ECE 385

Spring 2019

Final Project

Super Smash Bros. on System Verilog

Usha Tripuramallu and Aashna Wadhwa

ABJ Friday 2:00 - 4:50 pm

Xinbo Wu

I. Introduction

For our final project, we designed and implement a two-player version of Super Smash Mario Bros, the video game, on the Bridge of Eldin map (https://www.youtube.com/watch?v=8Q7d11IFzuk) and the PictoChat map (https://www.youtube.com/watch?v=zl_tkLivsPA) on the FPGA and interface with the VGA ports and and the NIOS IIe processor. Essentially, the two players can fight on different maps using preselected characters and attack each other to hurt the other player's health and win by finishing their health.

The game graphics are displayed on the monitor and the users are able to control the game using the keyboard. The actual game features included are character graphics, using the game sprites, scores for each player, and keyboard controls. The implementation, other than interfacing with the keyboard, which is done using the NIOS II CPU, is done in hardware.

II. Description & Operation of the Circuit

Design Abstraction

The design contains five parts: control logic, rendering logic, ROM, VGA display, and input devices.

The control logic is all implemented in System Verilog. It takes in the user input and communicates with the rendering logic to decide which sprite to print. Additionally, it also calculates score based on if one of the characters is attacking the other and is within a certain pre-dictated distance of the other character. It also keeps track of the 1-minute game timer, which it calculates based on the clock. Lastly, the control logic has an FSM, which keeps track of the logo, start, game, attack, and end states. These states help us set signals such as attack, which is

used to keep track of score, to implement additional features and also help us reset everything if the reset key is pressed by the user.

The rendering logic works with the ROM to produce the graphics on the screen. If there are multiple sprites in the same location, our logic prioritizes the characters over the background maps and the health bars. It also changes the sprites it paints based on keyboard input. For example, if a character is moving towards the left, it flips the original sprite, which shows the character moving to the right, to show motion towards the left.

The ROM encodes the sprites into the on-chip memory, reads them based on the current VGA X and Y coordinates and returns the color to be printed on the screen.

The VGA display handles actually drawing on the screen. It constantly goes across the screen based the VGA clock and prints the pixel based on the rendering logic.

The input device is a keyboard that is implemented using PIO blocks that take in keyboard input and interact with the NIOS II CPU for the purposes of interfacing with the USB keyboard as in Lab 8 to emulate the controllers for the game. Software C code is used for io_read, io_write, usb_read, and usb_write.

Operation of the Circuit

To start the game, we program the System Verilog code onto the FPGA and run the USB controller keyboard code on Eclipse, similar to Lab 8.

The game begins at the start screen, with the Super Smash Bros. logo and some music. It then proceeds to letting the users choose the map they want to play on, and then moves to that map with our characters, Mr. Game & Watch and Kirby.

On the game screen, the first user, playing with Mr. Game & Watch, can use “WAD” to jump, move left, and move right, respectively, and “X” to attack. The second player, playing

with Kirby, can use “IJL” to jump, move left, and move right, respectively, and “M” to attack. The characters are animated to move in the direction and attack when dictated by the user. When a player jumps up, we implemented gravity for the character to come back back down to the bridge. Additionally, a player can also jump over another character. When one player is able to go within a certain range of another player and attack them, they decrease the other player’s health by a pre-dictated amount, which is reflected in the health bars. Once one of the players loses their whole health, the other player wins and the game ends. At the end of the game, you see the winning player. We also implemented a timer for 60 seconds, which is based on the clock. Once the timer runs out, if neither of the players has already won, there is a tie.

The characters stand on the base in the map and can only move horizontally and upwards and cannot move beyond the boundaries of the screen. Additionally, each character has multiple sprites associated with it to show movement to the left and right and another sprite for attacking. This lets us animate the characters while in motion.

