TABLE OF CONTENTS

Introduction

In this final project, we created a two-player, 2D soccer game that uses bobble heads to move the ball around. We used SystemVerilog for the majority of the project, utilizing the Nios II software developer only to interface with the USB keyboard. Using simulated gravity, both the players and ball will have the ability to return to ground rather than going off into space. The game offers a start screen, is played until Patrick of Zuofu score 3 goals, then ends, offering the ability to replay the game. For points, we included: sprite images, a gravitational effect in the game, 4-key input detection, game start and end screens, collision detection, score, audio, as well as incorporating movement changes based on where objects collided. Our process began by using Rishi's provided scripts to put sprite images onto On-Chip memory. Once we had our images on the screen, we mapped them accordingly and gave the players movement and gravitational abilities using counters and state machines. We then moved on to collision detection and added movement, as well as incorporating score detection and created a state machine that enabled start and end screens. Finally, we added the ability to read 4 key presses from a USB keyboard by modifying our lab 8 as well as adding audio drivers given to us and using the provided files from ALTERA to add sound whenever a character jumped in the game. We attempted to have audio work every time the ball made contact with an object but could not implement it as it did not accept inputs made through logic, as it would force the input signal to ground.

Documentation of Hardware

Written Description of Game

The game is made up of several main parts, the sprite and color mappers, the movement and collision modules, as well as the peripherals such as gravity, audio, and the keyboard, and finally a master state machine that enabled the game. The gravity, audio, and keyboard peripherals will be described later on. The sprite and color mappers were created through a palette, which allowed us to keep the memory on chip. We would input the size and location of every sprite into a module, which would then decide what image would be printed to the screen in frame_decider.sv. It would run DrawX and DrawY through the entire screen, identifying where each sprite was supposed to be located using X,Y coordinates and its corresponding size. Once decided, it sent that images data to our palette module, which output the correct RGB values to be displayed on the VGA screen. This is how the visuals were created for the project.

Movement and collision were taken care of in 3 separate modules, one for the ball, Patrick, and Zuofu. These modules were ball.sv for the Patrick player, ball_ball.sv for the ball, and zuofu.sv for the Zuofu player.The player modules, Patrick and Zuofu, contained cases similar to that of the ball in Lab 8, where inputs from the keyboard allowed for movement, but we added diagonal movement if the user input signals designated for up and left for example. They also contained their own collision implementation for one another by inserting the others position, and if they were touching, would jump in the opposite direction. If the player was near their goal however, they would not move and the opponent would, to prevent phasing into the goal. The ball module contained the largest amount of collision implementations. It would check collision with the corner of goals, within goal, top of goal, and different collisions with Patrick and Zuofu. It checked this collision through detecting each sprites X and Y position, plus each sprites area, and if within the picture, would then asses how to move. If it was detected the ball sprite would touch Patrick or Zuofu, it would determine where. If the ball was above, it would then move up on the screen, and depending what side, would also launch off to the right or left. Top right and top left corners of the players would result in the expected motion of the ball. If the player trapped the ball beneath it and the ground, the ball would shoot out to the right or left to prevent phasing and to simulate loss of control. It also completed a similar tactic if the ball was stuck between the two players, however shooting up this time to simulate a contested ball. The ball also detected when it would touch the ground to reset its gravitational effect, and thus move up again on the screen. The ball would also start without any x motion at the start of the game, but once touched in a way that would move it to the right or left, would then remain in that motion and change directions if opposing contact was made, through a state machine that output a general x_move wire.

Written Description of "Gravity"

The gravity effect was used on the two players and the ball. We designed our jump for the players to be smaller as to create fewer bugs when interacting with the ball, i.e. avoiding phasing into the ball and creating random movement. So, the gravity was almost uniform for the players as their possible jump height was set to a max. We achieved their gravity through a state machine, but found that the state machine clock was too fast to implement full gravity, so we implemented states to declare if the player was in the air or on the ground, and so would be enabled after the jump button was pressed for the player. Once pressed, the state machine output the Y movement number as well as would wait to receive a "touch ground" signal from the player module if it detected the sprite was in contact with the ground image.

