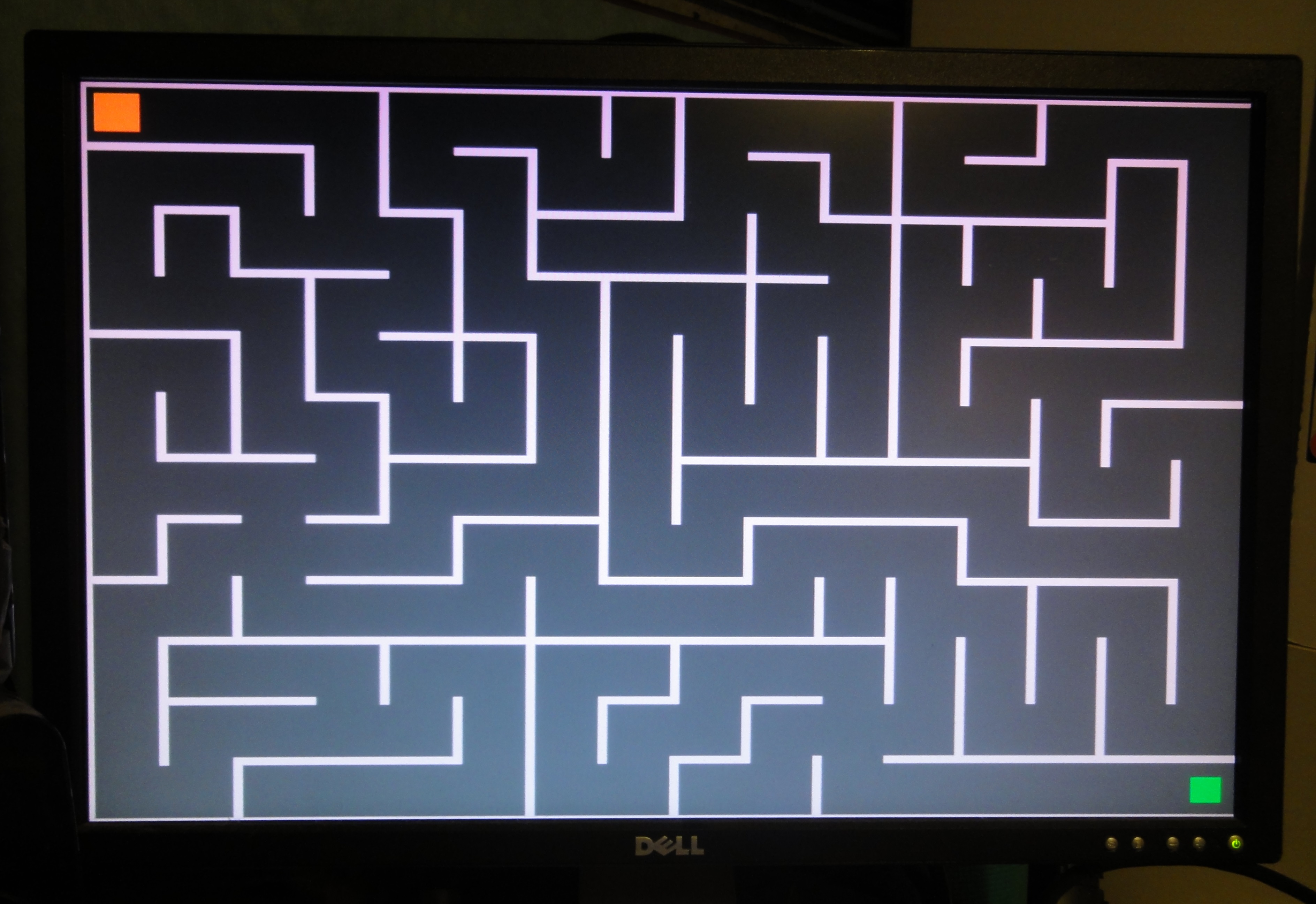
davidrobertsmith@comcast.net

**ABSTRACT**

I wrote a simple maze game for use on an FPGA. This paper summarizes how to make such a game in general and the methods I used to implement it on the Cyclone II FPGA starter board from Altera.

# INTRODUCTION

My term project is programming a maze game on an FPGA board. I’ve had experience with programming games so I come to this project with some game-making fundamentals. Where I have no experience is in programming hardware directly, so this made for an interesting challenge. I purposely picked a game that I knew I could program in about a day if I were to do it with a programming language I was familiar with such as Java or even C. Then I would have enough time to discover how to work with the hardware aspect.



