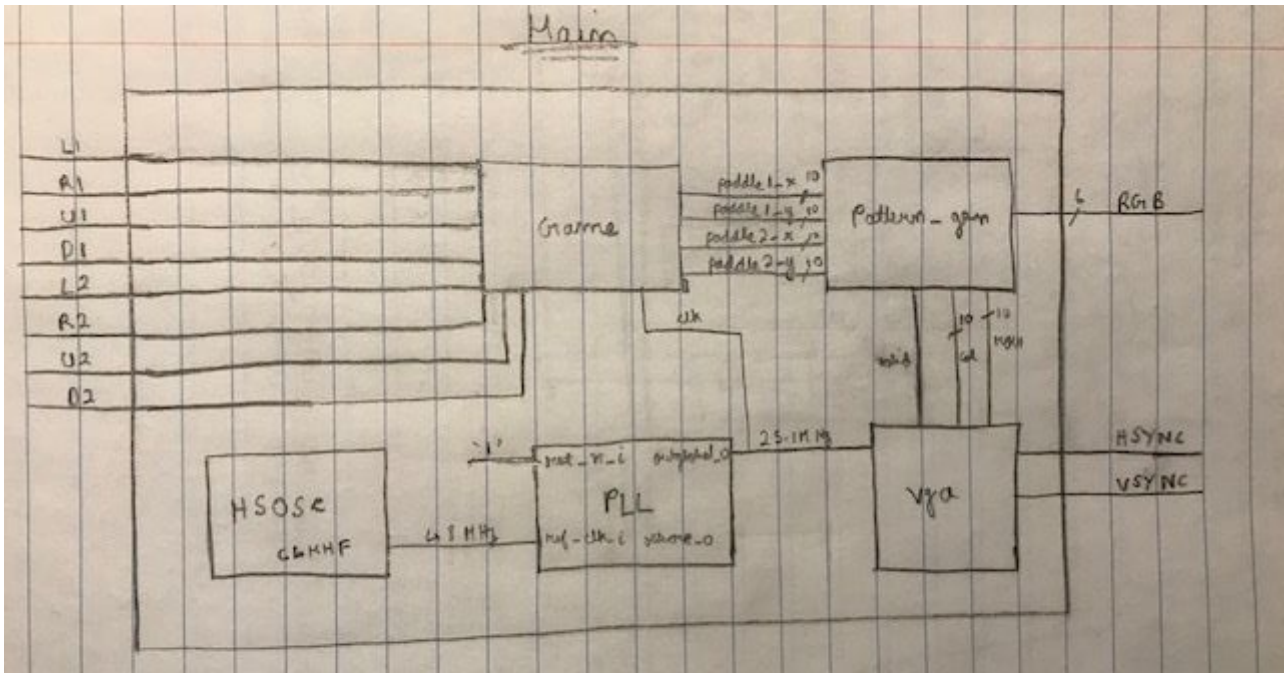
Final Project: aMAZEing Race

Group Members: Athokshay Ashok, Ryan Megathlin, Syed Taswar Mahbub, Kostas Tsiampouris

Overview:

Our project was to build a two player maze game that could be controlled using buttons built directly on the breadboard. There were four buttons for each player designating the up, down, left and right for their paddle. The players must navigate their way through the maze and reach the end to win. If either player touches a boundary or a wall inside the maze, they will be reset to their starting position and will have to restart while their opponent plays on. There is no way around the maze but through it.

There were 4 main pieces to the design: the clock adjustment modules, the game module, the pattern generation module and the VGA module. The clock creation modules used the FPGA's internal clock and the Pll module created by Radiant to convert the clock speed from 48MHz to 25.125 MHz needed for output to the VGA display. The VGA module used the clock taken from pll to create a row and column count that gets passed on to pattern_gen, along with the valid bit, HSYNC, and VSYNC outputs for the display. The game module handled the next position of the two paddles and set the restrictions on the movement of the paddles inside the maze. The pattern_gen module took each block's x and y coordinates and materialized the rest of the block based off of that. It also included the maze, which was hard-coded in.



