(1) Introduction

Our project was a 2 player tennis game on the FPGA. This involves 2 people simultaneously putting in keyboard inputs and playing tennis, where, based on their position, speed, and angle, the ball would deflect to the other side accordingly with varying X, Y, and Z positions. We used the MicroBlaze as our SoC, multiple ROM Files, the AXI protocol, the MAX3421E chip, and internal logic. We handled all of our communication from the keyboard inputs to the user display. Our final project implemented the two players with a score for each displayed on the screen. We also incorporated movement animations for the sprites, along with a start screen and restart functionality. Each player is controlled using the WASD and arrow keys, with the G and J keys being used as inputs for a swing. For this project, we used the Urbana board, which incorporated a Spartan 7 FPGA.

(2) Written Description of Final Project System

In this section, we will go in-depth into the overall workflow of the system. The game starts in the reset mode, where both players start in their positions with the ball on the side of 1 player.

The game is a slightly simplified version of tennis that incorporates the following rules: Every time a player scores the count goes up by 1, with a max score of 9 for each player; If the ball bounces 2x in bounds, the player who hit the ball wins the point; If the ball bounces 1x in bounds and then goes out, the player who hit the ball wins the point; Finally, if the ball bounces 0x in bounds and then goes out, the player who hit the ball loses the point. This does not account for the following cases that would occur in real tennis: The player hits the ball and it bounces on their side of the court causing them to lose the point; The players serve the ball and it must land in the opposite court serve box; The ball is out of reach from the player due to the height difference in the player and the ball. These cases are not accounted for in the consideration of the overall user experience. Accounting for these cases would include additional keyboard inputs, complicating user functionality and overall game design, as it is a 2D-based graphics system.

When a player presses their swing key (G or J) and the ball is in the hitbox, it registers as a hit. As a result, the ball X and Y speeds are randomly computed and changed to go in the opposite direction. The Z speed is computed such that the ball follows a certain height path. Like this, players can continuously swing and play the game. When the ball bounces (reaches a Z height of less than 25), it switches Z speed and loses energy, but keeps its X and Y computations the same. When any of the score rules are activated, the play resets, and the ball goes to the winning player's side to start the next point.

On a high level of the overall block diagram, multiple keyboard inputs are taken in and read through an SPI protocol using the USB host controller (MAX3421E). We then use AXI internally for the SoC Microblaze to communicate with the rest of the system in Vivado and the various blocks. We then write our internal logic, which will be detailed below.

(3) Block Diagram

An overall block diagram detailing how our MAX3421E chip was integrated can be seen below.

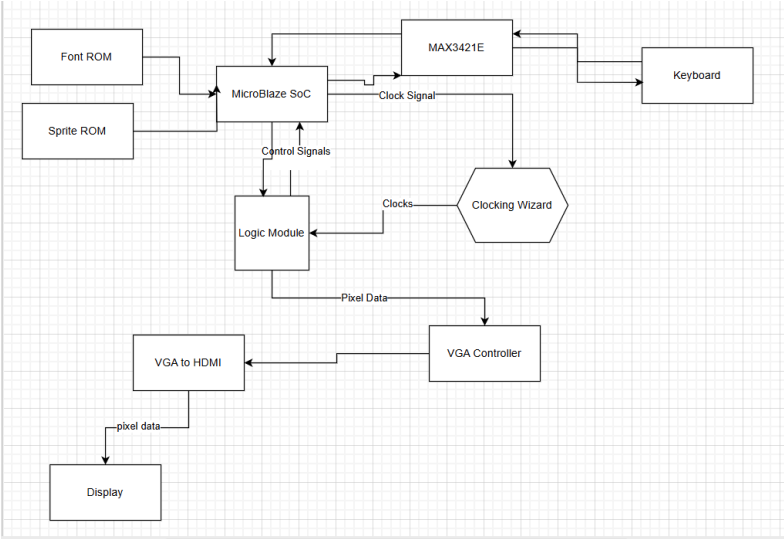

Figure 1. MAX3421E Chip Integration Block Diagram

Below, we can see our Vivado Block design and the various components we integrated to make the Microblaze System.

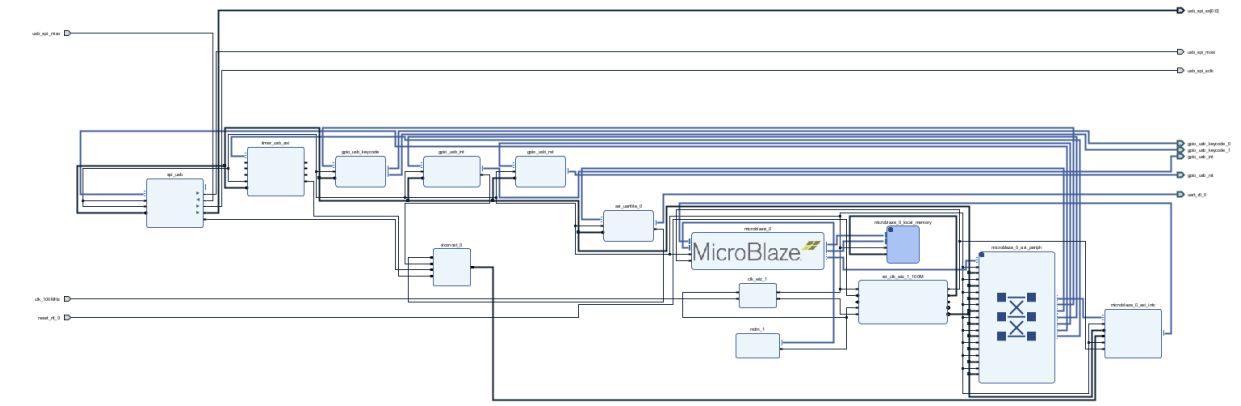

Figure 2. Overall SoC Block Design (View 1)

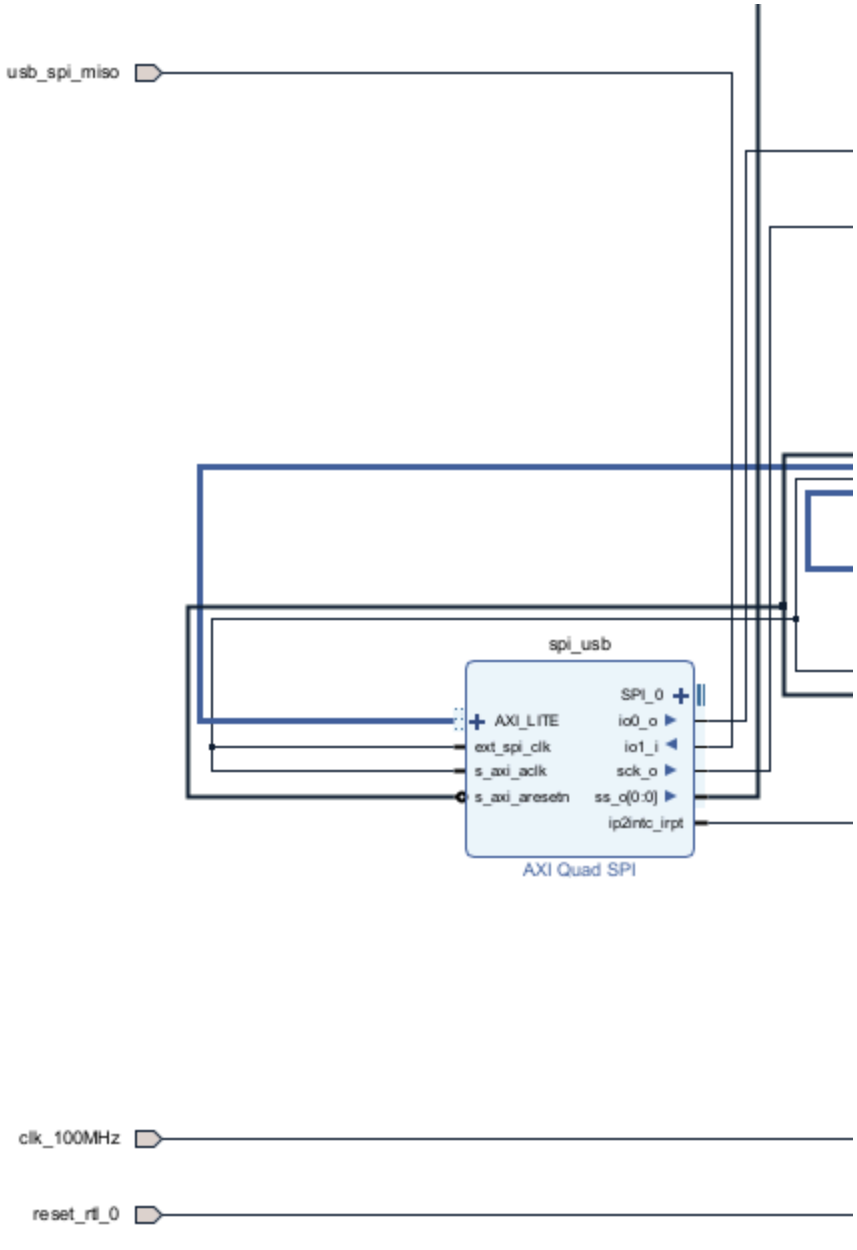

Figure 3. Overall SoC Block Design (View 2)

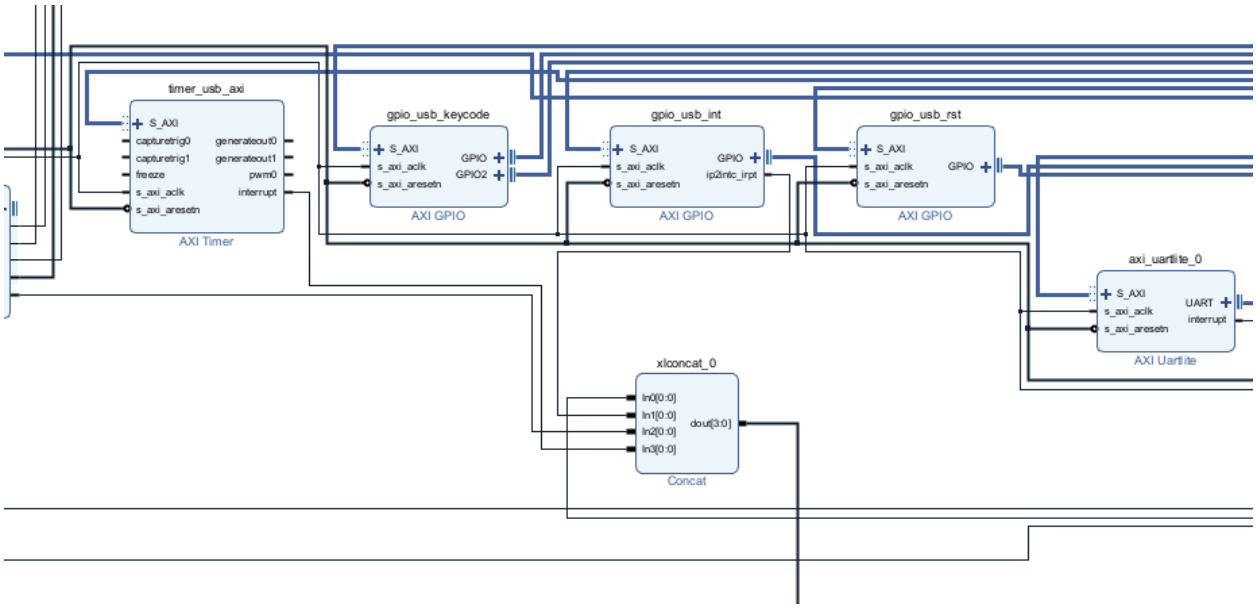

Figure 4. Overall SoC Block Design (View 3)