The ball gravity was implemented through a counter, that would reset if the ball was in contact with the ground, top of the players head, or the top of the goal. The value of the Y movement would decrease to 0 after a certain count, and then would slowly increment in the down movement as the counter increased, declaring higher Y changes in increasing intervals to simulate acceleration. This gravity was used anytime the ball was in free space.

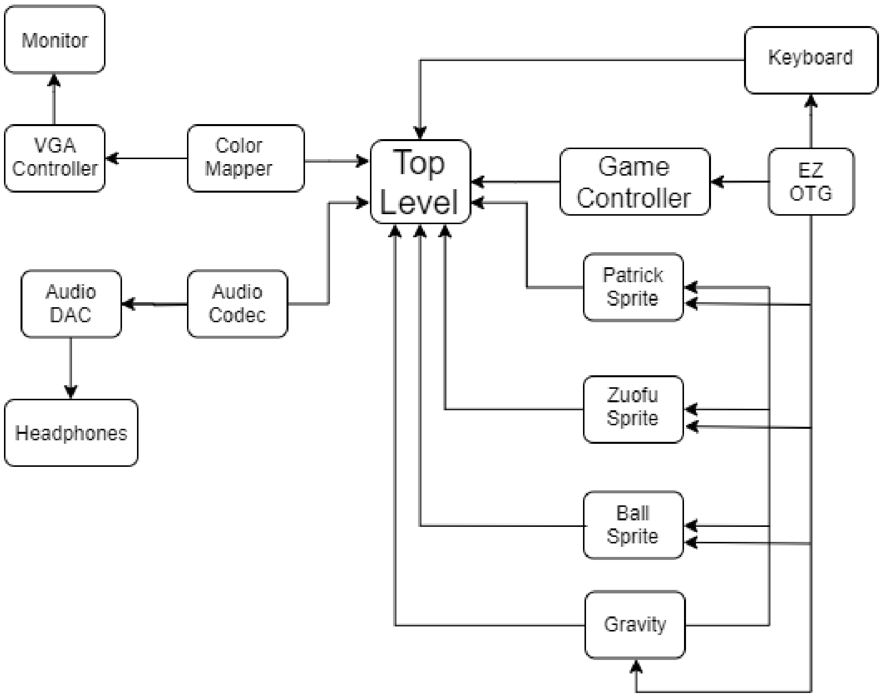
Written Description of Audio

Much of this source code was provided or summarized in the ALTERA DE2-115 disc, as we made modification to what input would be mapped to which sound. We mapped the sounds to be enabled by the designated 'up' keypress as well as 'w', so a sound would play every time the players would jump. We gave these values inside our top-level program, and then set every other audio wire to 0 within staff. We created a new SDC file to create a clock for the audio modules. The audio codec is the module that produces the sound based on the staff module which will select the sound that will be played.