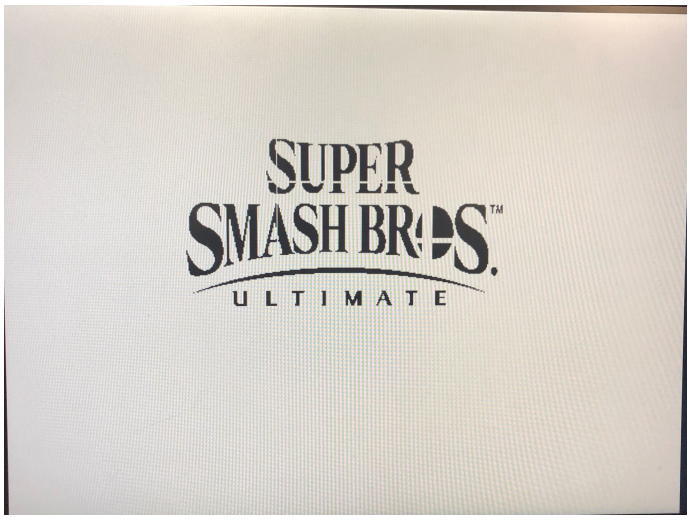


Figure 1 - Game start with logo

Figure 2 - Bridge of Eldin map shown with Kirby jumping

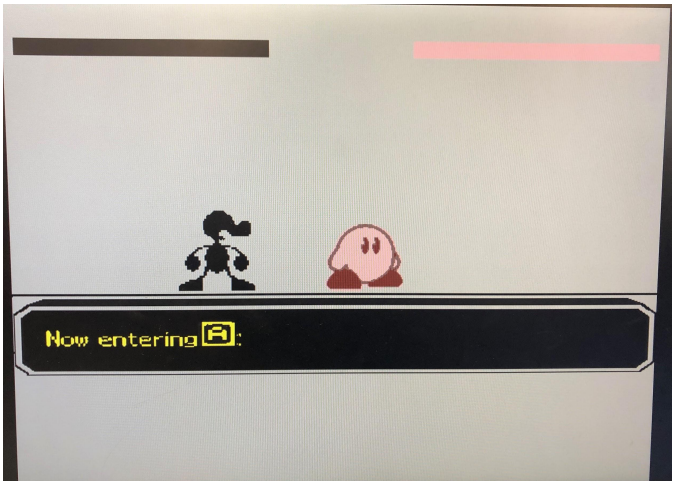


Figure 2 - Pictochat as second map choice, chosen with Key 2

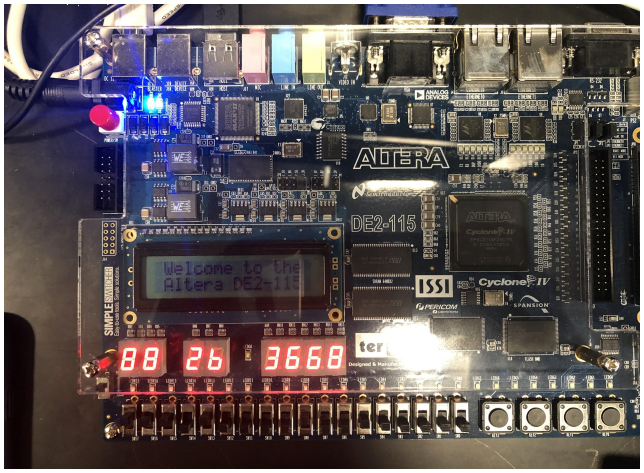


Figure 4 - Hex Drivers 5 and 4 show the game timer while Hex Drivers 3 and 2 show player 2's score and Hex Drivers 1 and 0 show player 1's score, all in hex