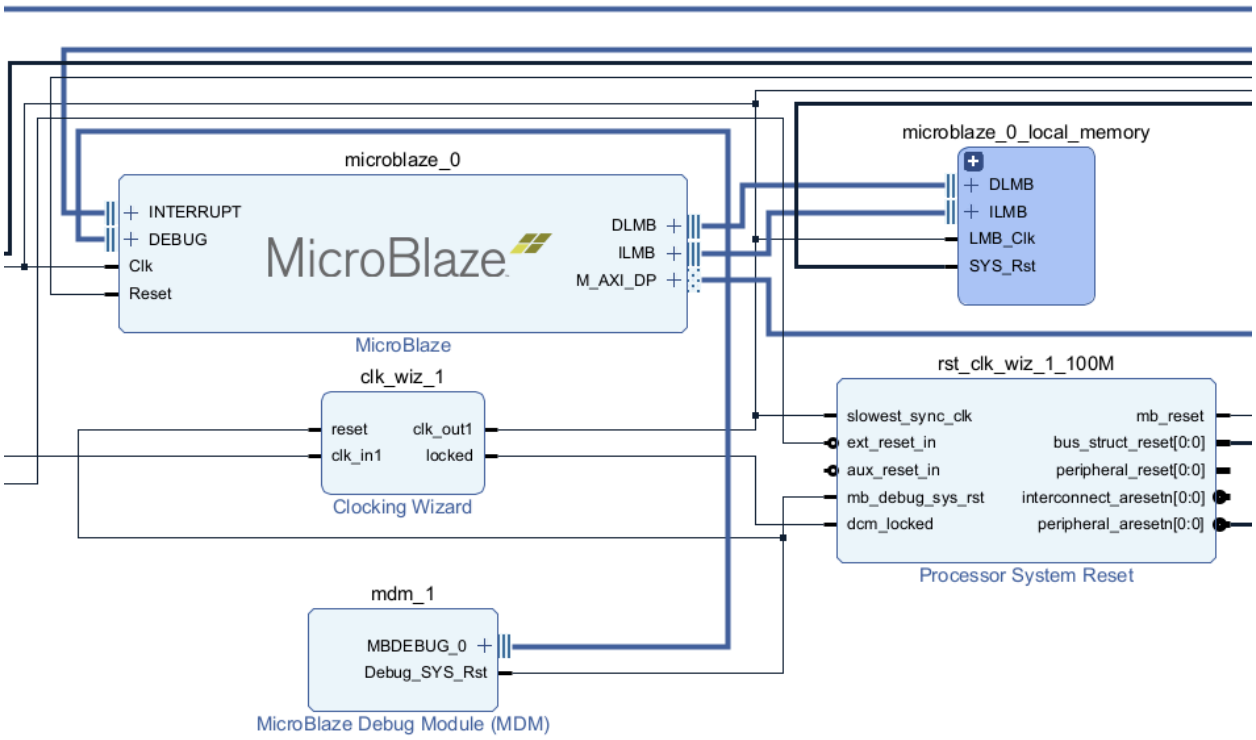

Figure 5. Overall SoC Block Design (View 4)

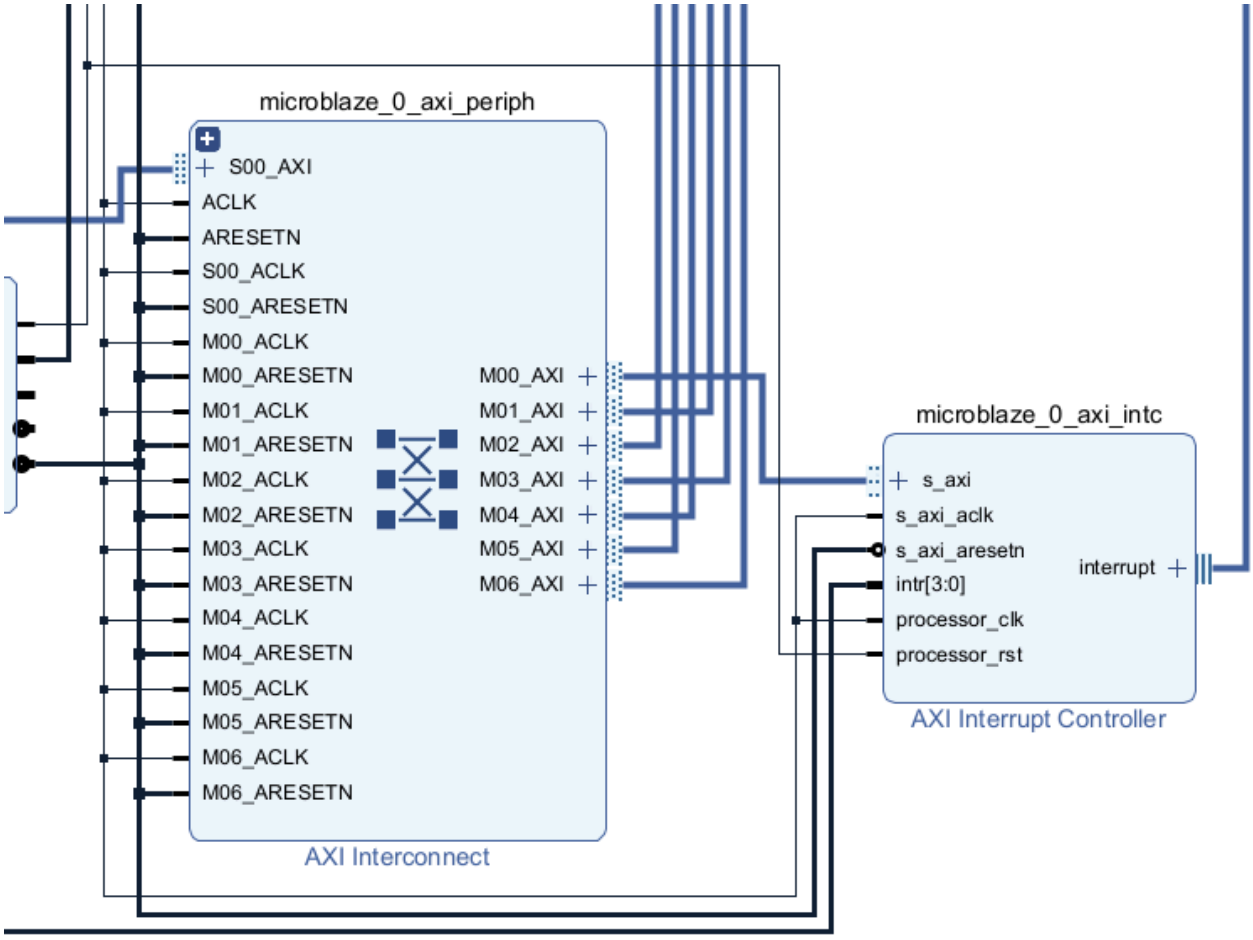

Figure 6. Overall SoC Block Design (View 5)

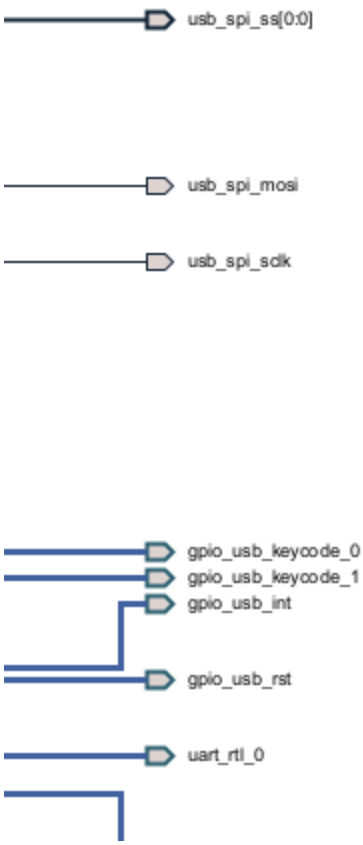

Figure 7. Overall SoC Block Design (View 6)

Finally, below here you can see the module organizations and how we integrated them to make our internal logic for player movement, ball movement, and scoring. We will go more in-depth into each module in the section below.

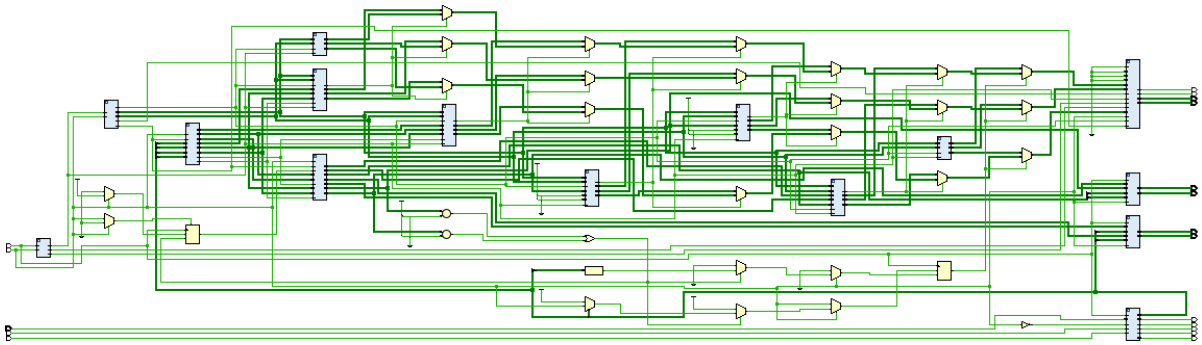

Figure 8. Overall RTL Design (View 1)

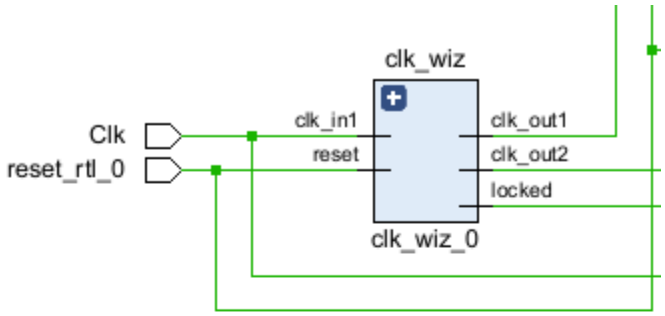

Figure 9. Overall RTL Design (View 2)

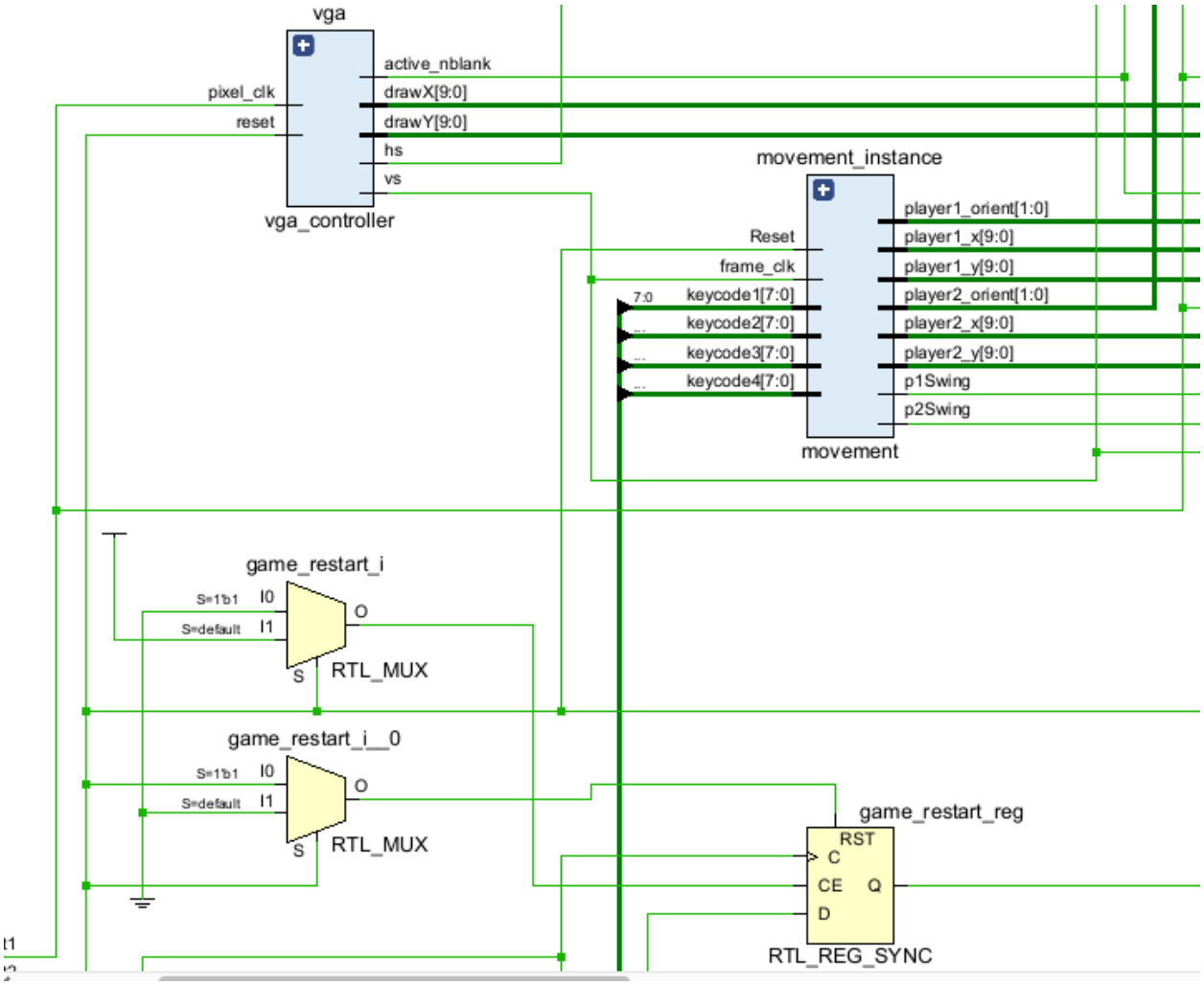

Figure 10. Overall RTL Design (View 3)