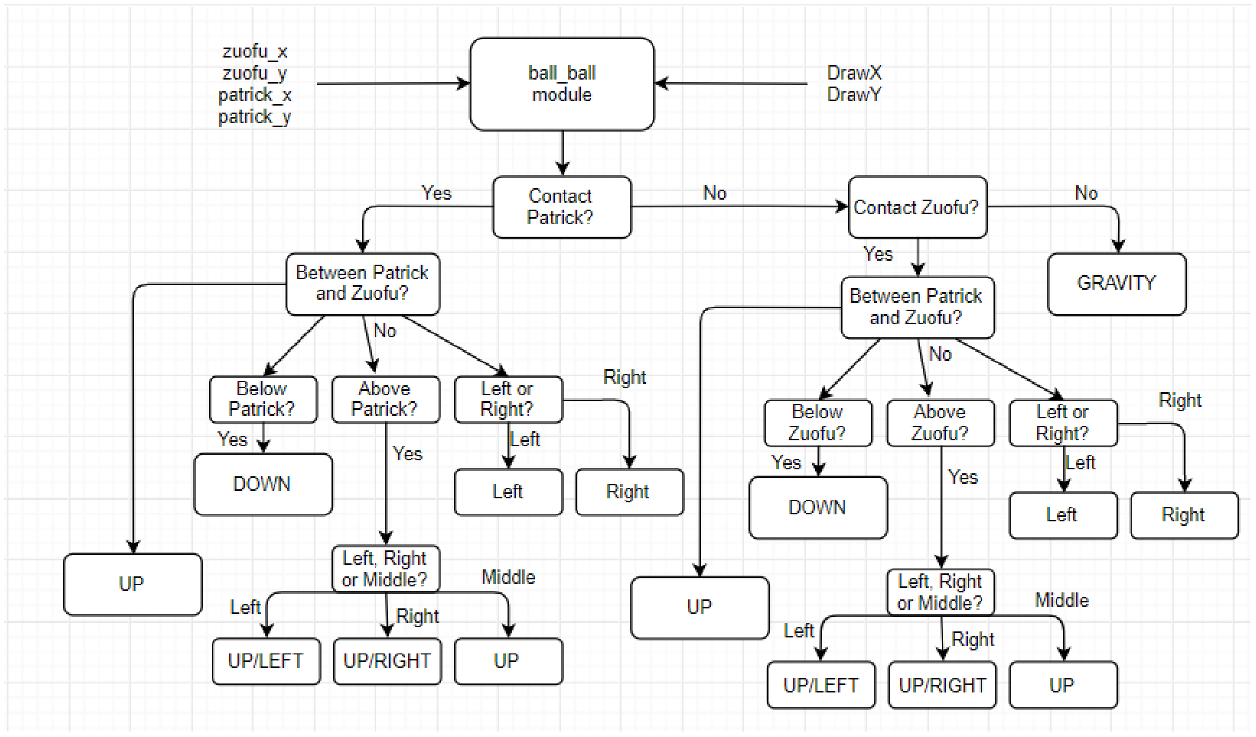
Written Description of Keyboard

There are 4 functions that we had to implement to read/write with the keyboard, as well as expanding the size of keycode given in Lab 8. USBRead reads the data stored in the CY7C67200's internal registers. USBWrite writes data to the CY7C67200's internal registers. IO_read is responsible for handling all the timing for the input signals for the HPI read cycle. It reads and return data from memory location specified by the address. IO_write is responsible for handling all of the timing of the input signals for the HPI write cycle. It writes data to the memory location specified by the address. Then, modifications from Lab 8 took place. In our Qsys, we increased the size of the PIO module from 8 to 32 bits, as well as increasing its size in our top-level file. This allowed us to read 4 different keystrokes at a time, and then used logic to map them to the movement of the players. We used logic to determine if any of the 4 key codes contained the correct input, and then declared the movement as on. For example, if the 'w' key was pressed, our program would determine if any of the 4 inputted bytes had the hex values x1A, and then declared up_on as 1, which would go into the Patrick module and be used to say that Patrick must jump.

