ECE385

Spring 2019

Final Project Final Report

Two Player Local Agar.io: BALL EATERS

Austin Lee (sal3)

Zuling Chen (zulingc2)

Tuesday 3PM ABE

Patrick Wang, Gene Shiue

Introduction

For our final project, we decided to make a game heavily inspired by popular online game Agar.io. The game is two-player, and the basic premise is to grow in size by eating non-player characters until one player is large enough in size to eat the other. The tools we used were built off of work we did in Lab 8; we used the keyboard to control the two different player sprites, the game graphics were displayed using a VGA controller connected to an external display, and the FPGA board buttons are used to reset or start the game. Our own additions include NPC spawns, overlap collision detection, scorekeeping, title screen, and end screens. The game begins in a start screen where the level is displayed on the screen with no player balls moving. As soon as the ENTER key is pressed on the keyboard, the title is removed from the screen and players are then able to move their respective balls. When either player ball hits a smaller NPC ball, that smaller ball is erased from the screen and the size is added on to that player's ball. The two players will compete to eat balls and increase in size. Once the bigger player eats the smaller player, the game is over and the corresponding player's color is displayed on screen as the winner.

Written description

The basis of our project built upon lab 8's use of the keyboard interface. By recording the keystrokes and their respective hex values, we then move either player ball in the direction specified by the keyboard. Player 1 uses the WASD keys and player 2 uses the arrow keys. When the game begins/reset is hit, our FSM goes to the start state which we defined as the two player balls not moving and not responding to any keyboard input. At this state, we also have the title of the game displayed on the screen. We converted a PNG image to an array of binary bits with dark colors of the title corresponding to the value of 1 and lighter colors to the value of 0. When the VGA controller scans through the screen and reaches the area where the title is, it then also reads through the array of binary bits and depending on the value, output a different colored pixel onto the screen. Hitting ENTER will begin the game from the start screen. For the two player balls, we had two copies of lab 8's ball.sv file that responded to different keypresses. The positions of the two balls were also recorded so that we could handle collisions in the colormapper module. The way that we handled collisions was we would compare both ball's sizes and if the distance between the X and Y coordinates of the balls were within the radius of the larger balls size, a collision would be detected (this was changed from a previous design where we would use the center of the balls as collision detection points). Once a collision was detected, we would have the color mapper module *not* draw that ball anymore and added that ball's size to the eater ball. Once either player wins, the FSM goes into a win state and displays the color of the winning ball on the screen. Hitting ENTER again will perform a soft-reset and the game starts over while score is preserved. Using KEY0 will perform a hard reset and return the scores and game to all default values.

Module descriptions and connections

Color_mapper
Inputs: reset, clk, is_ball, is_ball2, softreset, showtitle, [9:0] wballx, wbally, bballx, bbally, wballs, bballs, DrawX, DrawY, [0:11][0:88] titletext, blackwon, whitewon
Outputs: [9:0] wballplus, bballplus, [7:0] VGA_R, VGA_G, VGA_B, [1:0] gameover
Description: Assigns color of the pixel depending on what the corresponding pixel is (a player ball, a NPC ball, background or title screen, gameover screen, etc.), detects collisions based on ball overlaps by size, and also creates NPCs.
Function: This is used as a way to displaying the player balls and NPCs onto the screen. Data is passed into the colormapper and checks whether collisions occur through pixel overlap of two entities. When these collisions occur, we update the screen and get rid of any eaten NPCs, add the eaten size to the respective player ball, or go to the respective winning screen whenever a player wins.

Ball
Inputs: Clk, Reset, frame_clk, softreset, [7:0] keycode, [9:0] DrawX, DrawY, wballplus, Ball_X, Ball_Y
Outputs: wballx, wbally, wballs, is_ball
Description: Contains information and instructions regarding player 1's movement.
Function: Updates player 1's movement based on a keypress and handles edge cases. Data is also passed out from this module so that the ball's position may be used in colormapper to handle collisions with NPCs and the other player ball. Data is received from the colormapper to increase the player size depending on any collisions (eating) that occured.