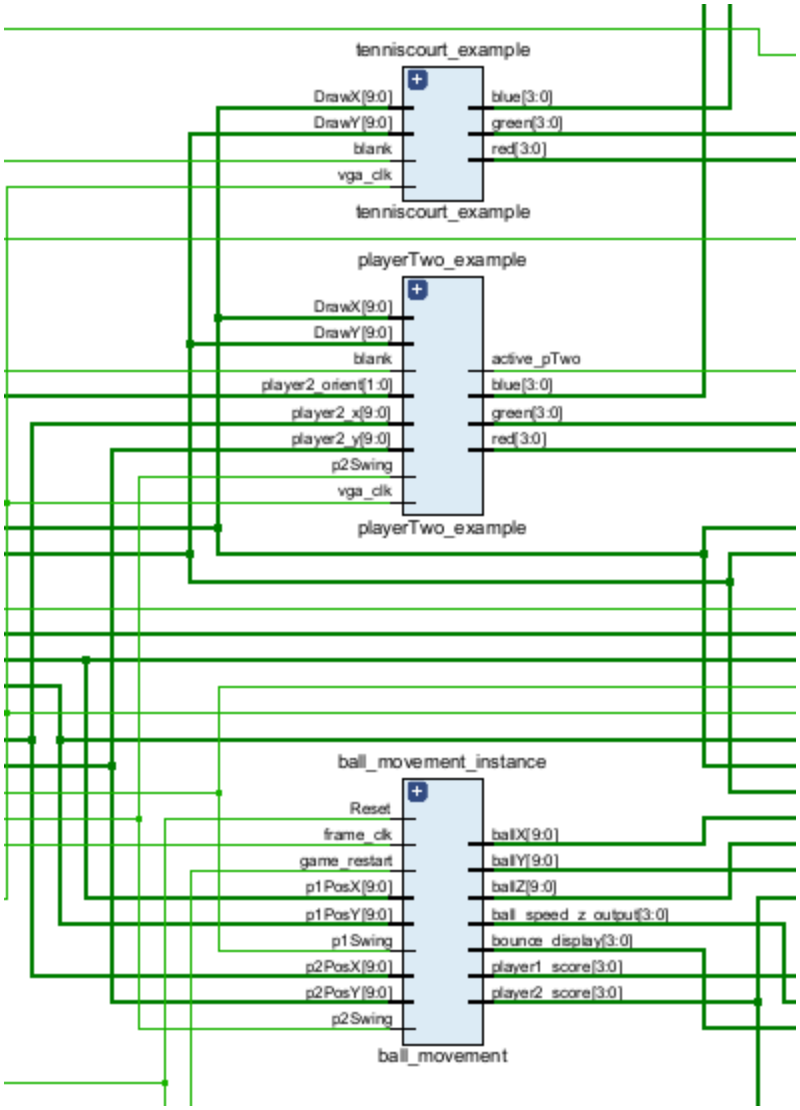

Figure 11. Overall RTL Design (View 4)

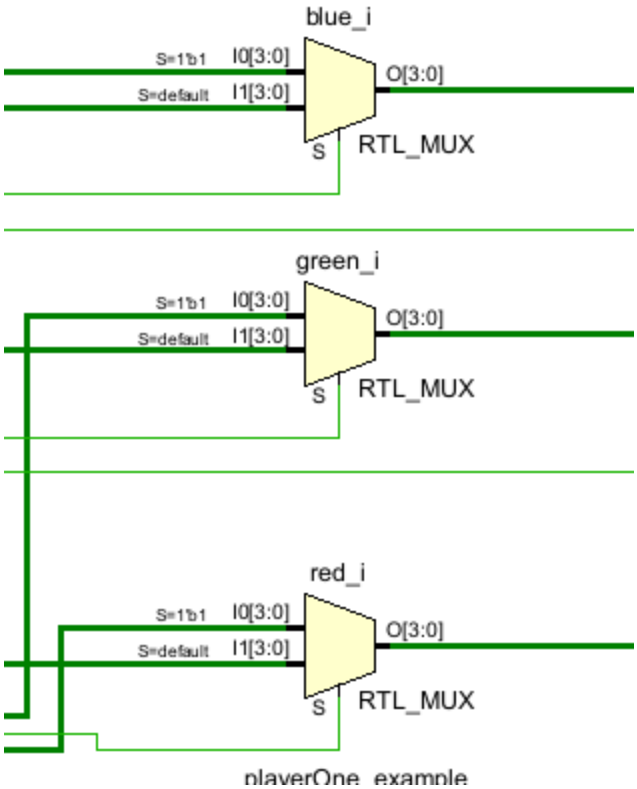

Figure 12. Overall RTL Design (View 5)

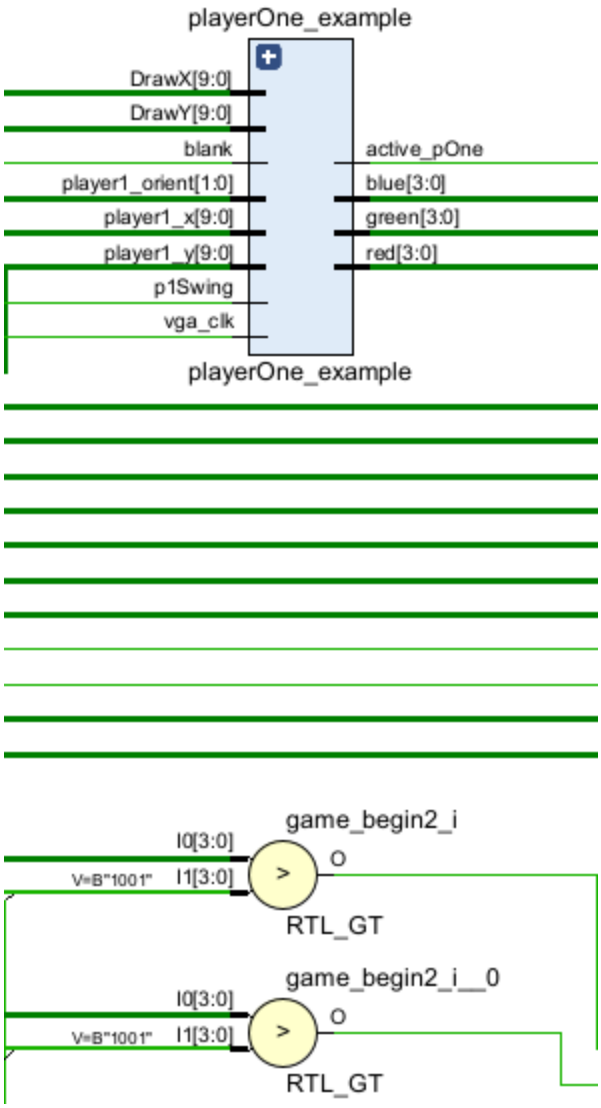

Figure 13. Overall RTL Design (View 6)

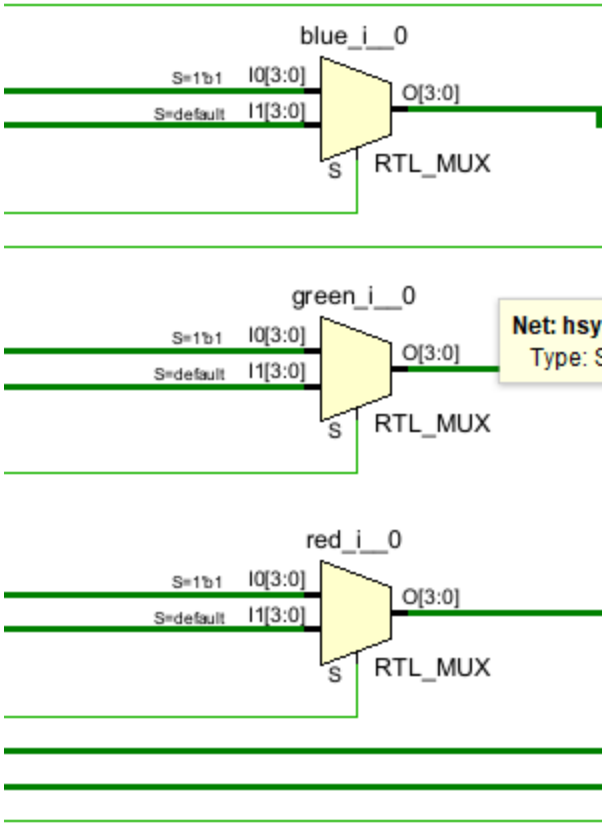

Figure 14. Overall RTL Design (View 7)

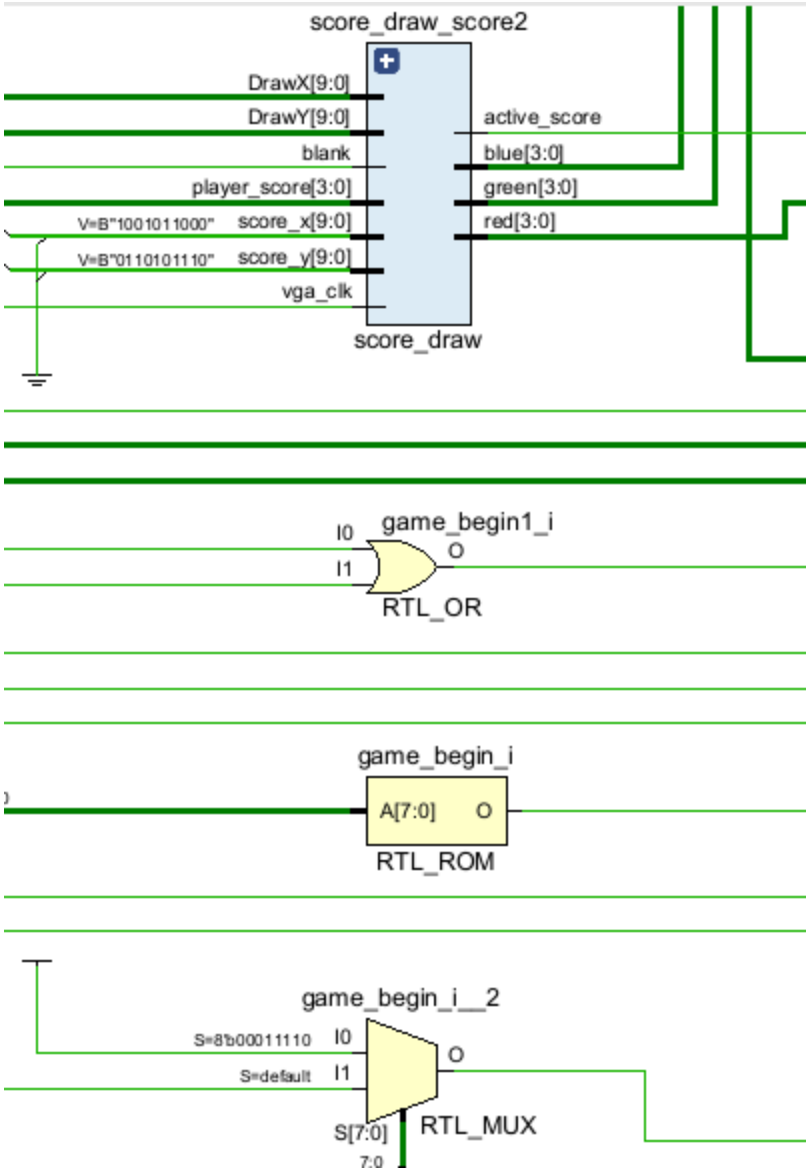

Figure 15. Overall RTL Design (View 8)