III. Modules in Hardware

1. Hardware Modules in Qsys:

Module: nios2_gen2_0 **Inputs:** clk, reset_n, reset_req, [31:0] d_readdata, d_waitrequest,[31:0] i_readdata, i_waitrequest, [31:0] irq, [8:0] debug_mem_slave_address, [3:0] debug_mem_slave_byteenable, debug_mem_slave_debugaccess, debug_mem_slave_read, debug_mem_slave_write, [31:0] debug_mem_slave_writedata **Outputs:** [28:0] d_address, [3:0] d_byteenable d_read, d_write, [31:0] d_writedata, debug_mem_slave_debugaccess_to_roms, [28:0] i_address, i_read, debug_reset_request, [31:0] debug_mem_slave_readdata, debug_mem_slave_waitrequest, dummy_ci_port **Description:** This is the Nios II processor, used to communicate with software. **Purpose:** This modules sends and receives the appropriate signals to and from the the NIOS II processor, which enables the machine to actually read in and store all data. The processor sends control signals to all other parts of the hardware, allowing them to function when necessary.
Module: onchip_memory2_0 **Inputs:** address, byteenable, chipselect, clk, clken, freeze, reset, reset_req, write, writedata **Outputs:** readdata **Description:** This instance serves as the on-chip memory block. **Purpose:** This modules instantiates a small on-chip RAM as a placeholder block and sends it the appropriate signals and reads the returned data, allowing for the highest throughput and the lowest latency.
Module: Jtag_uart_0 **Inputs:** clk, reset, s1 **Outputs:** irq **Description:** This module is used for communication between the PC and NIOS II. **Purpose:** This module is used to let us use the console when using Eclipse.
Module: sdram **Inputs:** az_addr, az_be_n, az_cs, az_data, az_rd_n, az_wr_n, clk, reset_n, **Outputs:** za_data, za_valid, za_waitrequest, zs_addr, zs_ba, zs_cas_n, zs_cke, zs_cs_n, zs_dq, zs_dqm, zs_ras_n, zs_we_n **Description:** This module is responsible for interactions with the SDRAM (synchronous dynamic RAM) and sends it the appropriate signals to make it function and receives the appropriate signals to send to other modules.

Purpose: This module interacts with the off-chip SDRAM, which is used to store the software program in memory and sends the instructions to the processor to actually execute the desired operations.
Module: sdram_pll **Inputs:** [1:0] address, areset, clk, configupdate, [3:0] phasecounterselect, phaseupdown, phasestep, read, reset, scanclk, scanclkena, scandata, write, [31:0] writedata **Outputs:** c0, c1, locked, phasedone, [31:0] readdata, scandataout, scandone **Description:** This is a PLL (phase-lock loop) component, which allows the system to generate and stabilize the clock signal. **Purpose:** This module instantiates the PLL component, which provides the required clock signal for the SDRAM chip. This is because the SDRAM requires precise timings, and the PLL allows us to compensate for clock skew due to the board layout.
Module: keycode **Inputs:** address, chipselect, clk, reset_n, write_n, writedata **Outputs:** out_port, readdata **Description:** Holds the actual key pressed by the host from the keyboard. **Purpose:** Used to translate key press to appropriate ascii value to be handled in system verilog.
Module: otg_hpi_address **Inputs:** address, chipselect, clk, reset_n, write_n, writedata **Outputs:** out_port, readdata **Description:** Indicates the register address at which to read or write in the chip. **Purpose:** Used for IO_write and IO_read to indicate which register is being used for the USB to write to or read from.
Module: otg_hpi_data **Inputs:** address, chipselect, clk, reset_n, write_n, writedata **Outputs:** out_port, readdata **Description:** Holds the data to either be input to the chip or the data read from the chip. **Purpose:** Used for IO_write and IO_read to input the data or read out the data, respectively.
Module: otg_hpi_r **Inputs:** address, chipselect, clk, reset_n, write_n, writedata **Outputs:** out_port, readdata **Description:** Indicates to the system when the chip is in read mode. **Purpose:** Used for IO_read to indicate that we have read from the chip.

Module: otg_hpi_w **Inputs:** address, chipselect, clk, reset_n, write_n, writedata **Outputs:** out_port, readdata **Description:** Indicates to the system when the chip is in write mode. **Purpose:** Used for IO_write to indicate that we have written to the chip.
Module: otg_hpi_cs **Inputs:** address, chipselect, clk, reset_n, write_n, writedata **Outputs:** out_port, readdata **Description:** Indicates when the chip is ready for data transaction. **Purpose:** Used for IO_write and IO_read to indicate that the chip has been selected correctly.
Module: otg_hpi_reset **Inputs:** address, chipselect, clk, reset_n, write_n, writedata **Outputs:** out_port, readdata **Description:** Indicates when to reset the OTG chip. **Purpose:** Used for IO_initialization.