Note on switching projects, and our work on Brick-Breaker:

Our original plan was to build a brick-breaker game where the idea was to have a paddle move along the bottom of the screen with a ball bouncing around and breaking bricks at the top of the screen. If the ball touches the bottom of the screen, the player loses. The player had to control two switches which moved the paddle left and right and tried to keep the ball from crossing the bottom boundary, which would mean game over. We started of by writing a C++ implementation of the game and then tried reinterpreting the algorithm in VHDL. We were very careful not to use any feature that would prove to be too costly (e.g. divide by non-power-of-2). This allowed us to go through all the logic and simplify it so that we had a definite plan going into VHDL. The C++ version looked like this:

One of our biggest initial concerns was how to allow the ball to have any velocity without the use of floating points. We decided to divide each pixel in 32 sub-pixels, so that a ball with speed of ~`1 pixel/clock` could have a large number of possible directions.

Having the ball bounce in the walls and the paddle was simple. We would simply compare the coordinates of the ball with the ones of the walls/paddle, and if the ball was further out, we would reverse one of its velocities (vx for horizontal hits, etc.), and in the case of the bottom, we would make sure the horizontal position was between the sides of the paddle.

Managing the blocks was a bit more challenging. We didn't want to store the coordinates of all blocks, as this would waste precious memory. We simply stored a bit-vector, for which each

value corresponded to whether a brick was alive or dead. To find if the ball would collide with a brick, we would simply compare the balls coordinates with the rectangle where bricks could exist, and if it was inside we would hash the ball's position, by this formula :
`collision? = bricks[BRICKS_PER_ROW * BALL_Y / PIXELS_PER_BRICK + BALL_X / PIXELS_PER_BRICK]`

If a hit was identified we would continue by erasing the brick and changing the velocity of the ball. Whether the ball would bounce horizontally or vertically was determined on whether `abs(BALL_X-BRICK_X) < abs(BALL_Y-BRICK_Y)`.

We were very careful to set `BRICKS_PER_ROW` and `PIXELS_PER_BRICK` to a power of 2, so we could do shifting instead of division/multiplication. We also, made sure to put margins on the left and right, to avoid any potential unsigned underflows.

Unfortunately, implementing this in VHDL was more challenging than we expected:

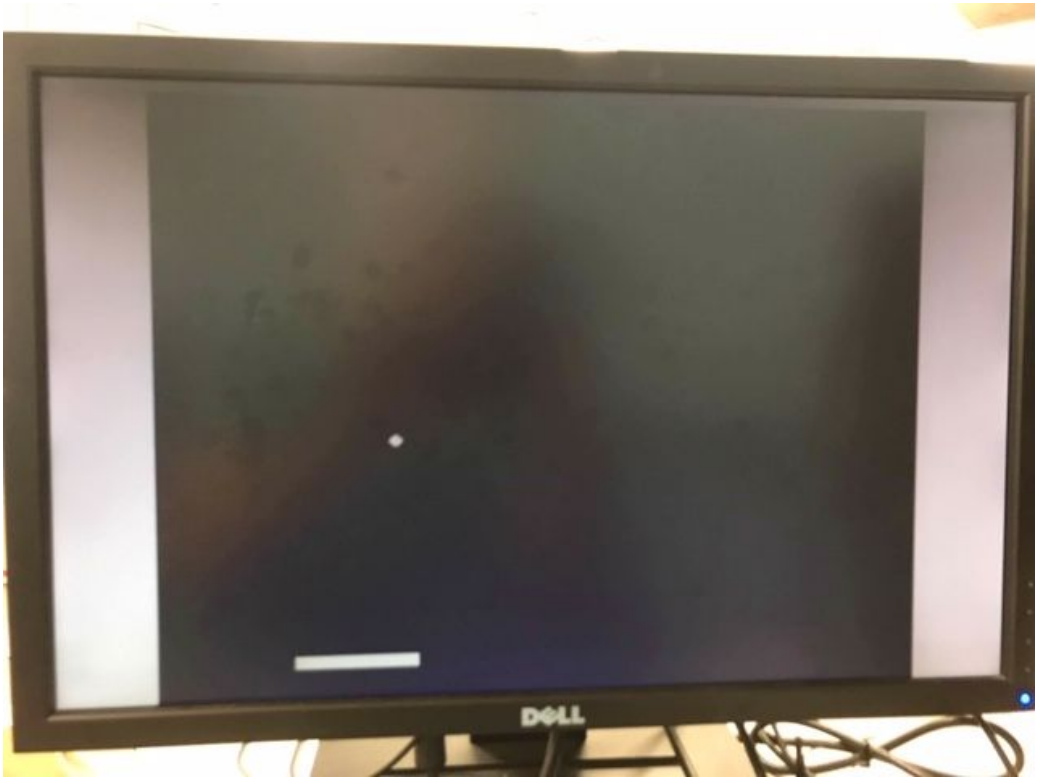
We first began with getting pattern gen to display a paddle and a ball, but ran into issues with the VGA module, which turned out to be slightly incorrect from what we previously used to display a pattern in lab 5. Most of our initial designing of the brick-breaker game was spent trying to get a proper design for the VGA module. Whenever we tried to change the pattern gen module to get a paddle and a ball to display on the screen, the VGA module would crash and not display anything. After figuring out the issue with VGA, which turned out to be an error with updating HSYNC and VSYNC properly, we worked on getting the paddle to move left and right.