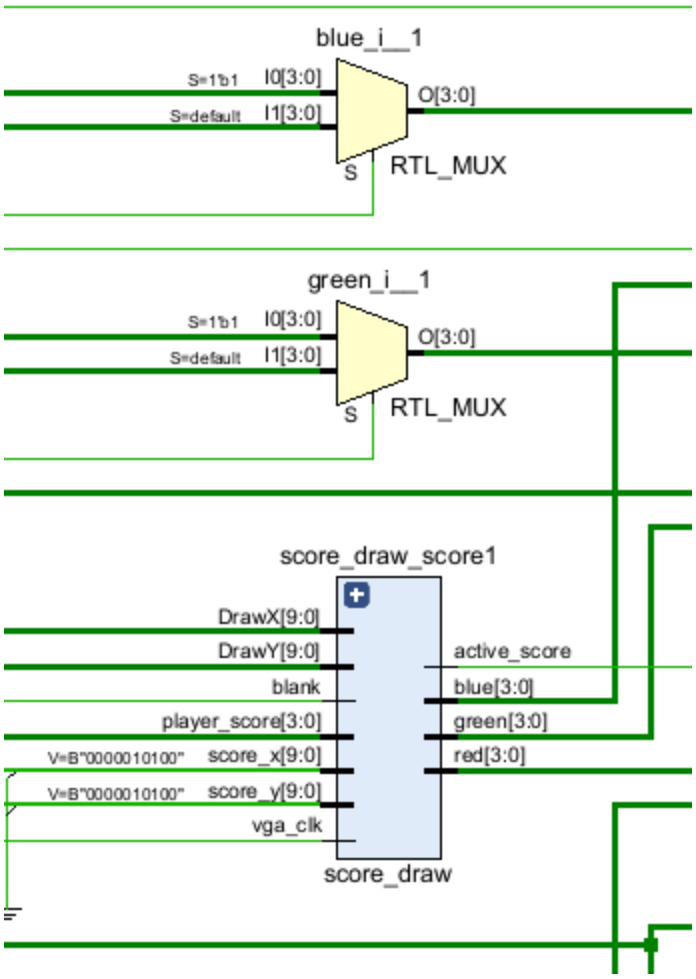

Figure 16. Overall RTL Design (View 9)

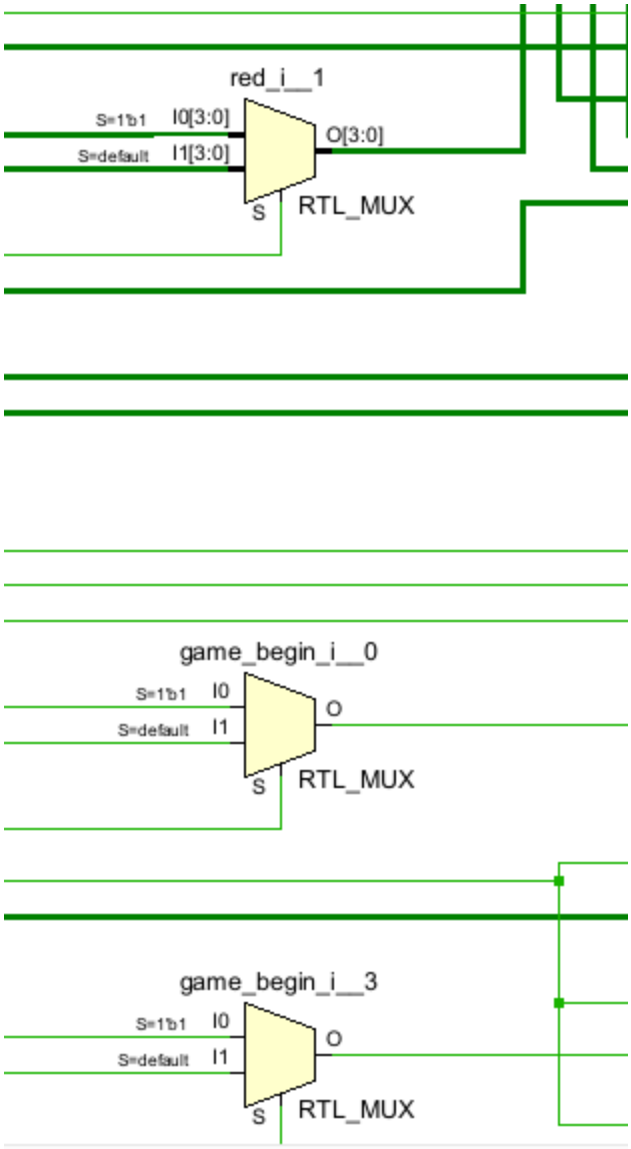

Figure 17. Overall RTL Design (View 10)

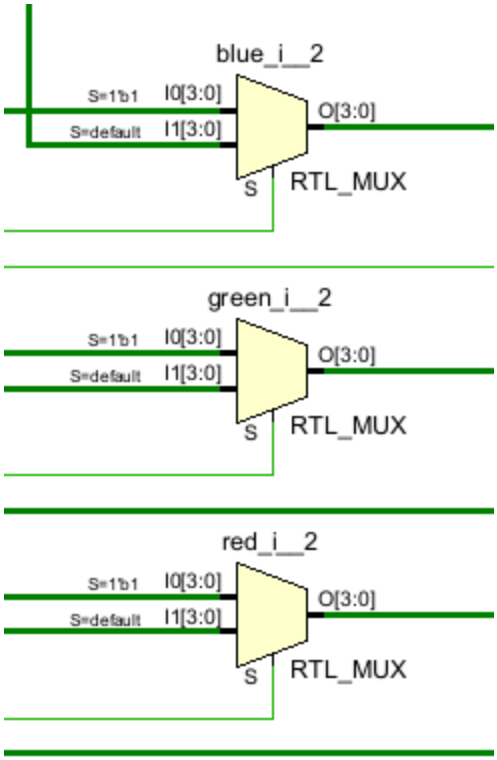

Figure 18. Overall RTL Design (View 11)

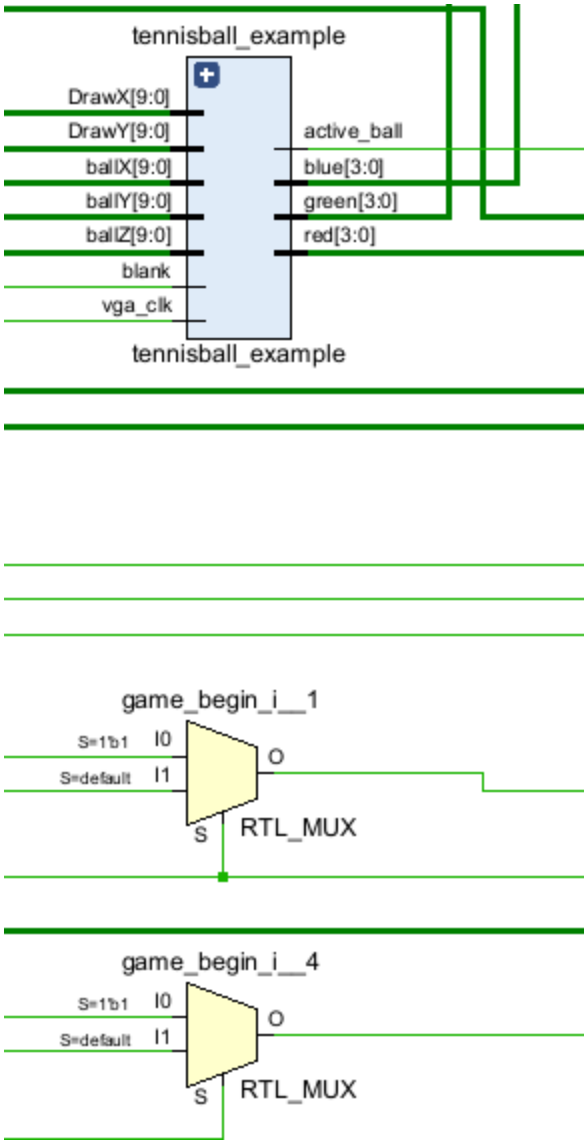

Figure 19. Overall RTL Design (View 12)

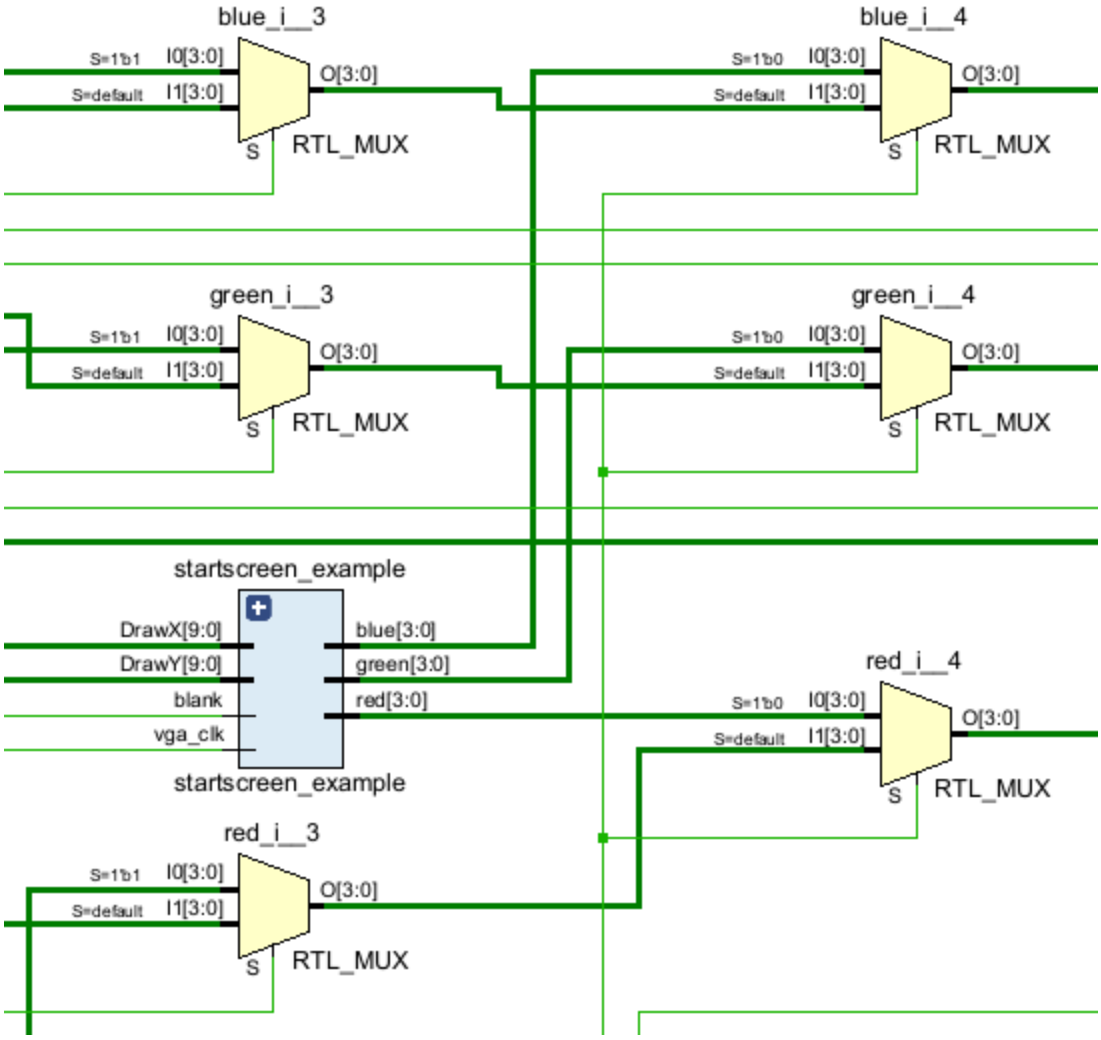

Figure 20. Overall RTL Design (View 13)