2. Hardware Modules in System Verilog:

Top Level
Module: lab8 **Inputs:** CLOCK_50, [3:0] KEY **Inout:** [15:0] OTG_DATA, [31:0] DRAM_DQ **Outputs:** [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, [7:0] VGA_R, [7:0] VGA_G, [7:0] VGA_B, VGA_CLK, VGA_SYNC_N, VGA_BLANK_N, VGA_VS, VGA_HS, [1:0] OTG_ADDR, OTG_CS_N, OTG_RD_N, OTG_WR_N, OTG_RST_N, OTG_INT, [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:** This modules calls the lab8 platform design and passes in all of the appropriate signals into the hardware, specifically, values for the sdram, the values of the buttons, and the values of the buttons. This way, the values could be used in software. **Purpose:** This is the top-level module that integrates the Nios II system with the rest of the hardware created within System Verilog. The top level connects all the different modules and also keeps track of the score by doing collision detection, or making sure that a player is within a certain distance from the other player when attacking, and checking that the correct keycode is pressed.

I/O Modules
Module: keycode **Inputs:** address, chipselect, clk, reset_n, write_n, writedata **Outputs:** out_port, readdata **Description:** Holds the actual key pressed by the host from the keyboard. **Purpose:** Used to translate key press to appropriate ascii value to be handled in system verilog.
Game Control Logics
Module: ball **Inputs:** Clk, Reset, frame_clk, [7:0] keycode, [9:0] DrawX, [9:0] DrawY **Outputs:** is_ball, [9:0] Ball_X_Pos, [9:0] Ball_Y_Pos **Description:** This module returns whether or not the current pixel belongs to the character or to the background, using the keycode and x and y coordinates. **Purpose:** This modules lets us realize if the character has hit the border walls so we can make sure the character goes in the other direction and does not cross the wall.
Module: FSM **Inputs:** Clk, Reset, [9:0] score1, score2, [5:0] game_timer, x_on, m_on, ball1_x, ball2_x, size, ball1_y, ball2_y **Outputs:** attack1, attack2, is_end_2, is_end_1, is_tie **Description:** This is the state machine for the game. **Purpose:** The purpose of this module is to send appropriate signals to the top level to indicate starting control logic for the logo screen, game, attacks for each player, a tie, and the end screen.
Module: reg_4 **Inputs:** CLK, Reset, [9:0] in_count, bound, count_up **Outputs:** [3:0] out_count **Description:** This module holds the values for counters that are 10 bits wide. **Purpose:** This module is used to store the counter that keeps track of score and how many times one character has attacked the other.
Module: timer **Inputs:** CLK, [5:0] count_in **Outputs:** [5:0] count_out **Description:** This is a counter for the number of frame cycles of VGA_VS. **Purpose:** This module was used for timing the game. Since VGA_VS is a 60 MHz clock, everytime 60 clock cycles of VGA_VS passed, one second passed. Therefore, when 3600 clock cycles passed, the game terminated.

Rendering Modules
Module: color_mapper **Inputs:** CLK, start, map1, map2, is_ball, is_ball2, [9:0] DrawX, [9:0] DrawY, [9:0] Ball1X, [9:0] Ball1Y **Outputs:** [7:0] VGA_R, [7:0] VGA_G, [7:0] VGA_B **Description:** This module performs object or shape rendering and coloring by putting out RGB signals to the monitor. **Purpose:** This module was used to compute RGB values given the current position of the wall and then draws the ball on the monitor using its measurements.
Modules: frameRAM, frameRAMGameWatch, frameRAMGameWatch2, frameRAMGameWatchAttack, frameRAMGameWatchStanding, frameRAMKirby, frameRAMKirbyStill, frameRAMKirbyWalk2, frameRAMKirbyAttack, frameRAMlogo, frameRAMBackground2 **Inputs:** [18:0] read_address, Clk **Outputs:** data_Out **Description:** These modules read the text files for the sprites and return them based on the address. **Purpose:** All these modules are used for different sprites: map background, the character Mr. Game & Watch, the character Kirby, and the start screen logo, respectively. They dictate what the color should be based on where the pixels drawings the screen are.
Module: frame_counter **Inputs:** CLK, [6:0] count_in **Outputs:** [6:0] count_out **Description:** This is a counter for the number of frame cycles of VGA_VS. **Purpose:** This module was used to animate the characters. Based on the number of cycles that were passed, different sprites were drawn. This allowed for movement to be animated.