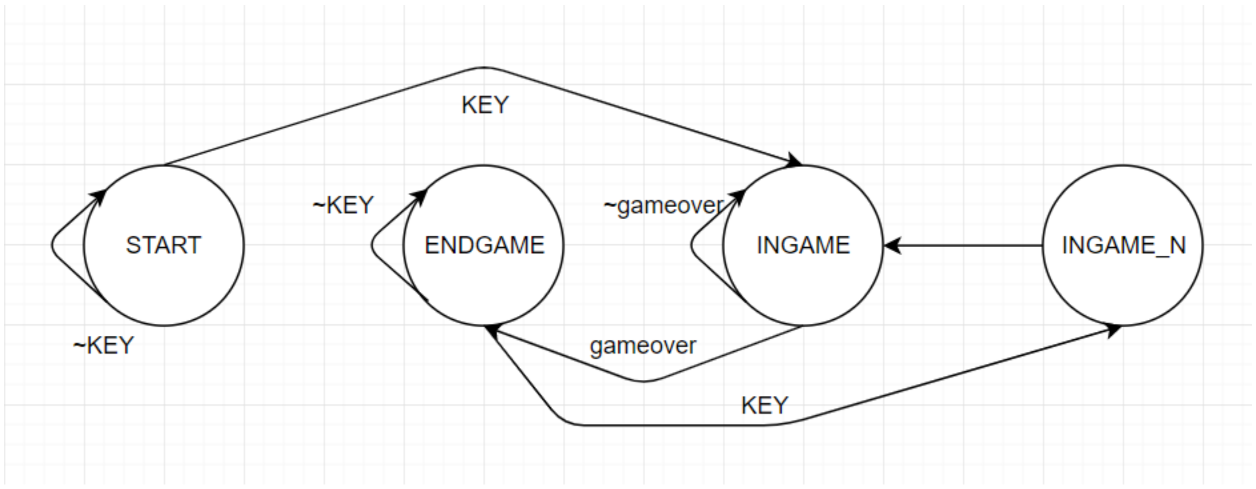
Top Block Diagram



Ball Movement Flow Chart



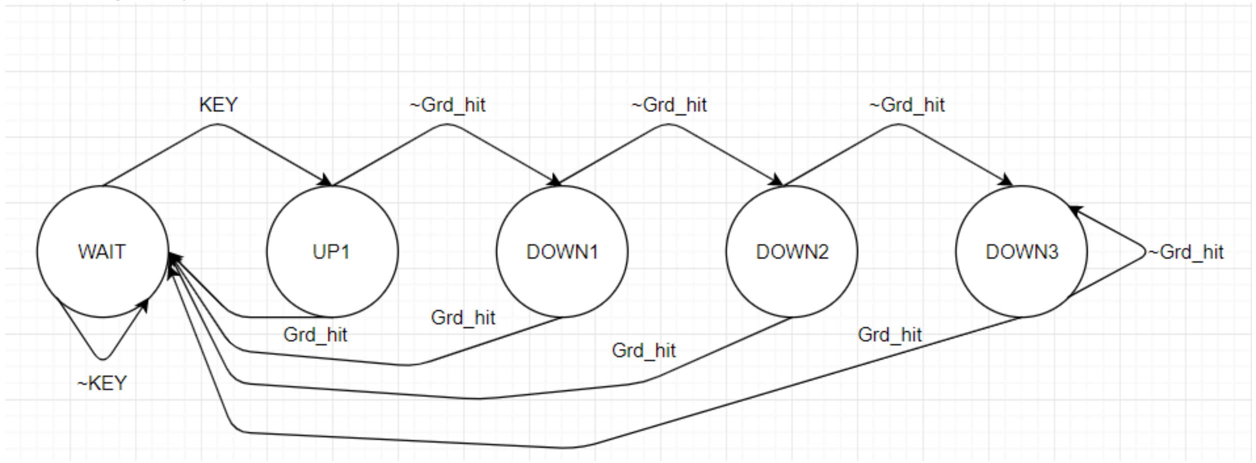
Game State Machine



This state machine controls the score, outputs to the screen, and keyboard inputs. While in START or ENDGAME, the keyboard is disabled and only the 'space' key is enabled. Once pressed, the game is launched to its natural state, without displaying the starting screen images or endgame images. During the game, this module also increments the score of Zuofu and Patrick if they score, and once one hits three, game over is enabled, ending the game. The end screen is displayed, declaring that the game can be played again. If played again, the state machine goes to INGAME_N to reset the game and then enter the regular game mode.