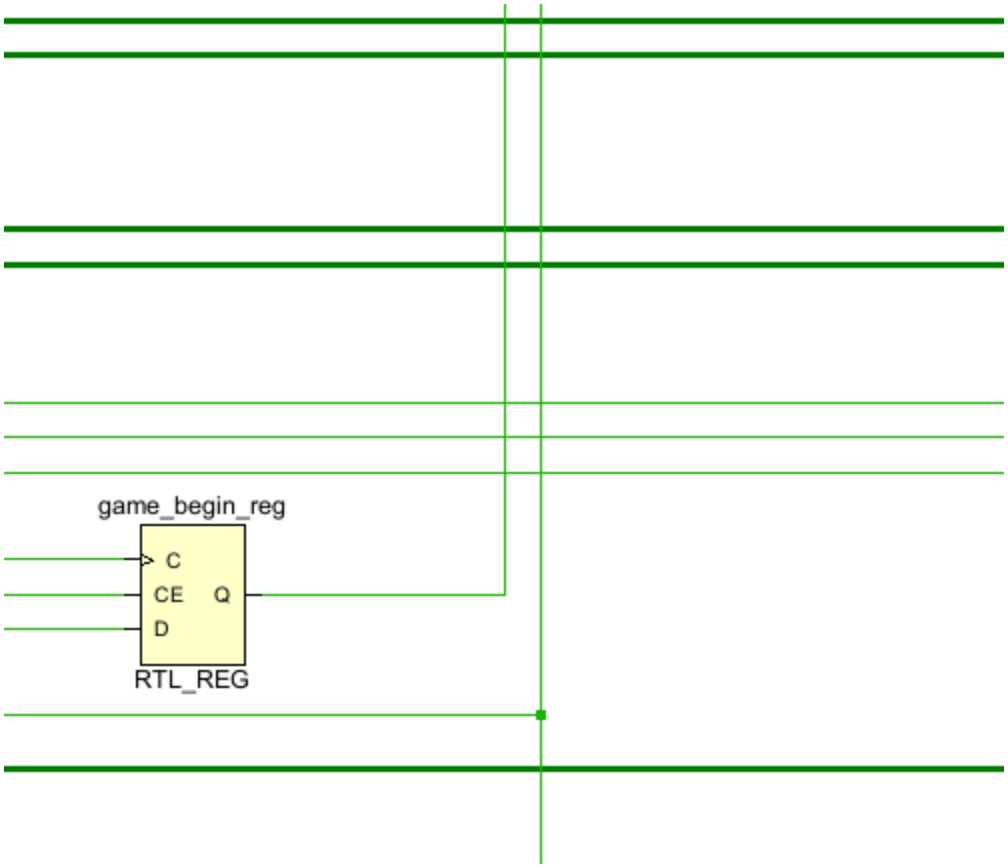

Figure 21. Overall RTL Design (View 14)

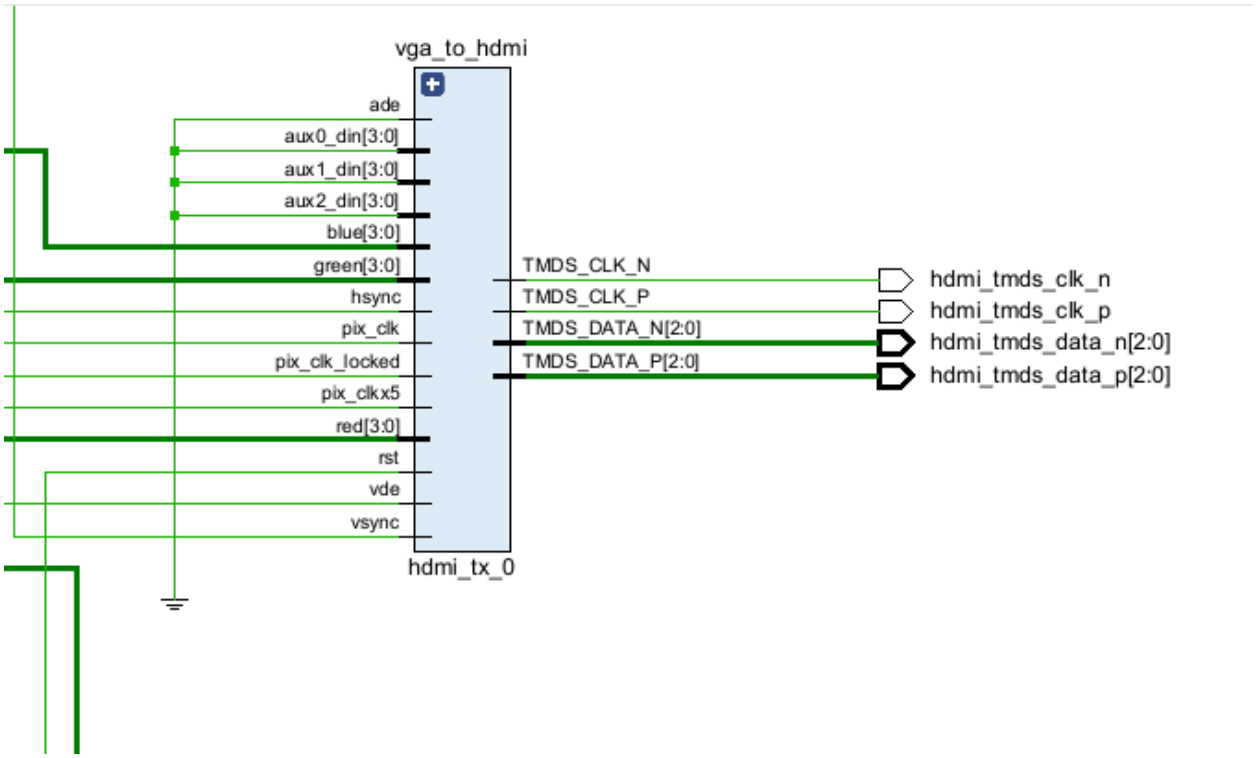

Figure 22. Overall RTL Design (View 15)

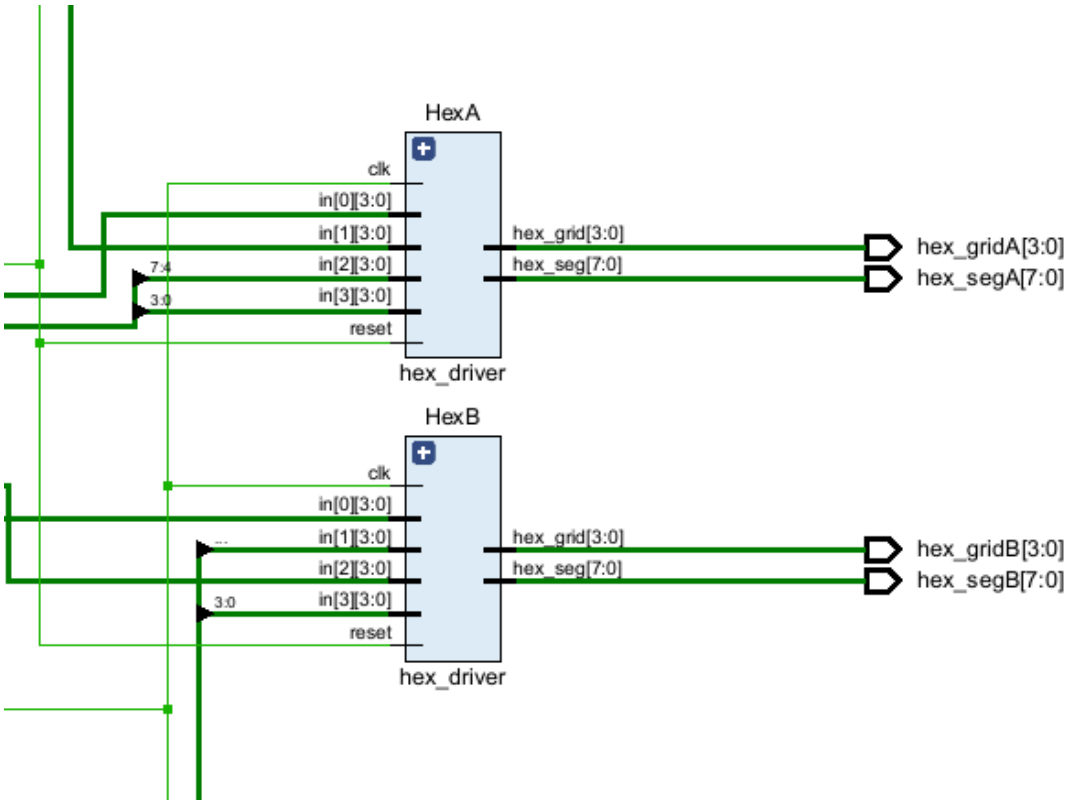

Figure 23. Overall RTL Design (View 16)

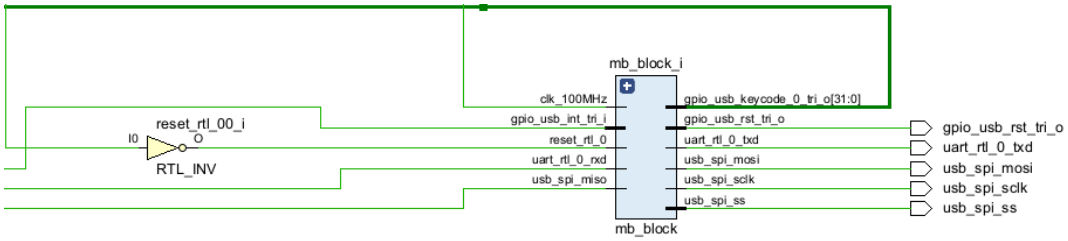

Figure 24. Overall RTL Design (View 17)

(4) Module Descriptions

Module: tennisball_example.sv

Inputs: vga_clk, [9:0] DrawX, DrawY, blank, [9:0] ballX, [9:0] ballY, [9:0] ballZ

Outputs: [3:0] red,[3:0] green, [3:0] blue, active_ball

Description: This module computes a seam based on a radius offset and sets RGB based on the ball radius and seam, which varies with the Z positioning of the ball (bigger Z corresponds to bigger radius to bigger ball)

Purpose: This module draws the tennis ball.

Module: player_rom.sv (player1_rom, player1left_rom, tenniscourt_rom, startscreen_rom)

Inputs: Clk, addra (based on the number of pixels of the sprite)

Outputs: douta

Description: This ROM is the size of the number of pixels of the sprite/background. Each pixel has a number associated with it that indexes into the palette to choose the RGB values,

Purpose: It contains all the information to map to the palette.

Module: player_palette.sv (player1_palette, player1left_palette, tennisball_palette, tenniscourt_palette, startscreen_palette)

Inputs: [3:0] index

Outputs: [3:0] red, green, blue

Description: This module takes in an index and outputs 3 4 bit numbers

Purpose: This determines the palette for each sprite. It contains up to 16 combinations of RGB values that map to each pixel of the sprite in the ROM.

Module: player_example.sv (player1_example, player2_example)

Inputs: vga_clk, [9:0] DrawX, [9:0] DrawY, blank, [9:0] player_x, [9;0] player_y (this varies with the player), pSwing (varies with player)

Outputs: [3:0] red, [3:0] green, [3:0] blue, active (changes with player)

Description: This initializes the rom and palette modules, and then sets active signals as well as the rgb outputs based on the palette. It sets active to 1 when we are in the sprite bounds and the color is not white and otherwise 0.