After a few iterations of testing this and running into issues such as the paddle starting and moving off screen, we got the paddle to move from side to side within the bounds of the screen. We then displayed a ball and got it to move across the screen, but it kept appearing out of bounds and getting stuck there (as its velocity was being continuously reversed). We tried everything to make the ball start on screen, from initializing the signals to transporting the ball there with user input, but nothing seemed to work. Another issue we had was undefined behavior in comparison statements in VHDL. The ball would bounce successfully on the right and bottom of the screen, but not the respective lower bounds of left and top. Comparing with a non-zero value seemed to always give true. We concluded that was because of undefined behavior caused elsewhere in our code.

Despite spending hours trying to debug this and talking to several TAs about this issue, we could not resolve the bugs we were having. We realized the issues we were having were becoming too difficult to fix and decided to salvage the parts that were working, namely the paddles that were moving left and right and the working VGA module. After coming up with the

idea for the maze, we converted the paddles to move in both x and y directions and moved forward from there.

At the point where we switched projects, our brick breaker game had a paddle moving from side to side and a ball that started at a specific spot, moved to the bottom left, bounced off the bottom, bounced off the left wall and kept floating around while going through walls and looping back around. If the ball hits the paddle, it would glitch and start moving from left to right along the level of the paddle.



Technical Description for the maze game:

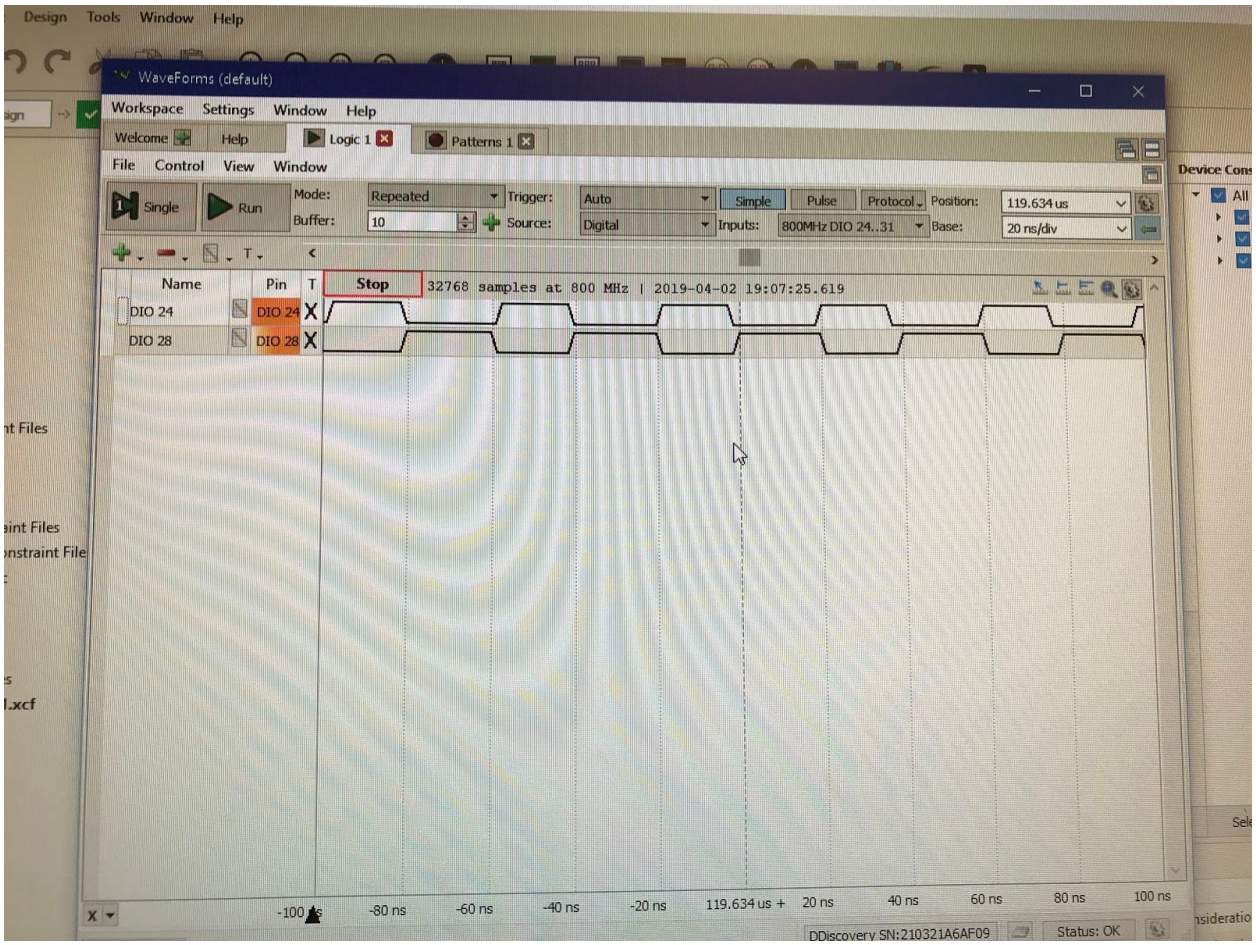
HSOSC:

The HSOSC component is a built in clock in the FPGA that generates a rising edge at a specified frequency (which we chose to be 48 MHz). This component was included in the Game module.

PLL:

The PLL component is another built in module that takes in a clock frequency and returns another clock value of a different frequency. In this case, the component took the 48 MHz frequency and returned a 25.125 MHz frequency clock value, and this same clock was used for every other component in the top level code.

Proof of working pll clock: Top line (DIO24) is a Waveforms signal set to 25MHz. DIO28 is the created 25.125MHz signal which is slightly longer than the 25MHz signal.



VGA:

The VGA module takes as input the clock output from pll and outputs the row, column, valid bit, HSYNC, and VSYNC values. This module essentially gives us the information needed to get the VGA driver to function properly. The 640 by 480 pixel display has several regions: sync, back porch, visible area, and front porch. As shown by the diagrams below, we want HSYNC and VSYNC to be high if the current pixel position is in the back porch, visible area, or front porch region. The VGA module uses the clock to iterate through the pixels of the screen, keeping a row and column count. This row and column count is fed to the pattern_gen module in order to determine which pixels should be what colors. A 'valid' bit is also determined that is high when the pixel count is in both the horizontal and vertical 'visible areas'. This bit was fed into the pattern_gen as well and assists with determining when pixels should be high or not so as to not confuse the VGA display.

Pattern_gen:

The pattern gen module was where the maze and paddles were processed to be displayed. We got rid of the two margins we were initially displaying for the brick breaker game as well as the ball, but kept the paddle. To display the paddles, we took in as input the x and y locations of both the paddles, as well as the clock. Given the locations, we printed a box that was 16 by 11 pixels relative to the x and y positions. The paddles had different colors to avoid confusion. The code to print one of the paddles is shown here:

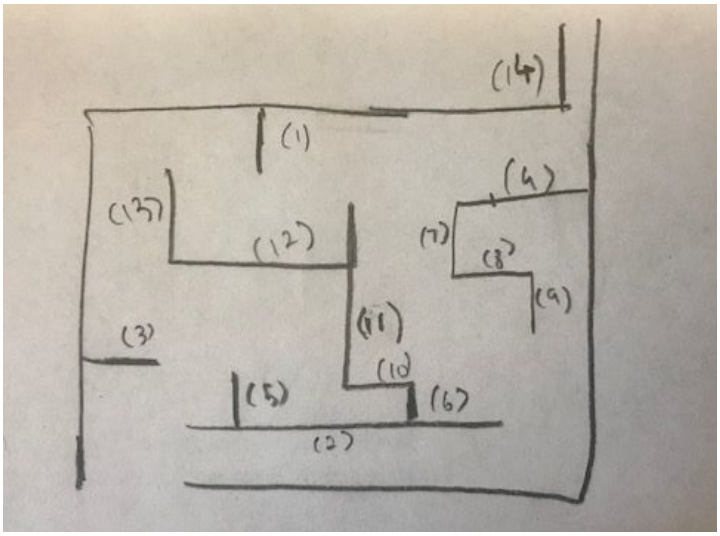
```
col < (paddlex1 + 15) and col > paddlex1 - 1 and row < paddley1 + 10 and row > paddley1 - 1
```

To display the maze, we first had to create a maze. After looking through various maze patterns online, we came up with a design that was simple yet challenging. The maze design had two openings, one for the start and another for the finish.

Each wall of the maze had to be individually hardcoded. That is, we had to tell RGB to print a white wall if the rows and columns were between certain values. Therefore, the conditions to print the walls were all included in one long when-else statement. We started with the boundaries of the maze, printing a box with an opening at the bottom left corner and the top right corner for the start and finish lines. We then numbered each of the walls in the interior of the maze and began printing them one by one to make sure that they were printing in the right positions. We also had to make up the sizes of the walls ourselves, so the first few interior walls were crucial since they determined the scale of our maze. The code for the first wall is shown here:

```
(row > 150 and row < 200 and col > 300 and col < 310)--1
```

In this way, we went through and got each wall to print and we modified the sizes of the walls along the way to make the game intentionally harder. Close attention was paid to ensure that the walls were not too close such that the paddles could not move through them anymore.



Game:

The game module was used to control the movement of the paddles based on the input and to determine the bounds of the paddles. This module takes in as input the left, right, up, and down controls for both the paddles, as well as the clock, and outputs the x and y locations of both the paddles, which pattern gen uses to display the paddles.

To move the paddle, we added one pixel in the direction corresponding to the input. We also checked if the paddle was within the bounds of the maze rather than within the screen when we were doing the brick-breaker game. The code to move the first paddle is shown here:

```
paddlex1 <= (paddlex1 + 1) when (L1 = '1' and R1 = '0') and paddlex1 < 14000 else
            (paddlex1 - 1) when (L1 = '0' and R1 = '1') and paddlex1 > 6500  else
            paddlex1;
paddley1 <= (paddley1 + 1) when (U1 = '1' and D1 = '0') and paddley1 < 12000 else
            (paddley1 - 1) when (U1 = '0' and D1 = '1') and paddley1 > 5000  else
            paddley1;
```

This not only kept the paddles within bounds, but also made it impossible for players to cheat by going around the maze to the finish line. At this point, we had the paddles that could move freely within the maze regardless of the walls, even though they displayed, so the next step was to implement this.

We set the paddles to an initial start position, and wanted the paddles to return to this location each time the paddles hit a wall. This was done by taking the wall locations that we had created in pattern gen and telling the paddles to return to the start position if the x and y positions of either of the paddles was in any of the wall regions. The code for setting the initial paddle positions and resetting the position if the first paddle hits wall 1 is shown here

```
if once = '0' then -- This happens once
  once <= '1';
  paddlex1     <= 15d"4800";
  paddley1     <= 15d"5000";
  paddlex2     <= 15d"5000";
  paddley2     <= 15d"5200";
end if;

if paddlex1 = 14000 or paddlex1 = 6500 or paddley1 = 5000 or
  (paddlex1 > 9600 and paddlex1 < 9920 and paddley1 < 6400 and paddley1 > 4800) --1
```

In this way, we were able to get a fully functioning maze that would not let the player move freely inside it.