Ball2
Inputs: Clk, Reset, frame_clk, softreset, [7:0] keycode, [9:0] DrawX, DrawY, wballplus, Ball_X, Ball_Y
Outputs: wballx, wbally, wballs, is_ball2
Description: Contains information and instructions regarding player 2's movement.
Function: Updates player 2's movement based on a keypress and handles edge cases. Data is also passed out from this module so that the ball's position may be used in colormapper to handle collisions with NPCs and the other player ball. Data is received from the colormapper to increase the player size depending on any collisions (eating) that occured.

FSM
Inputs: Clk, reset, [7:0] keycode, [1:0] gameover
Outputs: showtitle, Ball_X_step, Ball_Y_Step, blackwon, whitewon, softreset, add
Description: Finite State machine that handles score keeping, starting screen, gameplay, and winning screens.

Function: Depending on the game state, this module outputs signals to the colormapper that lets the color mapper know whether to display a victory screen, a starting screen, or display gameplay. Once the FSM receives signal from the color mapper than one player has won, it moves onto a victory screen and waits for an ENTER key to be pressed to restart gameplay. A hard reset was also included to get back to the title page.

Scorekeeper

Inputs: Clk, Reset, blackwon, whitewon, add

Outputs: [7:0] whitepoints, blackpoints

Description: Adds points to respective players' scores.

Function: When the FSM is in a WIN state the scorekeeper receives an add signal which increases the point total by 1.

Lab_8 (top level)

Inputs: CLOCK_50, [3:0] KEY, OTG_INT

Inout: wire [15:0] OTG_DATA, [31:0] DRAM_DQ

Outputs: [6:0] HEX0, HEX1, HEX4, HEX5, HEX6, HEX7, [7:0] VGA-R, VGA-G, VGA-B, VGA_clk, VGA_SYNC_N, VYA_BLANK_N, VGA_VS, VGA_HS, [1:0] OTG_ADDR, OTG_CS_N, OTG_RD_N, OTG_WR_N, OTG_RST_N, [12:0] DRAM_ADDR, [1:0] DRAM_BA, [3:0] DRAM_DQM, DRAM_RAS_N, DRAM_CAS_N, DRAM_CKE, DRAM_WE_N, DRAM_CS_N, DRAM_CLK

Description: top-level file used to instantiate all modules, connect all ports and allow different components to communicate.

Function: This served as a connection between software and hardware halves of the FPGA board compilation and passes on signals from module to module.

Title

Inputs: None

Outputs: [0:11]titletext

Description: This was used to store an array of binary bits representing our title text.

Function: Whenever the FSM goes into a hard reset and goes back to the title screen, we wanted to display a title text onto the screen. We converted a phrase we typed out in word ("Ball Eaters") into binary representation using 1s to represent black spots and 0s to represent white spots.

The Hexdrivers, VGA controller and HPI I/O interface was given code from lab 8 that we did not modify.