Purpose: This module combines all the functionality to draw the sprite in the proper position based on the input positions from other modules. We also essentially remove the background here by turning off the active signal when the white color occurs (which only happens in the background of the sprite). Note the colors for player 2 are inverted of player 1.

Module: movement.sv

Inputs: Reset, frame_clk, [7:0] keycode 1, [7:0] keycode 2, [7:0] keycode 3, [7:0] keycode 4

Outputs: [9:0] player1_y, [9:0] player1_x [9:0] player2_y, [9:0] player2_x, p1Swing, p2Swing

Description: All possible valid values for the keycodes are checked and consequentially, the swing variables and the speed variables are set for each player. We then send the player positions within bounds and determine their next position based on their current position and speed.

Purpose: First, the player positions are set to their side of the court at the beginning of the game. The player sprites are then restricted to be only able to move on their half of the court. Based on the user input (WASD and arrow keys) we set the player speed and direction in X and Y coordinates. We output those to redraw the sprite in their new positions along with swing variables (when the user presses G or J) which are used in other modules to compute if the ball was hit.

Module: digit_rom_bold.sv

Inputs: [3:0] digit

Outputs: [1499:0] bitmap

Description: This module takes in a 4 bit input and in cases 0-9 outputs a 1500 bit value, otherwise defaults to 1500'd0.

Purpose: This module is used to get the bitmapping for the drawing of the score output (30x50 pixels). The input is a player's score from 0-9.

Module: score_draw.sv

Inputs: vga_clk, [9:0] DrawX, [9:0] DrawY, blank, [3:0] player_score, [9:0] score_x, [9:0] score_y

Outputs: [3:0] red, [3:0] green, [3:0] blue, active_score

Description: This sets the active_score signal along with the RGB outputs, taking in the bitmap from the digit rom and checking when it is in bounds to draw the sprite.

Purpose: This is instantiated twice (one for each player's score), and draws them in a 30x50 box as essentially static sprites.

Module: ball_movement.sv

Inputs: Reset, frame_clk, p1Swing, p2Swing, [9:0] p1PosX, [9:0] p1PosY, [9:0] p2PosX, [9:0] p2PosY, game_restart

Outputs: [3:0] player1_score, [3:0] player2_score, [9:0] ballX, [9:0] ballY, [9:0] ballZ, [3:0] bounce_display, [3:0] ball_speed_z_output

Description: This module assigns points for each player and handles logic for identifying when a bounce occurs, when the ball is being "served" and when the ball is inBounds. Hits are assigned based on the ball hitbox & if the hit button is pressed for the player. The court bounds are precomputed and continuously checked. Once the ball is out of play, the point ends. When either player scores a point or the game starts, all variables are reset. The ballZ position increases when hits occur or the ball bounces, but decreases under any other situations as an effect of gravity. When the ball is hit, a pseudo-random computation occurs for the ballX and Y speed. For example, for the ballX speed when player 1 hits, it gets set to the following: ball_speed_x <= ((p1PosX * 7 + p2PosX * 5 - recent_hit << 4) % 3) - 1; The goal of this is to keep the game fun through random speeds, but it is bounded in the range of -1 to 1. Other major computations include bounding box computations (where each sprite & ball is approximated as a square) and scoring computation which was previously detailed above.

Purpose: This module handles most of the computation including setting the ball XYZ positions, speeds, hitbox logic, and scoring/resetting play logic.

Module: VGA_controller.sv

Inputs: pixel_clk, reset

Outputs: hs, vs, active_nblank, sync, [9:0] drawX, [9:0] drawY

Description: This module sets and increments the hc & vc signals based on hs & vs limits and feeds that to both draw signals.

Purpose: This module essentially controls the VGA actual output. It loops through pixels on the screen and sets the drawX and drawY signals.

Module: hex_driver.sv

Inputs: clk, reset, [3:0] in[4]

Outputs: [7:0] hex_seg, [3:0] hex_grid

Description: The module outputs four 4-bit hex values (0-F) utilizing clock signals to light them up on the clock edge and LUTs to get the hex output.

Purpose: This module controls the display of the 16-bit hex values on the FPGA board.

Module: mb_usb_hdmi_top.sv

Inputs: Clkl, reset_rtl_0, [0:0] gpio_usb_int_tri_i, usb_spi_miso, uart_rtl_0_rxd

Outputs: gpio_usb_rsi_tri_0, usb_spi_mosi, usb_spi_sclk, usb_spi_ss, uart_rtl_0_txd, hdmi_tmds_clk_n, hdmi_tmds_clk_p, [2:0] hdmi_tmds_data_n, hdmi_tmds_data_p, [7:0] hex_segA, [3:0] hex_gridA, [7:0] hex_segB, [3:0] hex_gridB

Description: We instantiate the hex driver modules, the block diagram, the clocking wizard, the vga to hdmi converter, the vga controller, the movement and ball_movement modules, the 6 modules that draw the sprites (2 scores, 2 players, tennisball, background). We also include an always_comb block that checks the active signals and sets the rgb values.

Purpose: This is the top level module that instantiates all the other modules. It includes logic that checks the active signal for each sprite and draws them accordingly, defaulting to the background. It prioritizes the sprites such that the ball is on top, the scores are overlaid below, the sprites are next prioritized, and the background comes last.

Block Design Components

Block: xlconcat_0
Description: This is a concat block that takes in the various interrupt signals from the AXI blocks and concats them into 1 output that can be used for the interrupt controller.

Block: spi_usb
Description: This is an AXI Quad SPI block that connects the AXI protocol to the slave devices in SPI so that we can communicate with them.

Block: timer_usb_axi
Description: This is an AXI timer block that essentially just generates a clock for the program to use.

Block: gpio_usb_keycode
Description: This is an AXI GPIO block that lets us use the keycodes from presses on the keyboard.

Block: gpio_usb_int

Description: This is an AXI GPIO block that creates an interrupt signal for the usb functionality. This allows us to not have to constantly poll and waste resources to see if something happened.

Block: gpio_usb_rst
Description: This is an AXI GPIO block that creates the reset functionality for the usb. This allows us to reset and have a known state when we initially start.

Block: axi_uartlite_0
Description: This is an AXI uartlite block that lets us set up an asynchronous serial communication protocol so we can communicate between devices. (This is what allows us to communicate with the keyboard and MicroBlaze).

Block: microblaze_0
Description: This is the overall main processor for our design, that is a Microblaze block that handles our hardware functionality.

Block: clk_wiz_1
Description: This is a clocking wizard that allows us to manipulate our clock signal to achieve our wanted rates (25 & 125 MHz) and make sure all the various clocks are synced.

Block: mdm_1
Description: This is a Microblaze Debug Module that essentially provides us various tools that help us debug our Microblaze system.

Block: microblaze_0_local_memory
Description: As the name suggests, this is a local memory storage block that allows for fast on-chip memory access

Block: rst_clk_wiz_1_100M
Description: This is a Processor System Reset block that allows us to reset the system as a whole.

Block: microblaze_0_axi_periph
Description: This is an AXI Interconnect block that helps combine all the various AXI signals and choose operations and acts as essentially an overall controller to combine all the various operations.

Block: microblaze_0_axi_intc
Description: After the concat module combines the various interrupt signals, they go to this AXI Interrupt Controller block. This block prioritizes and sorts the signals and then gives it to the main processor which then determines what to actually do based on the interrupt.

Block: vga_to_hdmi
Description: This block converts the VGA signal that we are working with into an HDMI signal that we can output on our monitor.

(5) Simulation Waveform and Images

We did not run any simulations for this project. However, below you can see various pictures of the project functioning.