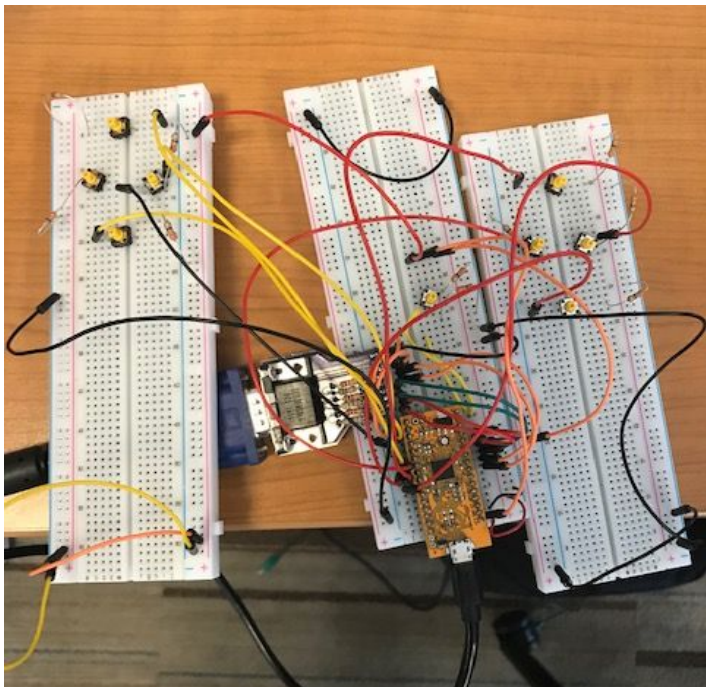
Main:

The main module is the top level VHDL module that consists of the HSOSC, Pll, VGA, game, and pattern_gen components. The inputs and outputs of each of these components are properly mapped in this module using temporary signals to connect different components wherever needed.

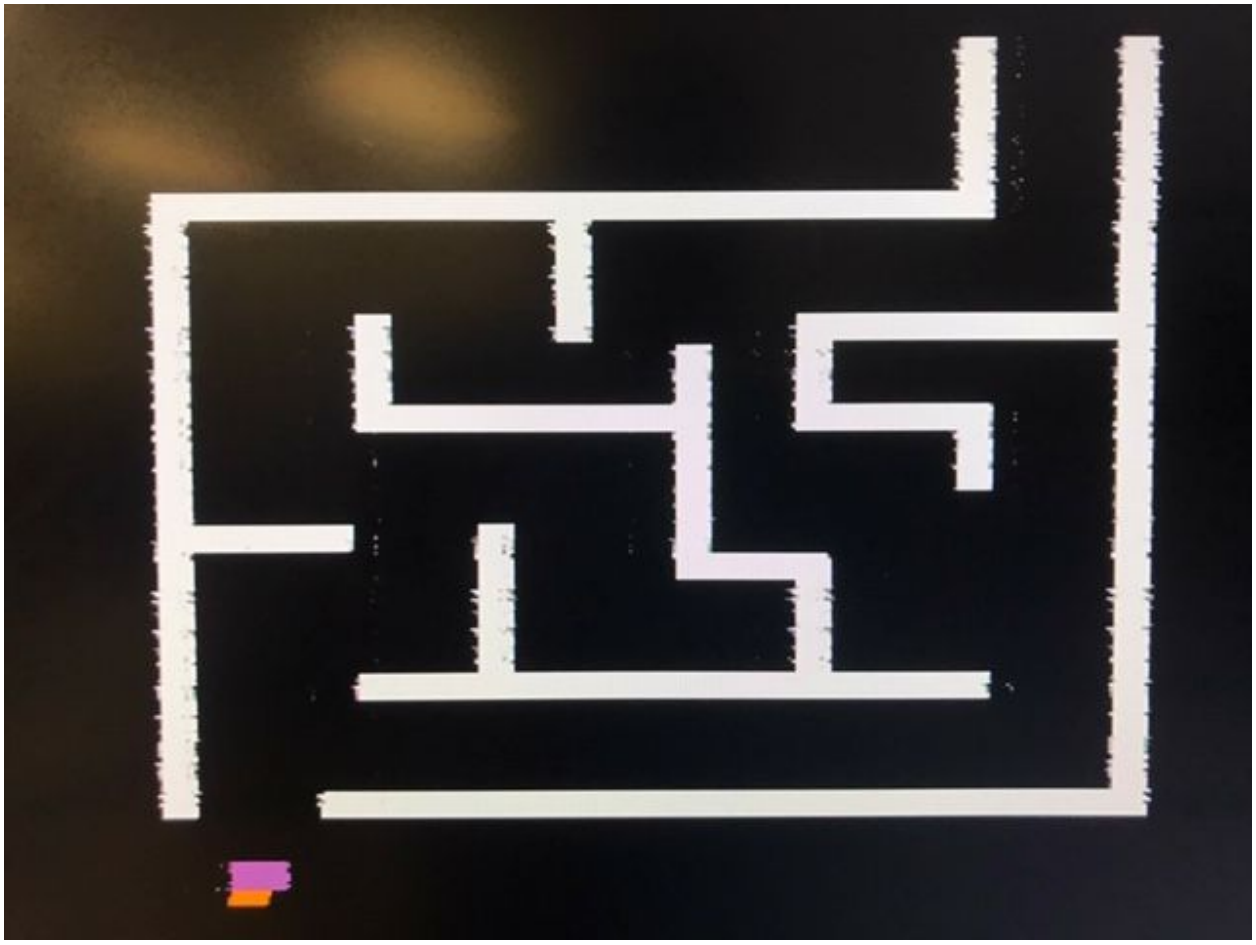
Circuit Design:

To build the circuit, the following materials were used:
3 breadboards, 8 switches, one FPGA board, one VGA driver with its own 6 resistors , 8 resistors to ground the switches, a VGA converter cable, and wires.

The idea was to have one board that handled the input and output using the FPGA, and to have two other boards that acted as the controllers for the two players. Each of the “controllers” had four switches for up, down, left, and right, which were organized in a comprehensive way. In the picture shown below, the middle board with the FPGA received the input from the switches on the left and right breadboards, and the output, which consisted of the 6 bit RGB values, the HSYNC, and the VSYNC bits, were fed to the VGA driver in the middle breadboard. One end of the VGA converter cable was then connected to the driver and the other was connected to the back of a computer screen, which was set to the VGA input mode.



Results:



Our final output was a functioning maze game with the two paddles, one which is horizontal and one vertical. This was done to make it strategic for the players when they pick a path to travel in the maze. Based on which paddle the player picked, they would find a certain path more favorable, especially when it comes to avoiding walls. The slight issue was that since we made our paddles relative to their x and y positions and did not design them around the center, the paddles could move slightly over the walls without getting reset. We would have liked to change this and make the movements more precise given more time.
Our VHDL code can be found on the github page here:
https://github.com/ktsiam/Brick-Breaker/tree/master/maze

Team Reflection:

Overall, our team was happy with our end result, considering the issues we ran into early on with the brick breaker game. We all collaborated together and used each other's strengths to break up the project into smaller parts and tackle these pieces. The biggest issue we had was coordinating times to work together since our schedules did not quite line up.

If we could start over with the brick breaker game, we would have gone to TAs earlier on to figure out how to get the ball to bounce off the walls. In addition, a simple ball-bouncing game was made in C++ before making it in VHDL. While modeling is often helpful, this seemingly put us in a mentality of trying to write "software" for the game as opposed to thinking about what hardware we wanted to design. It led us to issues particularly because of the different ways C++ and VHDL handle their respective codes. We could get the ball to bounce off the walls in C++ but when we mirrored that in VHDL, it wouldn't work. Ultimately, this might have led to our major speed-bump that forced us to abandon making the brick-breaker game.

If we were to redo the maze game, we would have either tried to display a message to show the victor or added a score counter to our game so that the players can keep track of who is winning, since the game keeps resetting the players to the start position if they get to the finish line. By keeping score, the game can be more competitive and fun. Secondly, we would increase the difficulty of the game each time the player gets to the finish line and starts back at the entrance. This could be done by increasing the speed of the paddle each time. The biggest, but most time-taking improvement, would be to add multiple mazes. We cannot tell the hardware to generate maze walls in random spots across the screen because we will have no guarantee of how challenging this will make the mazes. The output could end up being single walls placed across the screen with no real maze pattern. However, from the feedback and ideas we received at the presentation, we thought about hard-coding multiple maze patterns and randomly picking one of the mazes to display each time the game is played.