IV. Modifications in Software

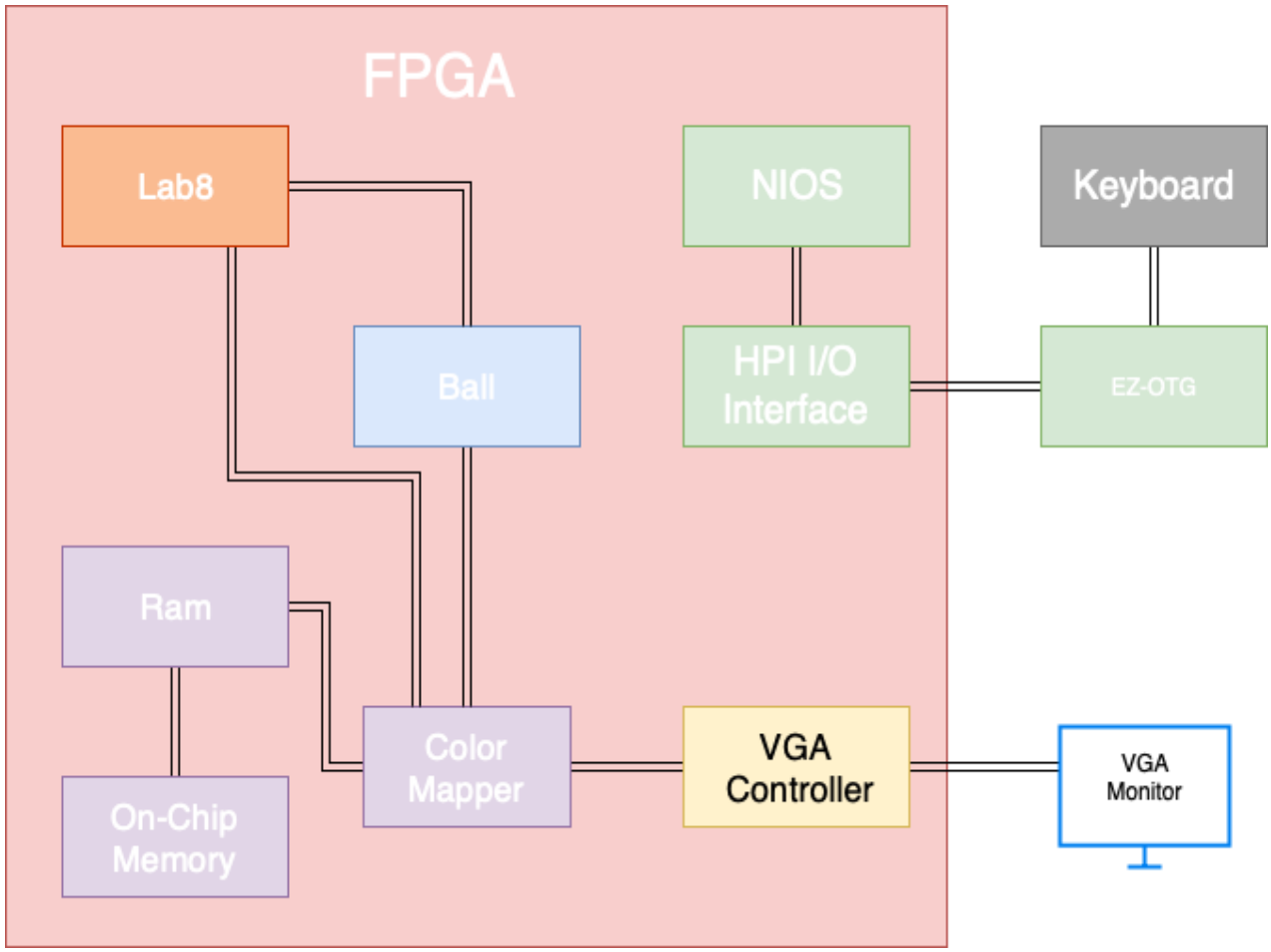
The USB code was the same as Lab 8, other than one key difference: since this is a multiplayer game, we implemented reading in multiple keycodes at the same time. We changed the PIO block to take in 32 bits so we can support upto four keystrokes, which would include fighting and attacking for both players at the same time.