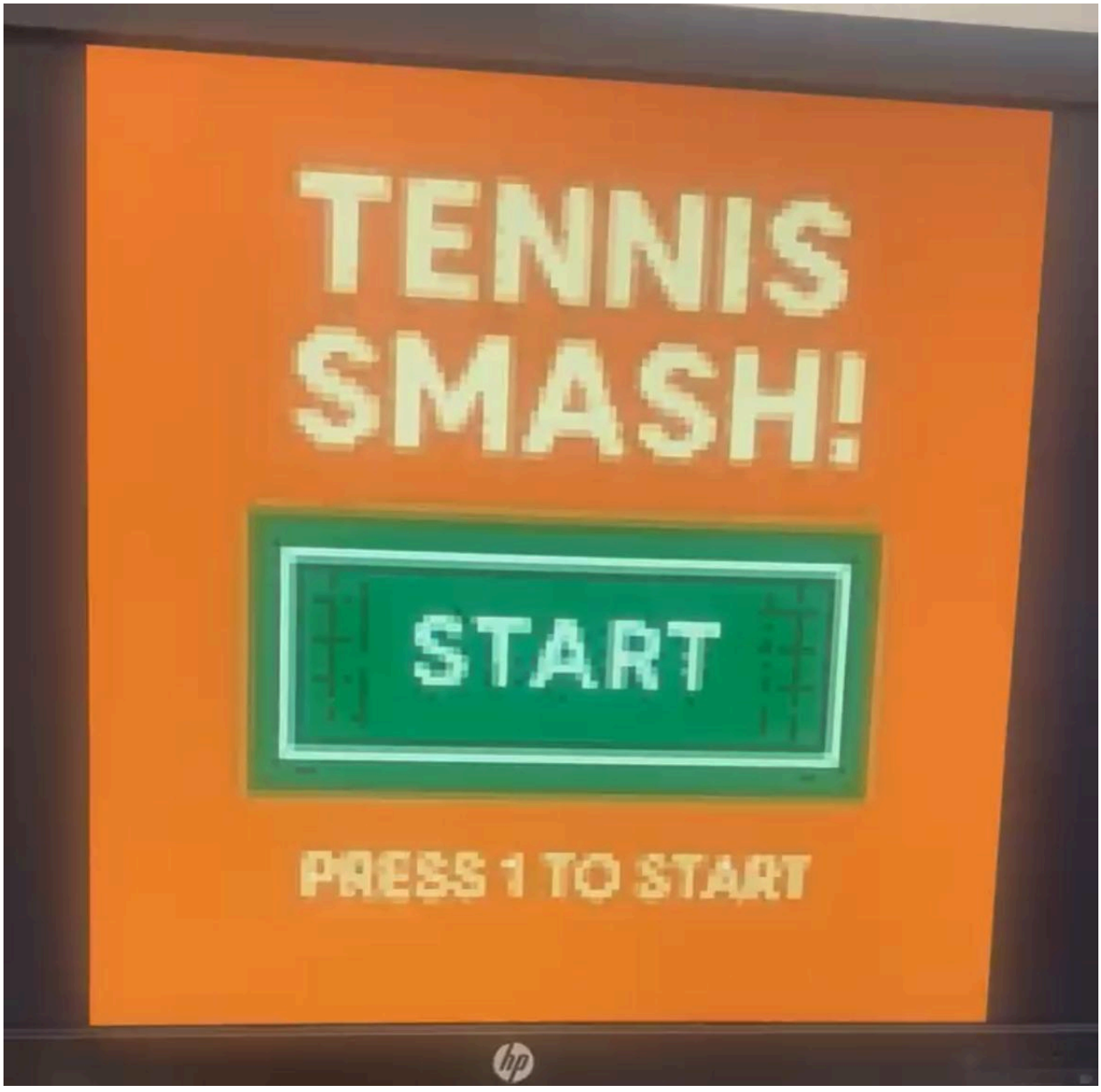

Figure 25. Tennis Smash Title Screen

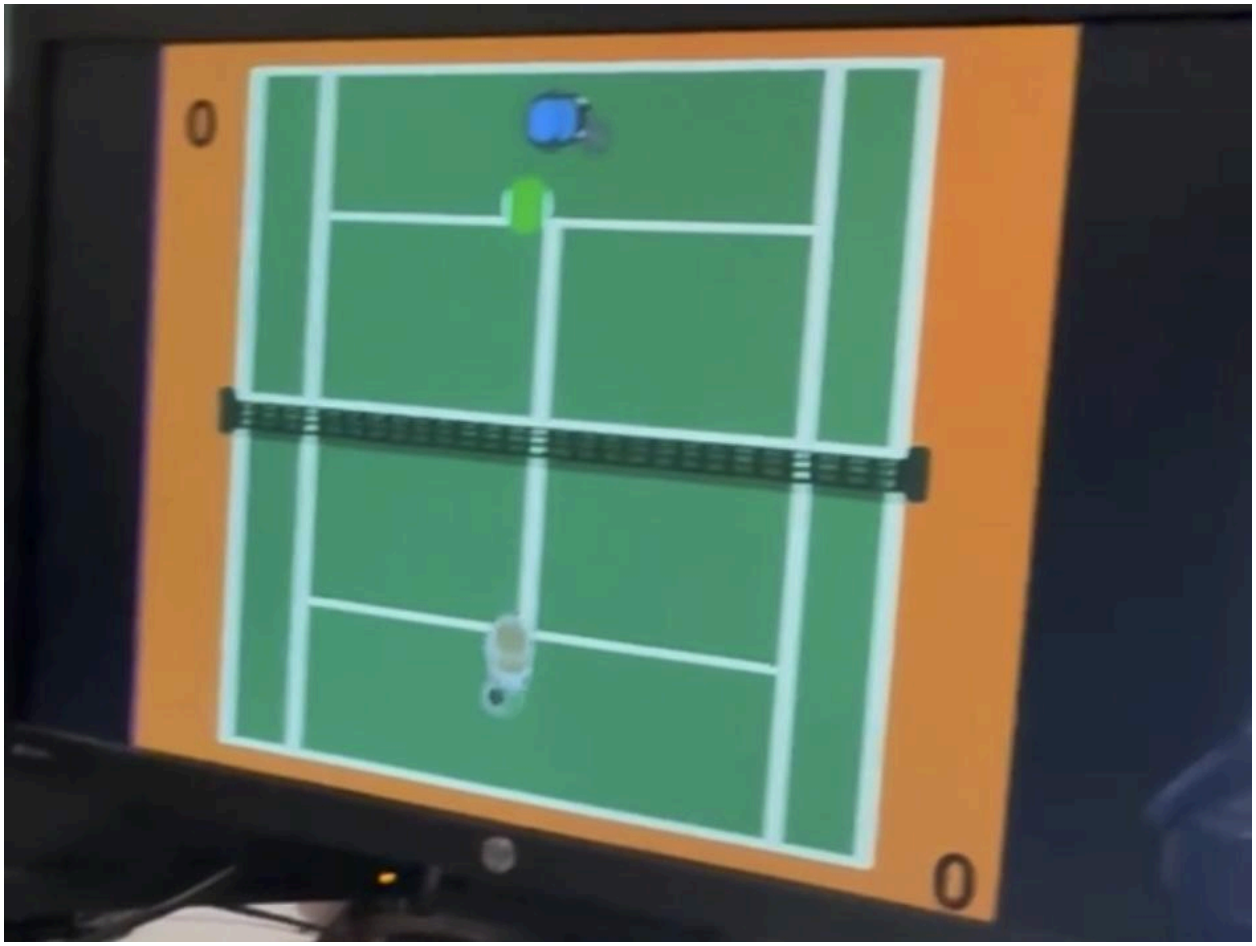

Figure 26. Tennis Smash Gameplay (View 1)

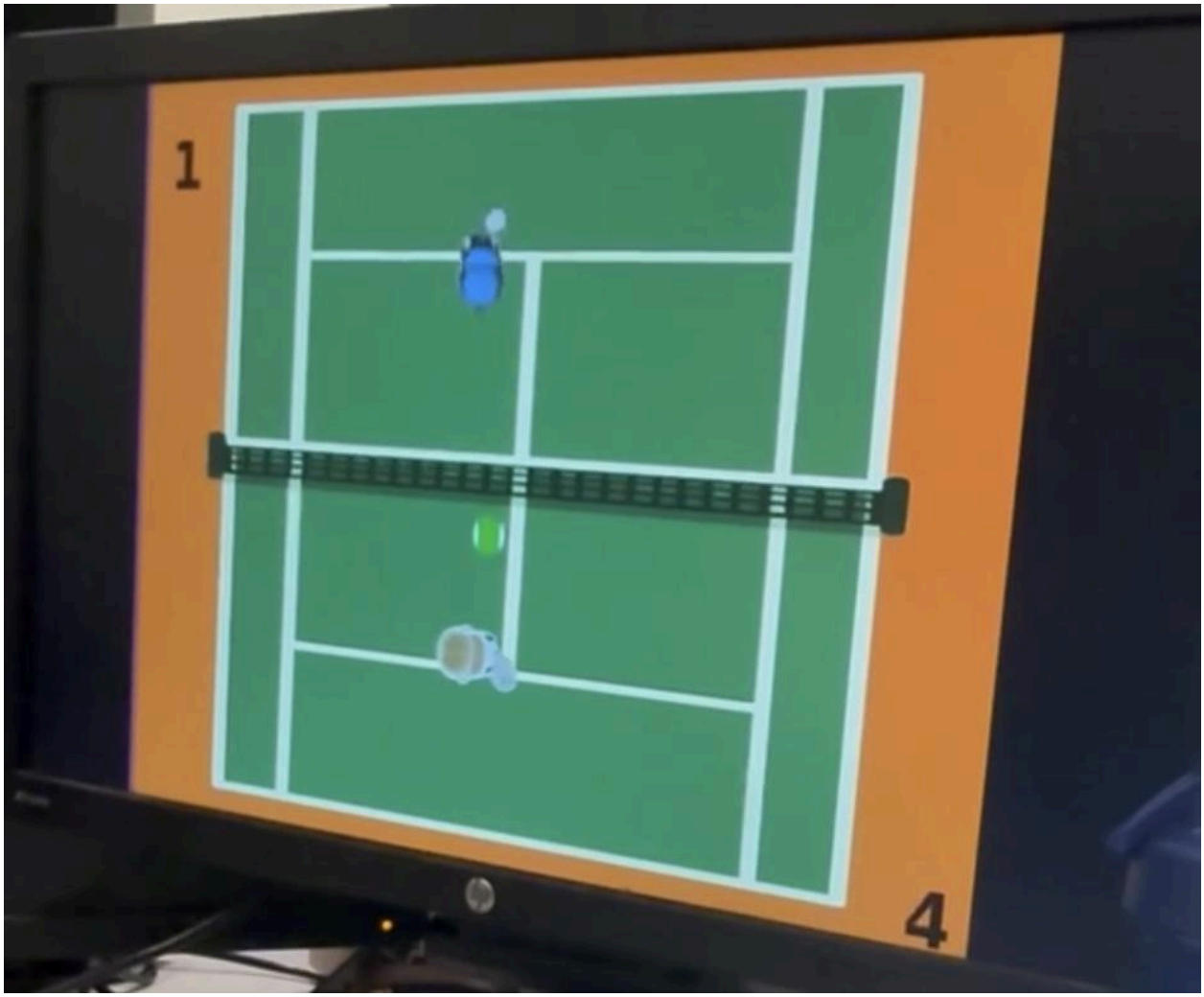

Figure 27. Tennis Smash Gameplay (View 2)

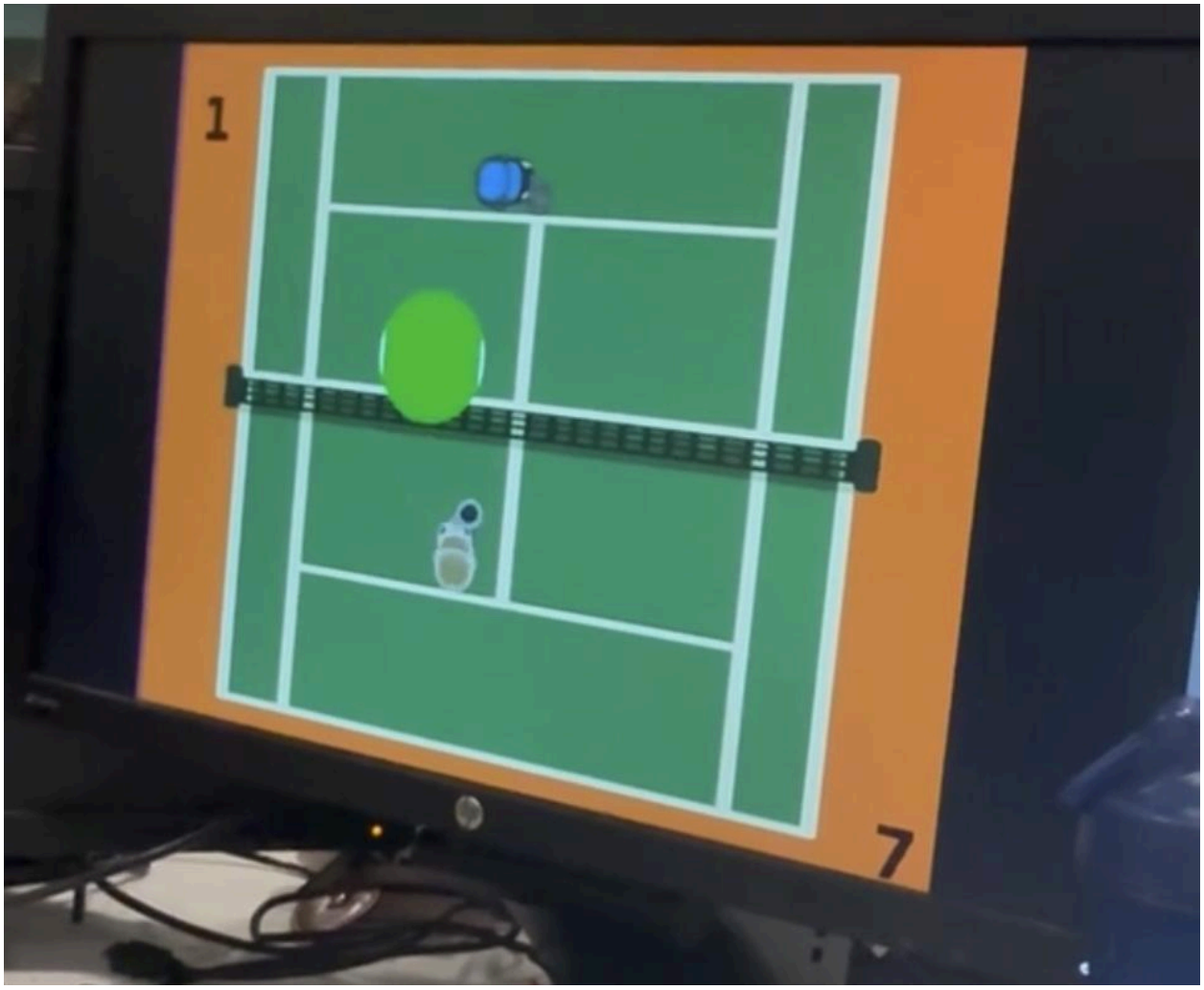

Figure 28. Tennis Smash Gameplay (View 3)