Block Diagrams

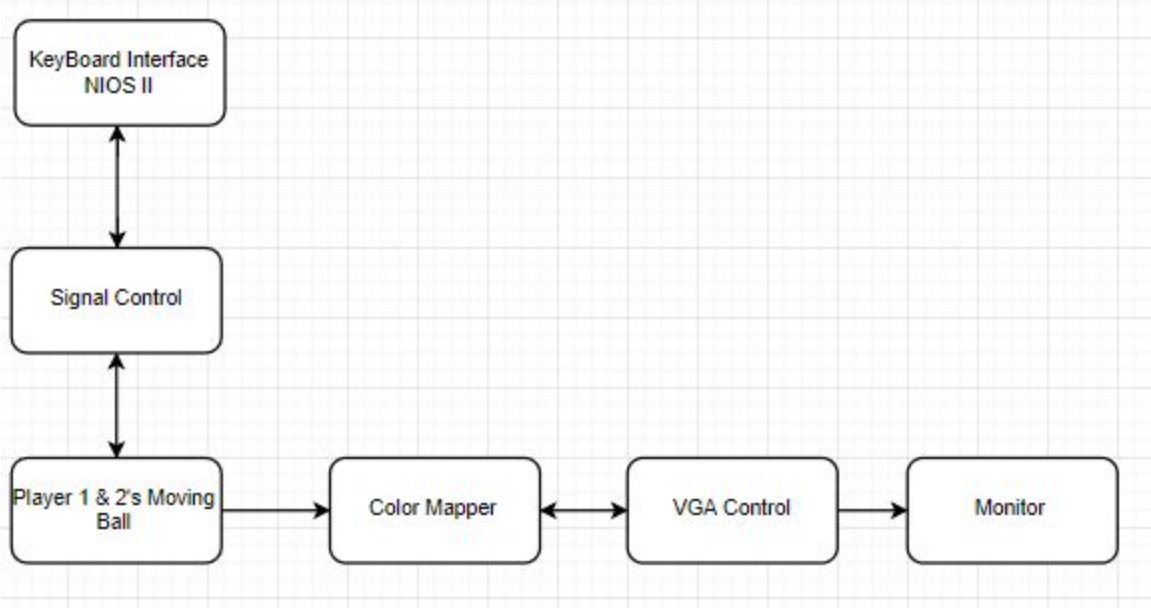


Figure 1: Basic high-level diagram

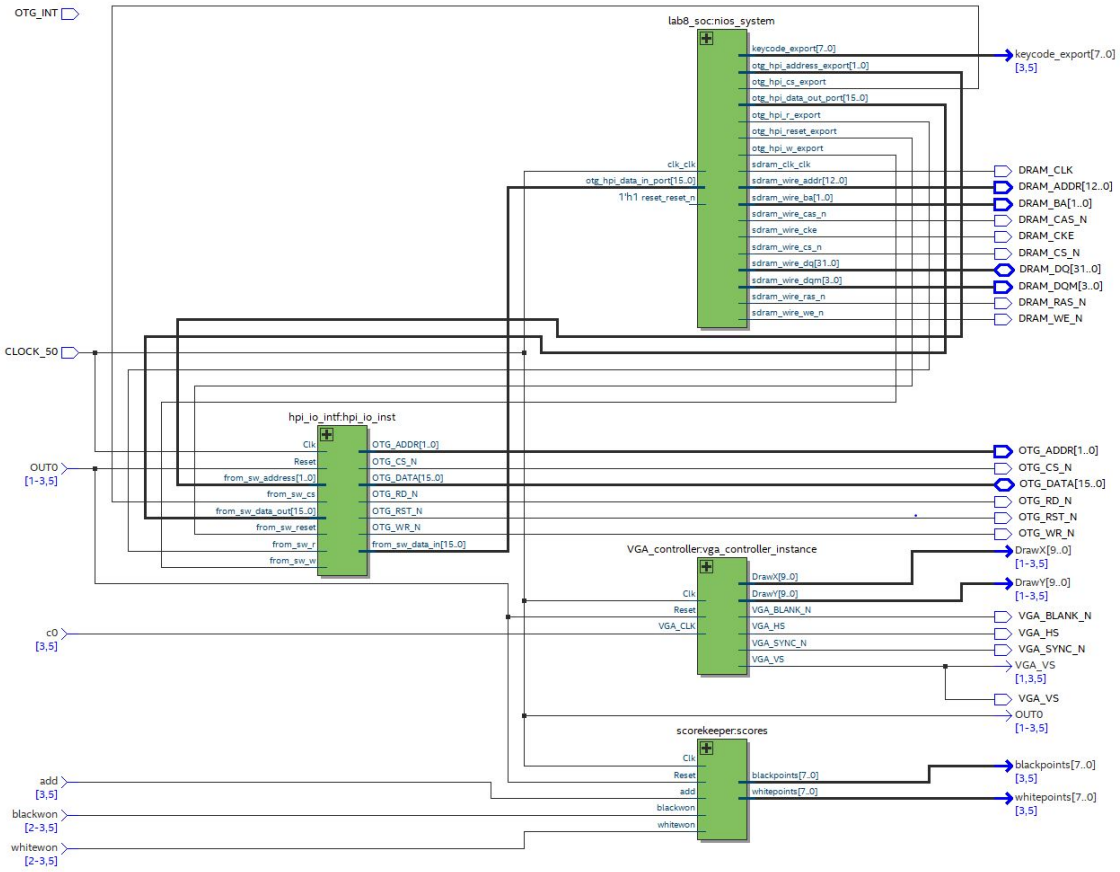


Figure 2: Top-level (part 1)