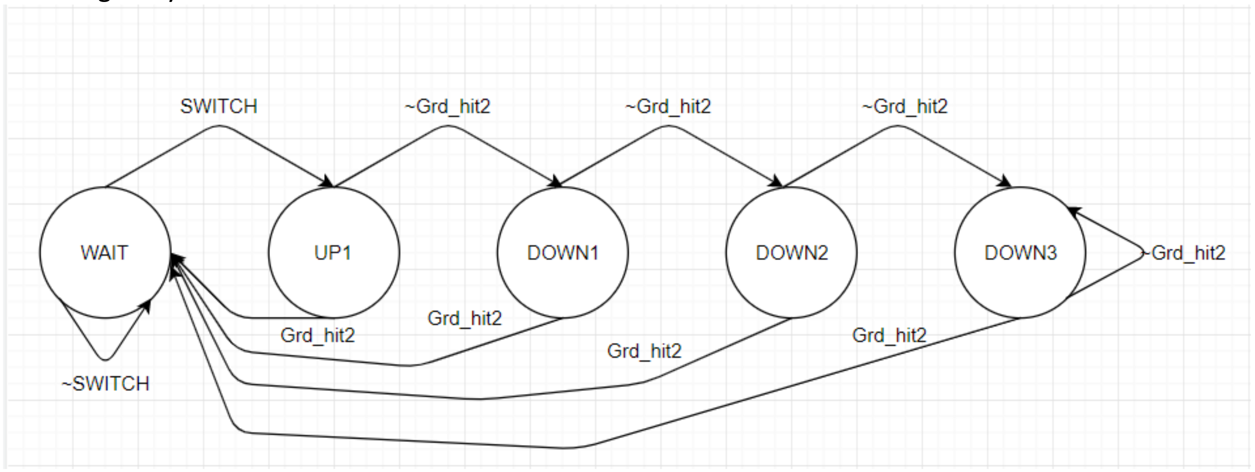
Gravity State Machines

Patrick's gravity state machine



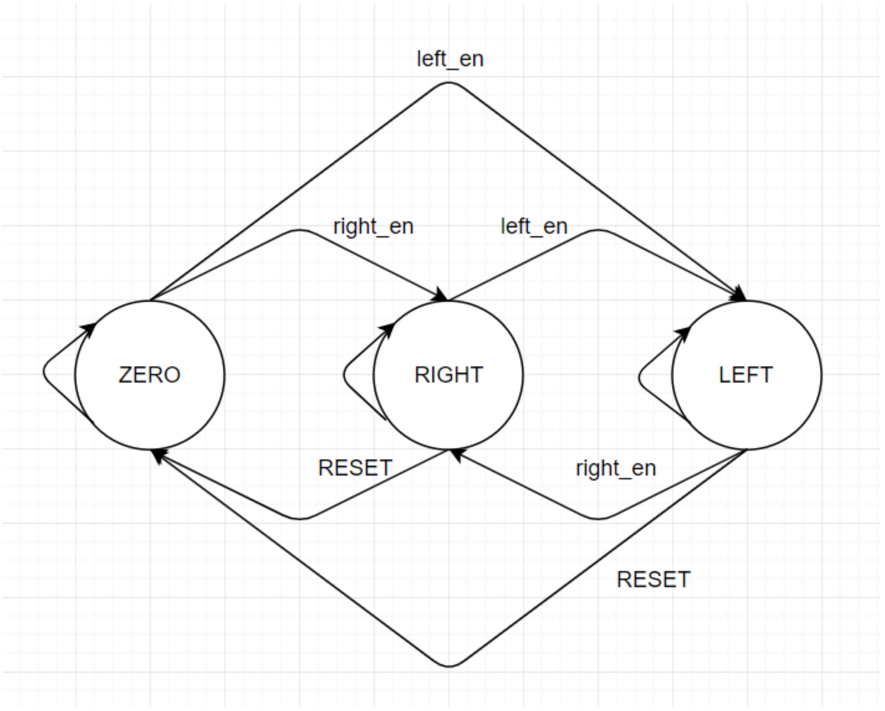
Each up and down had their set Y motion variables attached that would be input into Patrick if he was in a situation where he should feel gravity.

Zuofu’s gravity state machine



Each up and down had their set Y motion variables attached that would be input into Zuofu if he was in a situation where he should feel gravity.

X-Motion state machine



This state machine controls the x-direction of the ball. The ball module detects if it should move right or left and sends the left_en or right_en signal to the state machine. This machine then outputs a constant X value for the ball to move in.