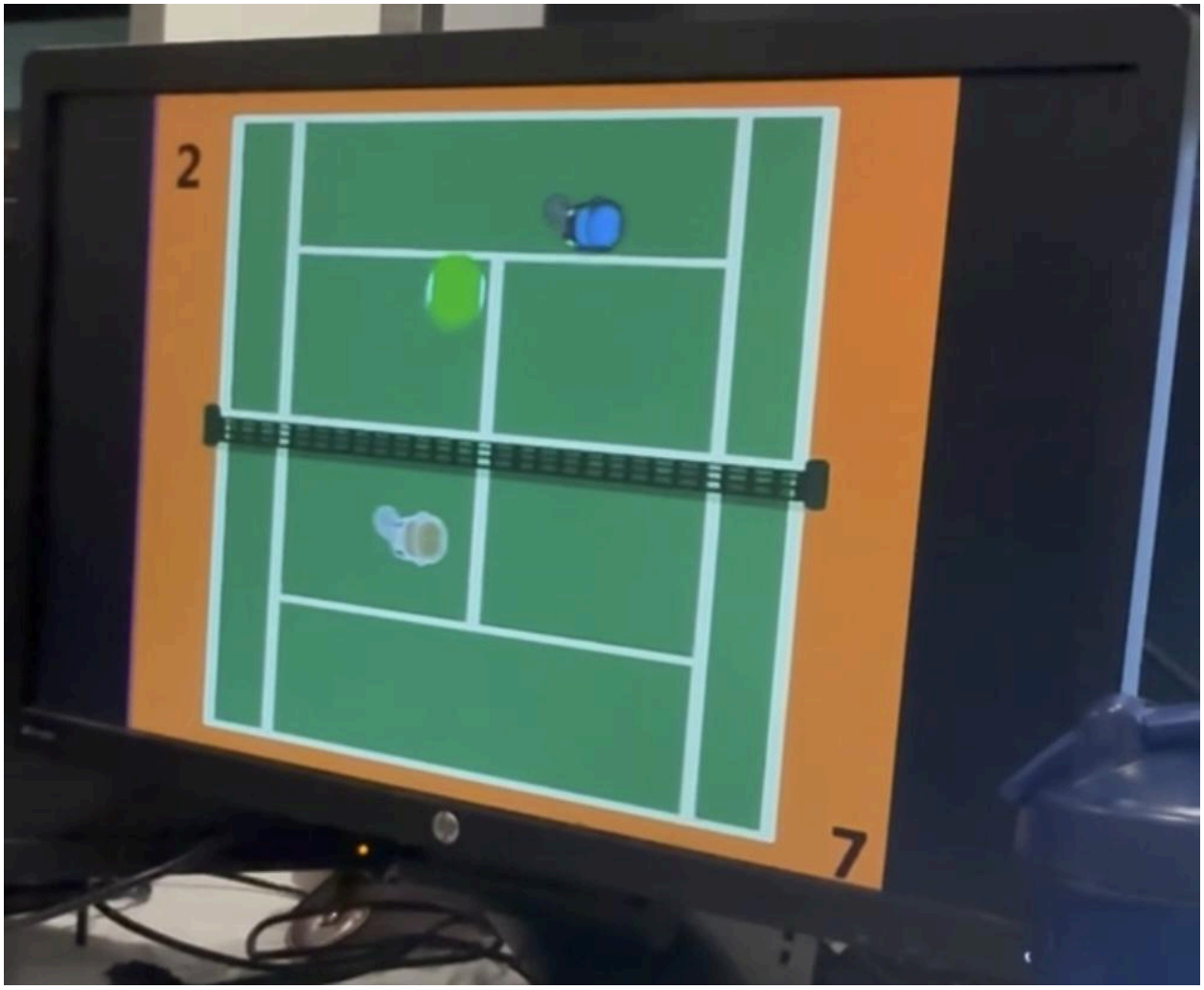

Figure 29. Tennis Smash Gameplay (View 4)

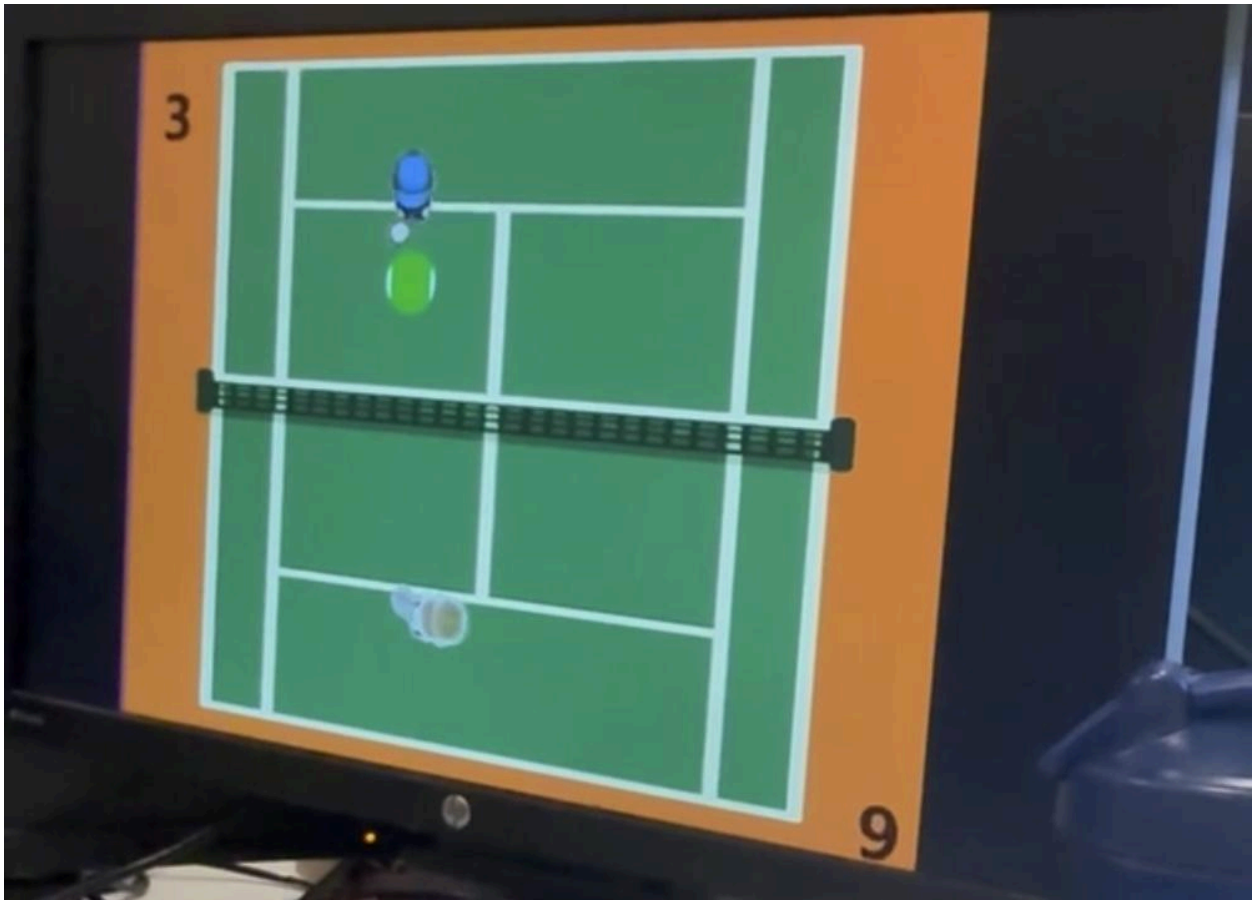

Figure 30. Tennis Smash Gameplay (View 5)

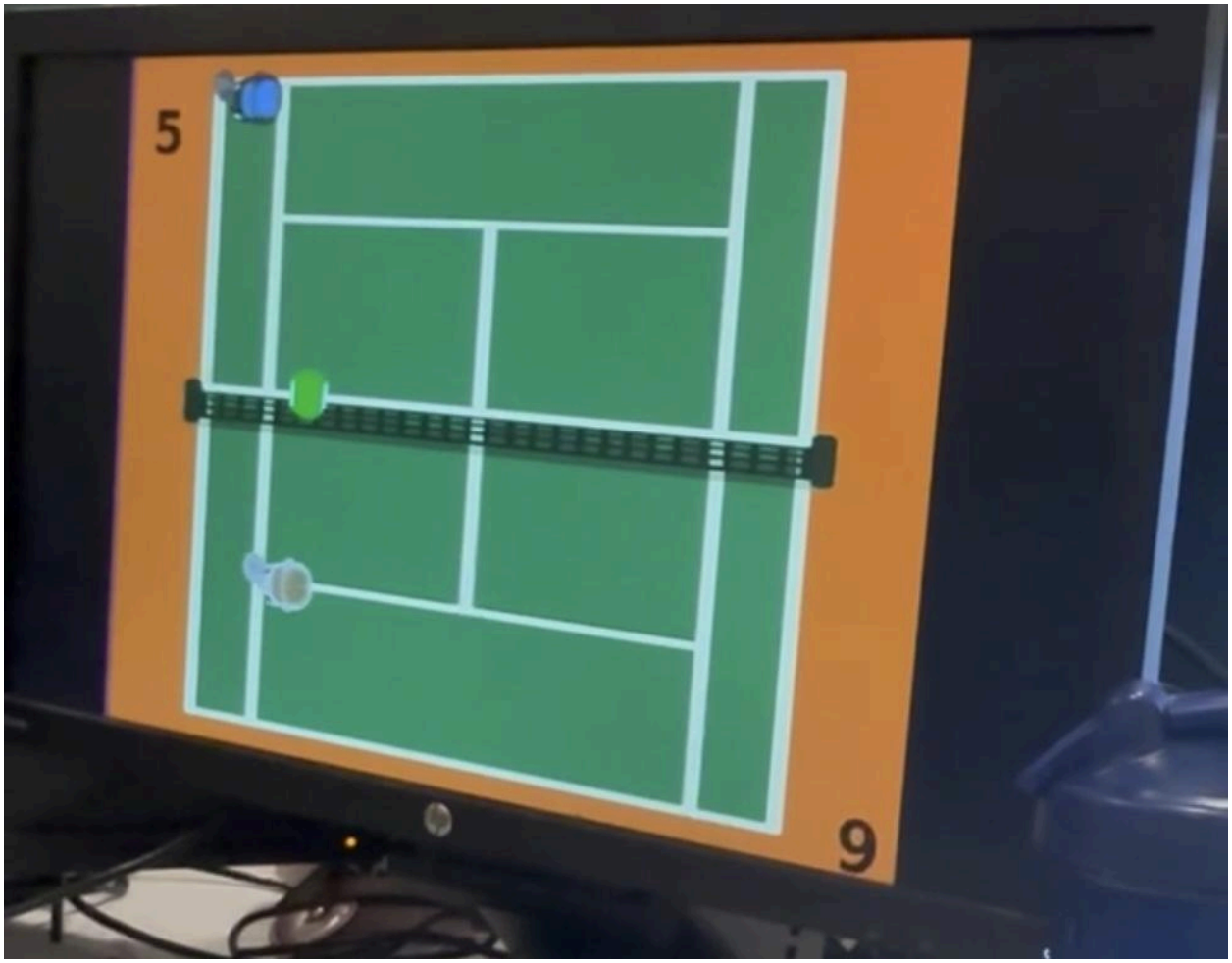

Figure 31. Tennis Smash Gameplay (View 6)

(6) Design Resources & Statistics

LUT	4197
DSP	10
Memory (BRAM)	49.5
Flip-Flop	3244
Latches*	0
Frequency	77.9 MHz
Static Power	.078 W
Dynamic Power	.401 W
Total Power	.479 W

(7) Conclusion

Our final project is a 2-player tennis game, *Tennis Smash*, that is implemented on our Spartan-7 FPGA Urbana board. The project features real-time gameplay that is controlled through the keyboard, where each player uses the WASD and arrow keys for movements, and the G and J keys for respective swinging to hit the tennis ball. The overall system is capable of tracking both players' positions and the dynamics of the tennis ball. The XYZ components are used to simulate realistic ball deflection and bouncing behavior. The game concludes when one player has reached 10 points, and is capable of being replayed. The system features sprite animations, a starting title screen, and the restart functionality once the game has been won by one of the players.

Future work on this project would be implementing audio, in which we would utilize the audio port on the FPGA to be connected to either a speaker or headphones to play audio during gameplay. To achieve audio output, we were planning on utilizing a Pulse Width Modulation signal in which an 8-bit audio sample is read from HEX characters in an external file. The PWM signal would switch between the high and low states, where it would approximate an analog waveform. This generated signal would be output to the left and right audio output pins (N13 and N14) on the Urbana Board. This output of varying duty cycles would then be interpreted as audio, where we would have a counter loop that would repetitively play the audio sample continuously throughout the gameplay. We were able to attain a faint repeating static buzz to be passed through the audio port when reading from our sample file, but we were unable to further this by achieving an audible melody/tune to be repeatedly played through the gameplay, likely caused by a misconfiguration in our signaling which was unable to be resolved in time.