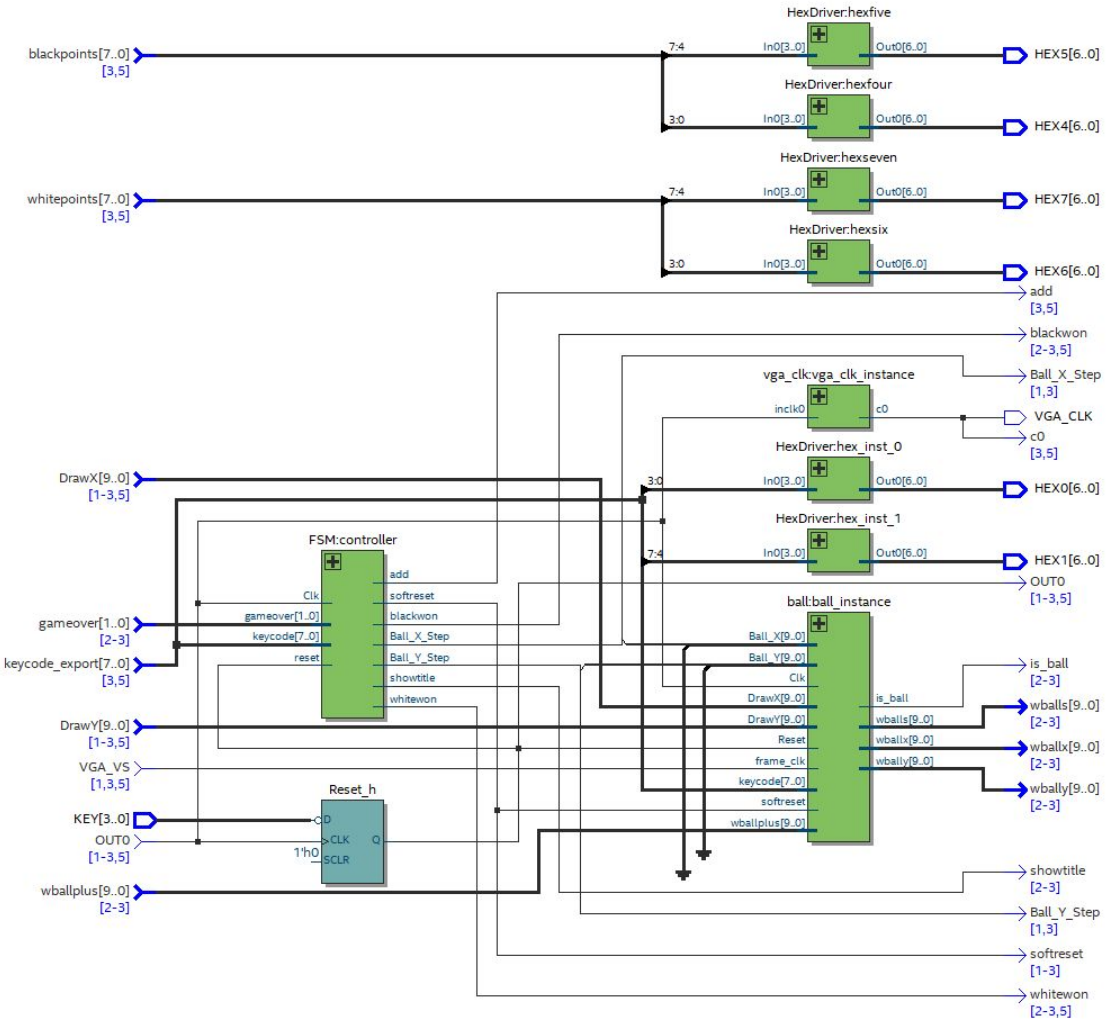


Figure 3: Top-level (part 2)