The Finished Maze Game

The most interesting challenge comes from the event driven nature of VHDL and the FPGA and how using clocks can overcome this.

# MAZE GAME DESIGN

Before I get into the discussion of the hardware aspect, the FPGA board, let me start with discussing some basics of game-making, this maze game in particular.

## The Ego Character

A maze game has a character that I’ll call Ego, which the user moves around the screen. Ego is limited to where it can go by the walls that surround it. Ego starts at the beginning of the maze and is trying to reach the end.

Ego can be anything, from a complicated sprite, to a simple square which I used in my maze program. The square is represented by its width, height and color. Ego’s position on the screen is stored in two variables, typically called x and y, for the horizontal and vertical position respectively. It helps to think of the screen as a standard Cartesian plane, and x, y being where the character is placed. In any given moment, on every clock cycle, Ego is drawn on the screen at current coordinates of x and y. For ego to move right, the value of x will increase, and for ego to move left, the value of x will decrease. Similarly, to move up and down, y will decrease and increase. As Ego is constantly redrawn on the screen, with changes in the values of x and y change, animation is achieved.

## The Maze Background

The maze and the background are immobile, but they still need to be drawn every cycle. The background is represented by a matrix of 640 by 480 pixels. Each pixel is a color value. This is the same information you’d see in an uncompressed image file. For my background, I chose to have a solid dark color. The maze is drawn on top of this and I chose to have another plain light color for it.

The next main thing the computer needs to know is where Ego can’t go on the screen. Basically, it can’t go where the walls of the maze are. I used the same internal values to represent where the maze should be drawn as well as where Ego can’t go. I could’ve also represented the maze as part of the background matrix if I had made it multi-colored. Instead, I made them as separate objects.

## User Input

The user inputs instructions by pressing one of four buttons or keys, where each button reference one of the directions up, down, left, or right.

# VHDL

The program for the maze game is written in a coding language called VHDL. The program is written on a PC, compiled, and then loaded into the FPGA board through a USB connection.

## Event Driven

Common languages, like Java or C run in a sequential way. That is, one line of code runs, followed by the next, and then the one after that. Of course, there are ways to run it out of order with loops and calls to functions. Still, you can expect that a block of code will execute in order. VHDL doesn’t work that way. Any line of code can run at any time and in any order depending on events. It’s an event driven language. For example, say one line of code involves assigning x <= y+z. If z changes somewhere in the program, this line is automatically executed, reassigning x to be y plus the new value of z. Of course, hardware is event driven, so it makes sense that VHDL would work this way as well.

Understanding the event driven nature of VHDL is the key to understanding how to program the board.

# VGA

FPGA outputs to a standard VGA monitor connected through the VGA connector on the board. When making a game using VHDL and an FPGA the biggest hurdle, or at least the earliest to overcome, is getting anything to appear on the VGA screen at all. Part of the reason for that actually is the sequential nature of a VGA display.

## How VGA works

VGA is a progressive scan display, meaning that pixels are drawn horizontally one line at a time. When one line completes, the next line down is drawn. Once the bottom line has been drawn, it goes back up to the top and starts again. One iteration from top to bottom can be thought of as a frame. Just like with motion pictures, frames are drawn very quickly one after the other, as fast as 60 frames per second, and the subtle differences between each cause the illusion of motion and animation. So, with the maze game, Ego can be shown in one position in one frame, and slightly over in the next.

## The VGA Challenge

The challenge with VHDL and the FPGA board is how do we communicate with the monitor to let it know what colors to put on the screen, at what positions and when. The secret to this is using a clock.

## The Clock Pulse

A clock is always moving; it is always consistent; and fortunately, the FPGA board has an onboard clock that can be used for programs just like this one.

The clock inside the FPGA board emits pulses, that is, a bit that switches between 0 and 1 in very consistent time intervals. Every time the clock changes to one, we call that a clock pulse. These pulses can be used to drive events.

## Pulses Driving VGA

In the case of the VGA monitor, with every pulse we move to the next pixel on the screen. We want to tell it what color that pixel should be, as well as where the next pixel should be drawn afterword. For example, if we reach the end of a horizontal line we want to move down to the next line and start at the front. If we reach the last horizontal line we want to tell it to go back up to the top of the screen.