Modules

Module: ball.sv
Input: Clk, Reset, frame_clk, Reset_New, up, left, right, [7:0] keycode, [9:0] DrawX, DrawY, Ball_Y_Move, zuofu_x, zuofu_y
Output: Grd_hit, [9:0] Ball_X_Pos, Ball_Y_Pos
Description: this module uses the current input of the Patrick player, and deciding whether its on the ground, in the air, or colliding with an object and the user key press to determine where it will move next
Purpose: this module determines the top left position of the Patrick player

Module: ball_ball.sv
Input: Clk, Reset, frame_clk, Reset_New, up, left, right, [7:0] keycode, [9:0] DrawX, DrawY, patrick_x, patrick_y, zuofu_x, zuofu_y, x_move, [10:0] Count
Output: grav_en, grav_reset, right_en, left_en, pure_grav, zuofu_goal, patrick_goal, [9:0] Ball_X_Pos, Ball_Y_Pos
Description: this module uses the current input of the ball, and deciding whether its on the ground, in the air, or colliding with an object to determine where it will move next
Purpose: this module determines the top left position of the ball

Module: Color_Mapper.sv
Input: CLK, Reset, game_start, zuofu_win, patrick_win, [2:0] patrick_score, zuofu_score, [9:0] DrawX, DrawY, patrick_x, patrick_y, zuofu_x, zuofu_y, ball_ball_x, ball_ball_y
Output: [4:0] is_right
Description: this module receives where DrawX and DrawY are, and based on existing locations of all the sprites, determines which sprite takes priority to be drawn
Purpose: determines what sprite will be enabled to be drawn to the screen in sprite.sv

Module: counter.sv
Input: CLK, CountEnable, pure_grav, RESET
Output: [10:0] Count
Description: this counter is a counter that caps at 1111111111, and resets whenever the RESET is high or pure_grav is enabled, signaling that something has forced the ball to go down
Purpose: this module controls the counter that controls the balls gravity, resetting counts or starting at certain values that correspond to the ball going up or down

Module: game_controller.sv
Input: CLK, Reset, patrick_goal, zuofu_goal, KEY, gameover,
Output: Reset_New, game_start, [2:0]patrick_score, zuofu_score)
Description: this module is a state machine that goes through the game states based on the input KEY, and if the scores have reached 3 to either player
Purpose: this module is a state machine that receives the goal inputs to increment the score as well as determine if the game is initiated or whether its on the start or end screen

Module: keycode_reader.sv
Input: [31:0] keycode
Output: w_on, a_on, d_on, up_on, right_on, left_on
Description: takes in all 32 bits of the keycode and checks if any of the bytes have the correct keycode assigned to moving the players, turns the output signal to high if true
Purpose: this module will detect if the keypress has been done and tell the player modules that they should move accordingly

Module: lab8.sv
Input: CLOCK_50, AUD_ADCDAT, [3:0] KEY, OTG_INT
Output: AUD_XCK, I2C_SCLK, AUD_DACDAT, VGA_CLK, VGA_SYNC_N, VGA_BLANK_N, VGA_VS, VGA_HS, OTG_CS_N, OTG_RD_N, OTG_WR_N, OTG_RST_N, DRAM_RAS_N, DRAM_CAS_N, DRAM_CKE, DRAM_WE_N, DRAM_CS_N, DRAM_CLK, [1:0] DRAM_BA, OTG_ADDR, [3:0] DRAM_DQM, [6:0] HEX0, HEX1, [7:0] VGA_R, VGA_G, VGA_B, [12:0] DRAM_ADDR
Inout: AUD_ADCLRCK, AUD_BCLK, AUD_DACLRCK, I2C_SDAT, [31:0] DRAM_DQ
Description: this module instantiates the soc system as well as the rest of our hardware
Purpose: it is the top-level module that connects the software data to the rest of the system, as well as acting as the hub for most of our internal signals

Module: player_grav.sv (grav_p, grav_z, grav_x)
Input: CLK, Reset, Hit, Grd_hit, KEY, right_en, left_en
Output: [9:0] Ball_Y_Move
Description: it is a state machine that takes in KEY, Grd_hit to determine if it needs to go up, else outputs a down moving Y output, or X output for grav_x based on right/left_en
Purpose: determines Patrick's or Zuofu's gravity and Y movement, as well as the ball's X oriented motion