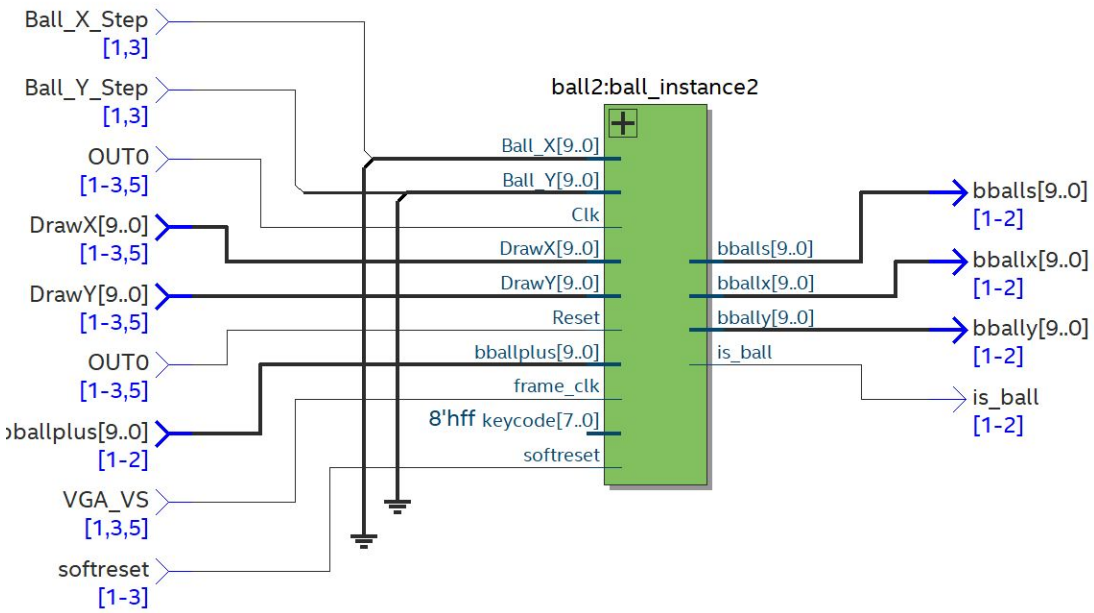


Figure 4: Ball.sv

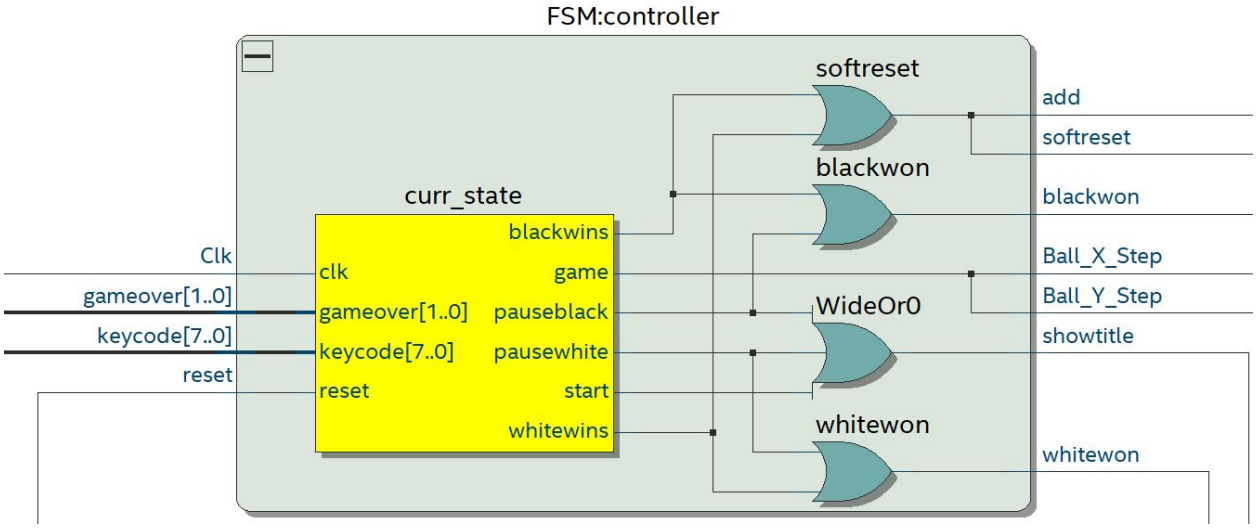