We added the following code to read 4 keycodes simultaneously in main.c:

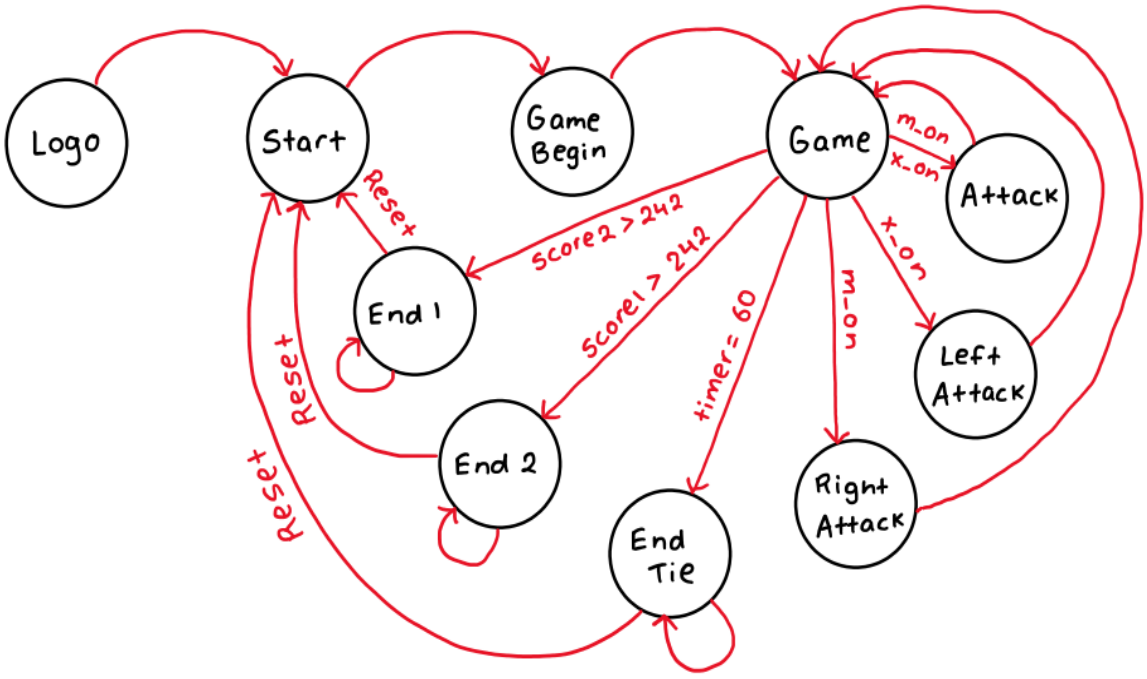
```
IO_write(HPI_ADDR,0x051e); //the address of byte 0~1
keycode = (IO_read(HPI_DATA)); //write first 2 bytes
IO_write(HPI_ADDR,0x0520); //the address of byte 2~3
keycode += (IO_read(HPI_DATA) << 16); //shift bytes 2 and 3 to write
to the beginning of keycode
printf("\nfirst four keycode values are %08x\n",keycode);
IOWR(KEYCODE_BASE, 0, keycode & 0xffff); //write to the PIO
```