## VGA Physical Connection

The physical connection from the FPGA board to the VGA monitor includes five connectors I use in my program. Three represent red, green and blue signals, making up standard RGB and the other two hsync and vsync. These let the monitor know when it’s time to draw the next pixel.

Once we get the VHDL code set up properly to take input of clock pulses, process the imagery, and output to the screen, we can move on to refining the images we want to display to the screen.

## Drawing Images to VGA Monitor

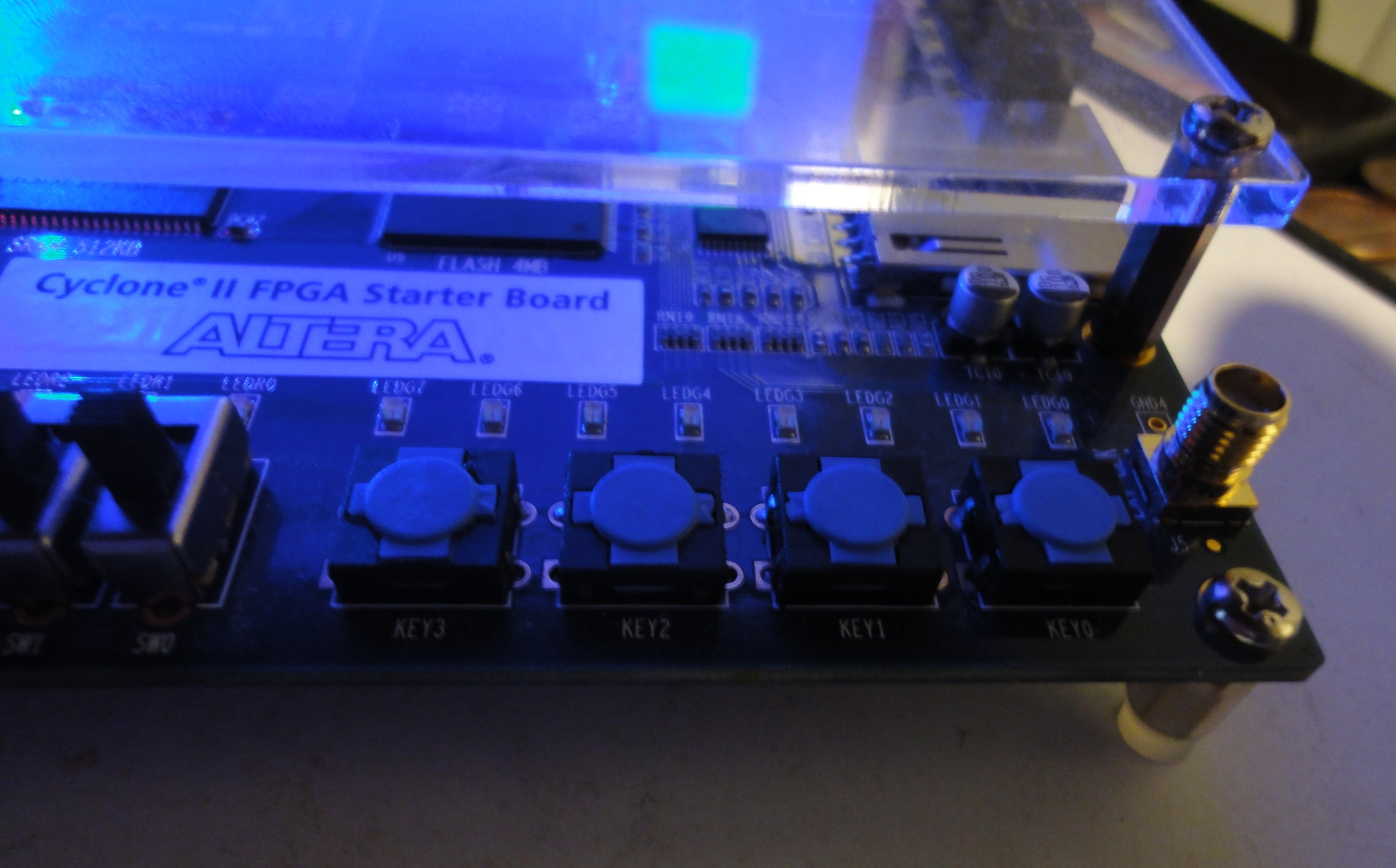
Fundamentally, drawing images on the screen means telling the monitor what color it should draw each pixel as it scans through them. So, if I want to draw an orange square starting at x = 100 and y=200 with a width and height of 50, I would tell the program to draw orange whenever it is at pixels of 100 to 150 of lines 200 to 250. I would use the same principle for the maze, background, or more complicated images.