Figure 5: FSM

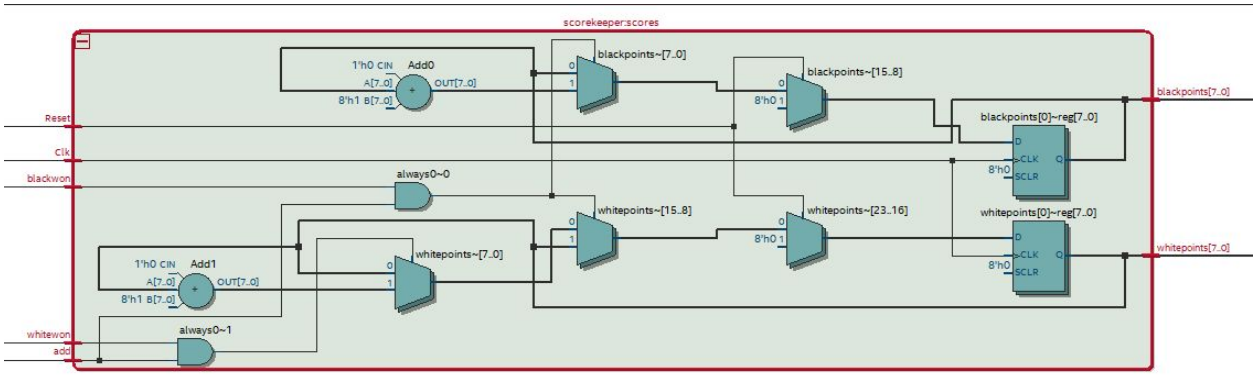


Figure 6: Scorekeeper

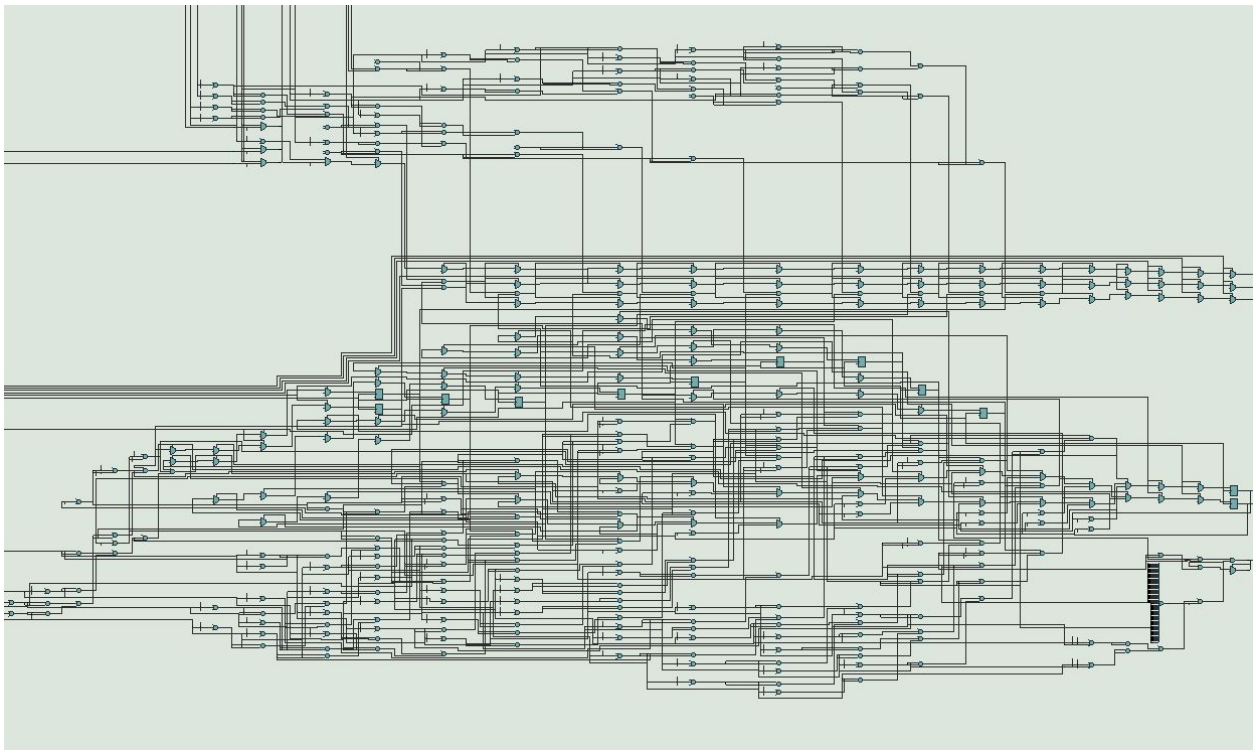