Module: sprites.sv
Input: [4:0] is_right, [9:0] DrawX,
Output: [7:0] VGA_R, VGA_G, VGA_B
Description: this module receives a location from is_right and then outputs RGB values that that value of is_right is assigned to
Purpose: this is how the color of the pixel that DrawX, DrawY are on is decided

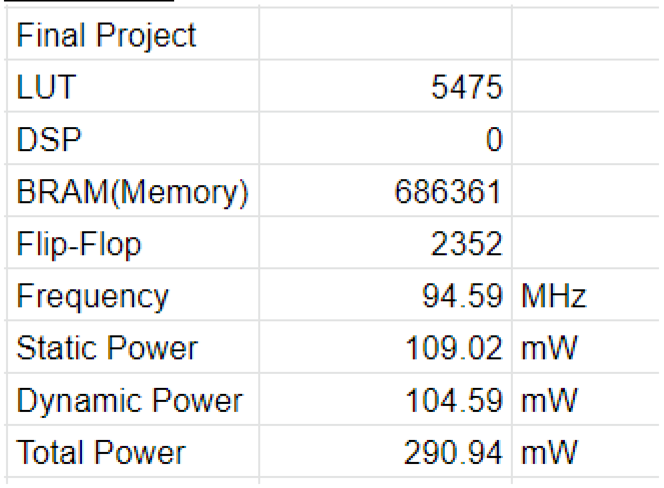
Module: zuofu.sv
Input: Clk, Reset, frame_clk, Reset_New, up, left, right, [7:0] keycode, [9:0] DrawX, DrawY, Zuofu_Y_Move, patrick_x, patrick_y
Output: Grd_hit2, [9:0] Ball_X_Pos, Ball_Y_Pos
Description: this module uses the current input of the Zuofu player, and deciding whether its on the ground, in the air, or colliding with an object and the user key press to determine where it will move next
Purpose: this module determines the top left position of the Zuofu player

Module: lab8_soc.v
Input: clk_clk, reset_reset_n, [15:0] otg_hpi_data_in_port,
Output: otg_hpi_r_export, otg_hpi_reset_export, otg_hpi_w_export, sdram_wire_cas_n, sdram_wire_cke, sdram_wire_cs_n, sdram_wire_ras_n, sdram_wire_we_n, [1:0] sdram_wire_ba, sdram_wire_dqm, otg_hpi_address_export, [7:0] keycode_export, [12:0] sdram_wire_addr, [15:0] otg_hpi_data_out_port, [31:0] aes_export_export_data
Inout: [15:0] sdram_wire_dq
Description: this module instantiates the wires between the JTAG-UART, the NIOS II processor, tand the EZ-OTG that enables keypresses to be read, as well as communication between hardware and software
Purpose: This module is used to connect all the individual SoC components of our system

Module: staff.v
Input: [7:0] scan_code1, scan_code2, scan_code3, scan_code4,
Output: sound_off1, sound_off2, sound_off3, sound_off4, [15:0] sound1, sound2, sound3, sound4
Description: maps input scancode from the keypress and assigns high/low values to the wires that communicate which waveform will be played
Purpose: to map what input will be pushed into the audio CODEC

Additional Data

Game Data



Final Project		
LUT	5475	
DSP	0	
BRAM(Memory)	686361	
Flip-Flop	2352	
Frequency	94.59	MHz
Static Power	109.02	mW
Dynamic Power	104.59	mW
Total Power	290.94	mW