# GRAPHICS

With the VHDL processing code all in place for communicating to the VGA monitor, the next step is creating the objects and in the case of Ego, animating it.

## Animating Ego

Programming Ego to move around the screen is fundamentally easy. Pushing and holding any of the four buttons, called KEY3, KEY2, KEY1 and KEY0, changes the respective button signal to change from ‘1’ to ‘0’. Releasing the button causes it to change back. So, if the button signal for the button I designated for up is ‘0’, the program should decrease the y value of ego’s position, to cause it to move up the screen.



The four buttons: KEY3, KEY2, KEY1 and KEY0

### Animation Pitfall

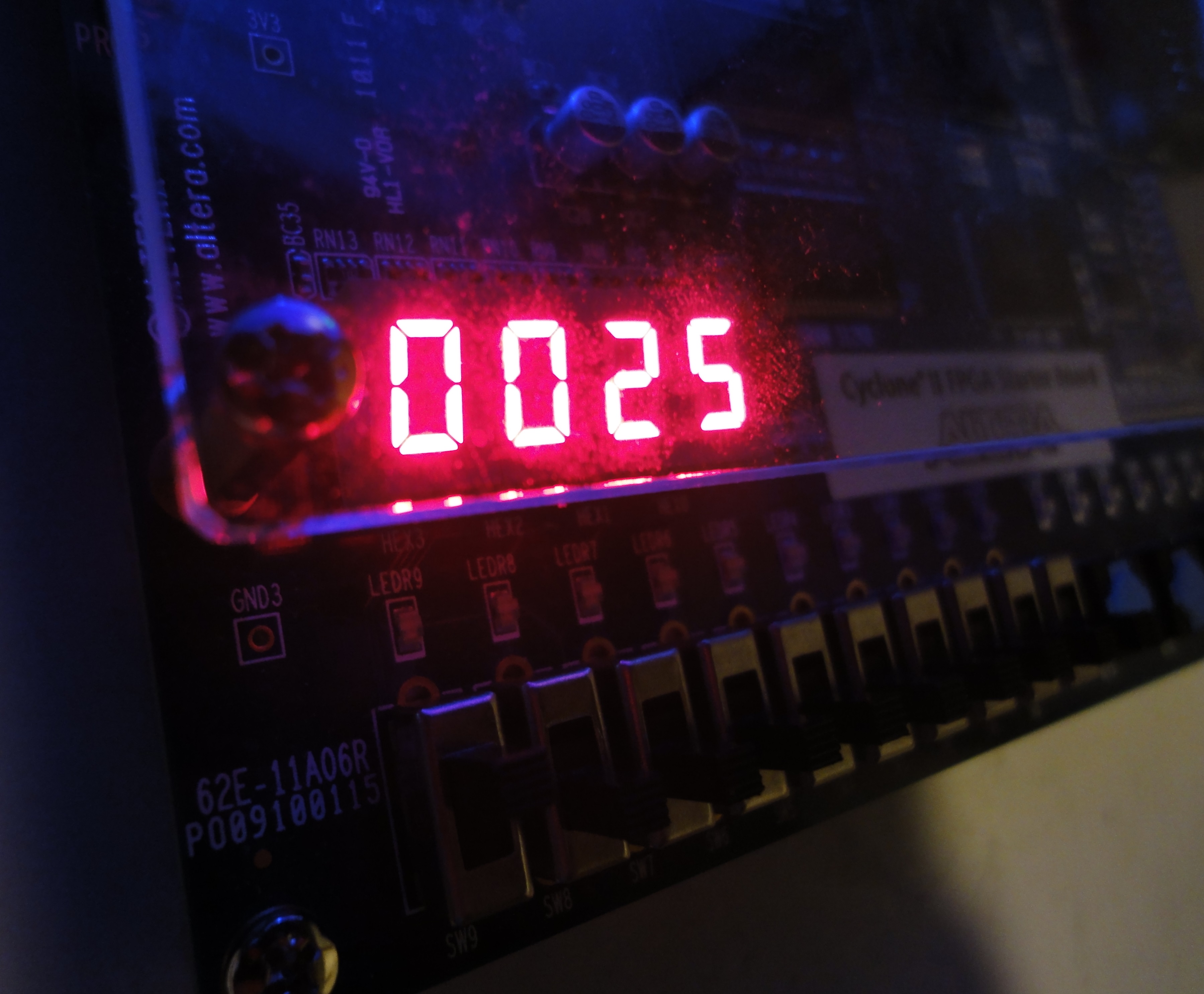
There is one important aspect to be mindful of when animating Ego or changing its position. This should not be done while the screen is being drawn. Doing so causes each frame to be choppy. Ego’s position should only be updated once a complete frame has been drawn yet before the next frame starts being drawn. This is achieved by only checking if a button has been pressed during clock pulses that are between frame draws. Similarly, other animations should only be updated at this time as well.