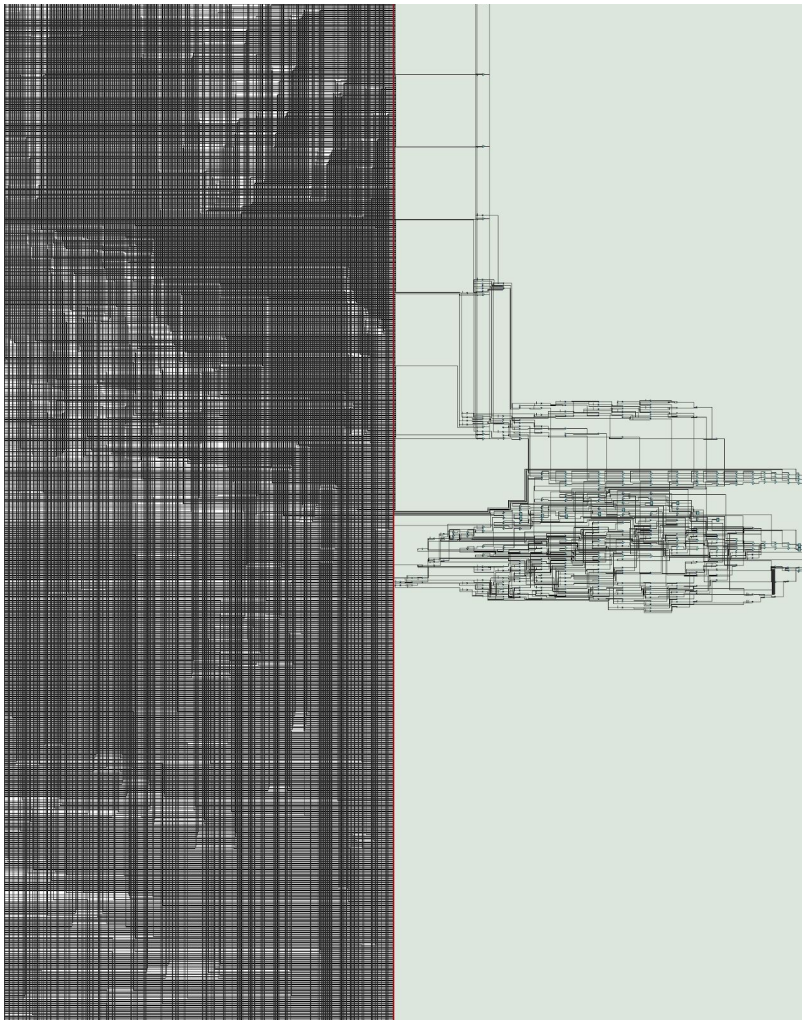
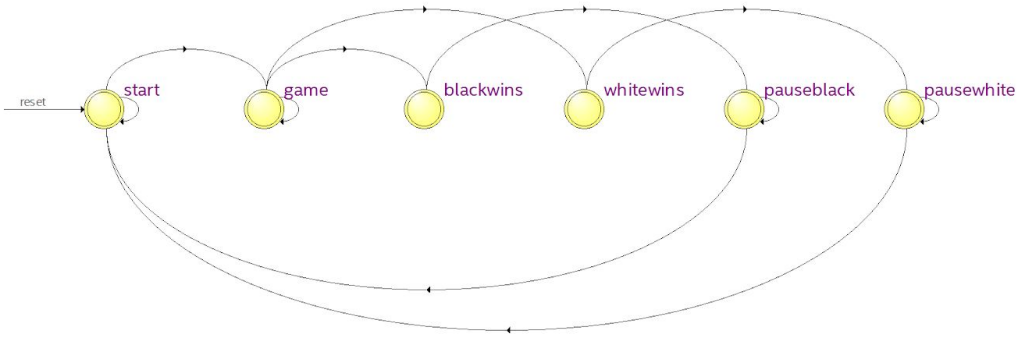