V. Game Organization

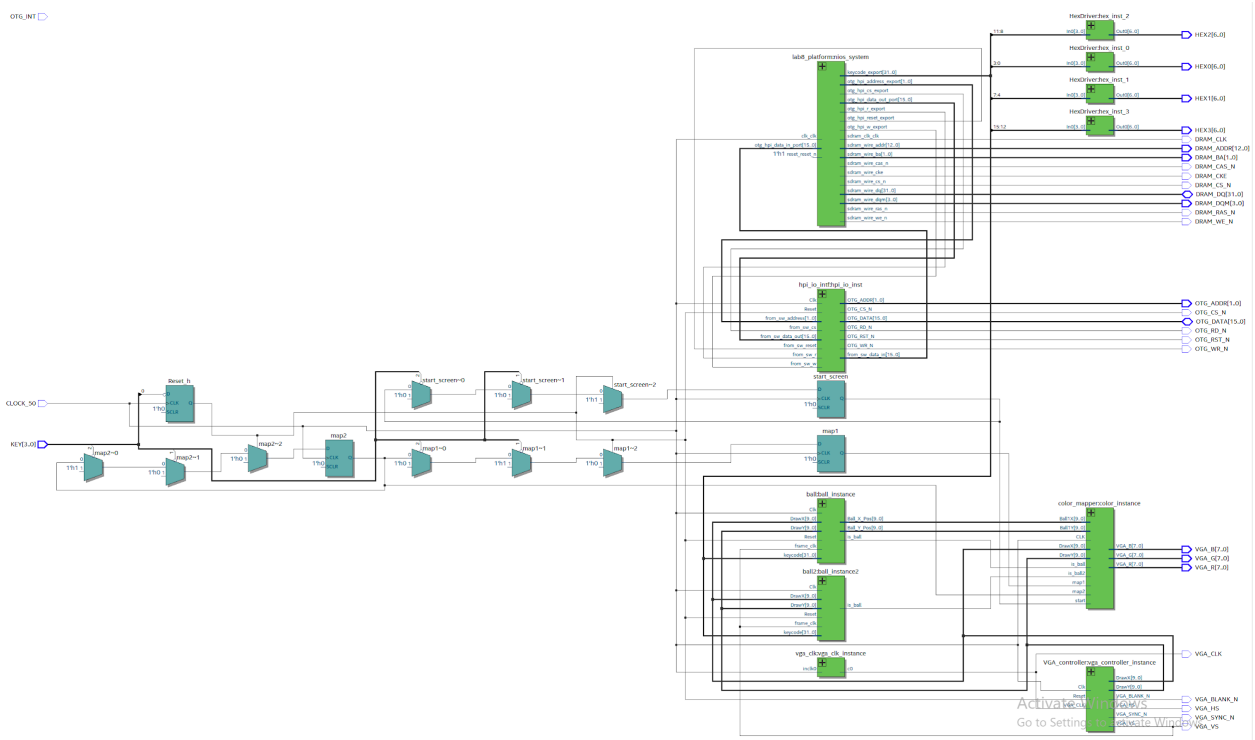
High level block diagram:



VI. State Transition Diagram

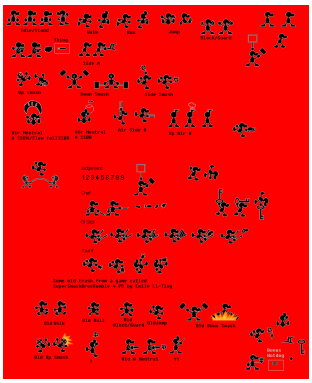


VII. Block Diagram



VIII. Sprites Description and Color Encoding

Sprites were used to animate the characters, generate the backgrounds, and display the logo screen. To generate sprites, images were processed using Rishi Thakkar's ECE 385 Helper tools. A total of 5 different color palettes were used, one for each character, one for each background, and one for the logo. The following sprite sheets were used for the characters:



The sprite data was stored within on chip memory, and accessed within the color mapper. Based on the pixel and the state of the FSM, the appropriate screen was drawn.

IX. Resource Usage

Resources	Usage
LUT	3,134
DSP	0
Memory (BRAM)	216
Flip-Flop	2,375
Frequency	141.36 MHz
Static Power	105.32 mW
Dynamic Power	0.82 mW
Total Power	187.74 mW

X. Conclusion

While this project was quite challenging and time consuming, it taught us a lot about System Verilog and hardware and working with the FPGA. By the end of the project, we managed to include all of our proposed functionality and even some advanced functionality. However, this project was very open ended and we could have and wanted to add more advanced functionality if given the time. One key thing we really wanted but didn't have the time to include was jumping and gravity.

This lab integrated everything we have learned in 385, from creating a high level design from the first three labs to actually implementing it in System Verilog from the last six labs and working with the VGA display from Lab 8. We started the project by taking the code from Lab 8 and modifying it. We started with including reading in several keycodes and having two balls instead of one, and moving both in any direction at the same time. We then moved to including additional keys as attack keys and changing the balls to actual characters from Super Smash Bros. with maps in the background using sprites, which we did research on and had to learn about. We then proceeded to calculate score and display a health bar for each character and also include sound on the starting screen of the game, which was another topic we had to extensively research.

This was the first project in our college careers where we have set the timeline and worked according to it, without being given assigned steps by course staff. It was also the first project we have done from scratch. It was a very enlightening experience to work at this project at our own rate and keep ourselves on task and on pace with the proposed timeline.

Overall, ECE 385 gave both of us a new perspective on what we want to do in the future as a career. We both extensively enjoyed the design and implementation process in hardware and

liked learning about interfacing between software and hardware. This course is a good combination of everything we learn in our majors and it is great to see the actual implementations and a physical end result of everything we have learned in the form of one of our favorite videos games.