### Step Size

I created a signal called stepsize that determines how far ego should move with each movement. This is the value that is added to or subtracted from x and y. The bigger the value the further ego moves each step. To make the game more interesting, I added functionally to one of the switches so that stepsize is either 1 or 5 depending on which position the switch is in.

# SEVEN SEGMENT DISPLAY

While I was setting up moving the character around the screen, I found it useful for debugging to actually have the coordinates of ego being printed out as Ego moved around. However, that’s another difficulty with VHDL and the FPGA board. There is no way to just simply do a printf statement. Printing text to the VGA monitor is even more complicated than figuring out how to print a square. Sprites have to be constructed that could represent the different letters and numbers.



The Seven-Segment Display

Instead, I decided to use the FPGA’s onboard seven segment display. This in itself offered an interesting challenge. When using the seven-segment displays you don’t just send to it the number you want to appear. You actually have to say specifically whether you want each of the seven segments turned on or off. So, I had to go through each segment and figure out for each number from 0 to 9 which of the seven segments should be displayed. Then I had to take each digit of my integer I wanted displayed, and use a case statement to say if it’s “1” use these segments, if it’s “2”, use these segments, etc. It was very satisfying to see it actually work. I used one of the switches on the board to switch whether the value of x or y is displayed on the seven-segment displays.

Having the seven-segment displays show ego’s x and y coordinates made debugging the ego movement much easier.

# THE MAZE

The next piece of the puzzle was creating the maze itself.

## Maze\_type

The maze data is stored in a maze\_type which contains two matrices. One matrix represents the vertical walls or columns of the maze and the other the horizontal walls or rows. Each individual row and column is made of a series of wall segments represented by binary values.

The data in the maze\_type is hard coded to be a map of size 16 by 12 blocks. I did this because 16 by 12 divides very nicely into the 640 by 480 screen size with each block being 40 by 40 pixels. Of course, I could create more maze\_types with different sizes, but that’s probably beyond the scope of this assignment. The data maze\_type contains can be any maze within those size parameters.

## Draw Maze Procedure

A maze drawing procedure determines which segments are to be drawn using the information in the maze\_type. Then it sends to the monitor to draw a rectangle representative of that actual wall segment.

## Walls Stopping Ego

To make sure the Ego can’t move through these walls, a function called “blocked” checks the same values of the maze\_type that are used to draw it. When changing the ego’s x or y value based on user input, the code checks to see if making such a move would cause ego to be placed on or through a wall segment. If it does, the move doesn’t take place.

## The Maze in this Game

Although, I had finished the framework of the maze game, I still needed the layout of an actual maze to input as data into an instance of maze\_type. For this program, I obtained a maze layout from a website called mazegenerator.net which generates mazes based on input parameters. I chose 16 by 12 and hard-coded the generated maze into an instance of maze\_type. Again, it’s probably beyond the scope of this assignment, but if I wanted to make a more elaborate program, I could’ve written my own maze generating algorithm, so that a random maze would appear every time!

# SUMMARY/CONCLUSION

That covers the basics of the maze game and how I implemented it on the FPGA board. I gained a new respect for how it must have been for programmers who made very similar games on the Atari 2600 back in the 1980s. Although the fundamentals are the same, making a game this way is very different than the way I used to make games when I started. I almost wish that I had learned with a system like this. I think it would have given me a much greater appreciation for all the intricacies of game-making beyond just the coding.

This experience taught me a lot about how actual graphic displays work, which for all the years I’ve been using them, was very illuminating.

Finally, learning about how a clock actually works to drive events on this board, which is presumably how it works with my desktop machine as well, adds a great new perspective for looking at computing.

# REFERENCES

[1] JGB Service. Maze Generator, mazegenerator.net, 2014.