Figure 7: color_mapper (partial)



The rest of Color_mapper has been purposely omitted from block diagrams because of the complexity of connecting a 12 by 88 array from title to color_mapper (see above).

State Machine



Power Measurements

LUT	5,832
DSP	325
Memory(BRAM)	55,296
Flip-Flop	2,191
Frequency	144.26 MHz
Static Power	105.47 mW
Dynamic Power	0.87 mW
Total Power	188.32 mW

Conclusion

Overall, we believe that we should get between 5 and 6 difficulty points for this experiment. Our game was displayed on the monitor using a VGA controller and we had a working demo with a few improvements that were added into the code afterwards (we changed collision to rely on either players' instead of the NPC's radius so that the collision detection would be based off of the edge overlap). We were able to get the USB interface working to control both players and was able to get overlap collisions to work. Alongside entity collisions, we kept screen edge collisions so that neither player ball would go out of bounds. We had basic graphics using spheres to represent players and NPCs, but we were able to dynamically change the size of each player in response to a collision with a smaller ball. Through the use of an FSM, we added game states and display states for a title screen and a victory screen for both players. For the title screen, we had a text title to display onto the monitor and this was done through storing a black and white image of the text into an array of binary bits. These binary bits were used by the color mapper to draw and fully display the text. The victory screen was a monochrome screen displaying the color of the winner ball. We were able to implement continuous gameplay through a soft reset (using the ENTER key) and kept track of score on the hex displays. This project was a nice wrap up to ECE385. It was satisfying getting a game of agar.io to work on the FPGA board. It felt great having creative liberties in the final project and to be able to use information that built upon previous labs.