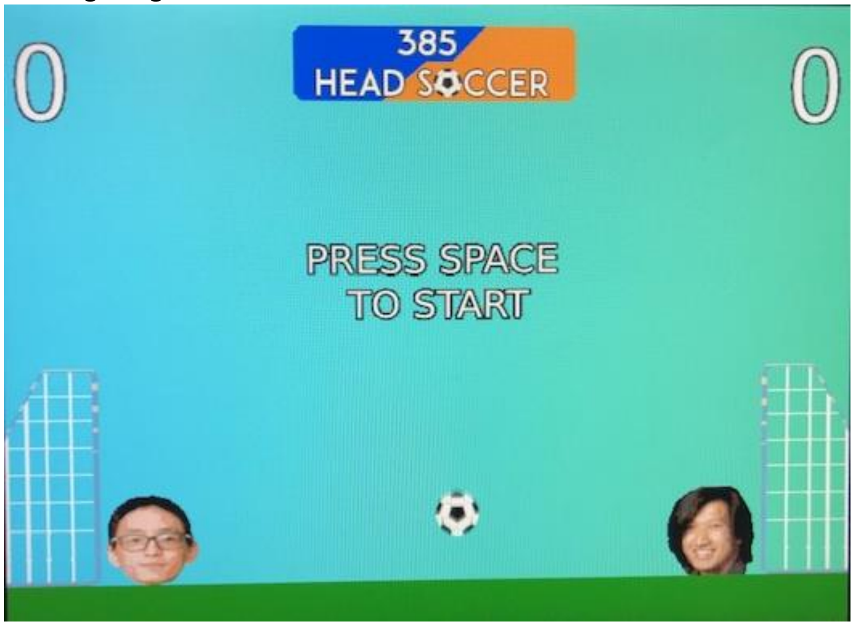
Conclusion

Overall, creating the head soccer ball was an intensive and challenging task to figure out, but then did present itself as a repetitive implementation. We built the program in stages, often working on a single element to completeness, such as the Patrick model, and then use that model and copy it onto other similar modules, such as the Zuofu model. It was difficult implementing various functions due to the restrictive nature of coding hardware and the always_comb blocks. Dealing with two different clocks when it came to the system and frame clock also made interaction a challenge. The project overall gave us great insight into the amount of work and number of variables that go into such achievements. We were able to meet all of our expectations but were unable to implement level selection or a “kick” motion that we had set as additional reach goals. Overall, we were pleased with the outcome of our project.

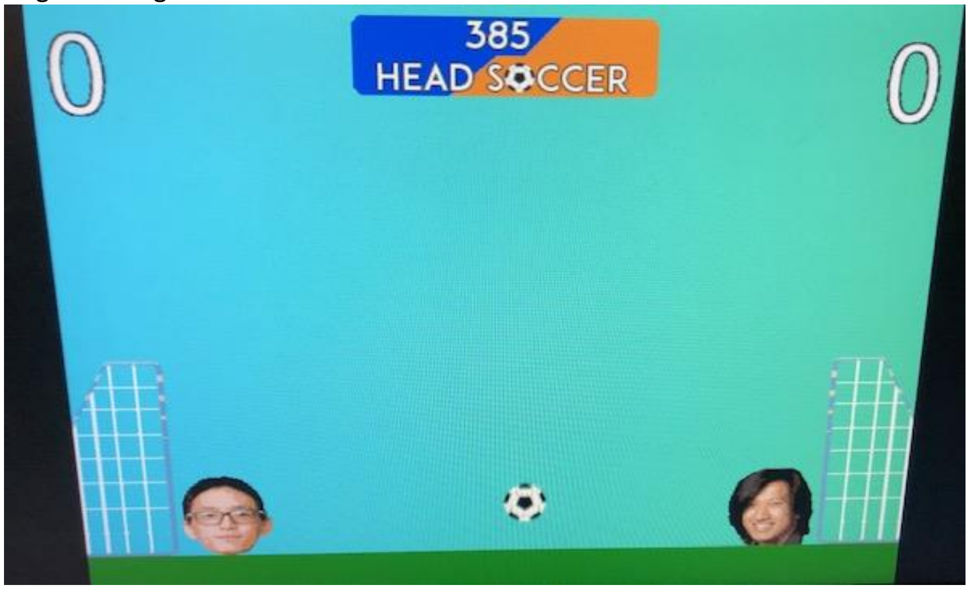
Appendix

Game Images

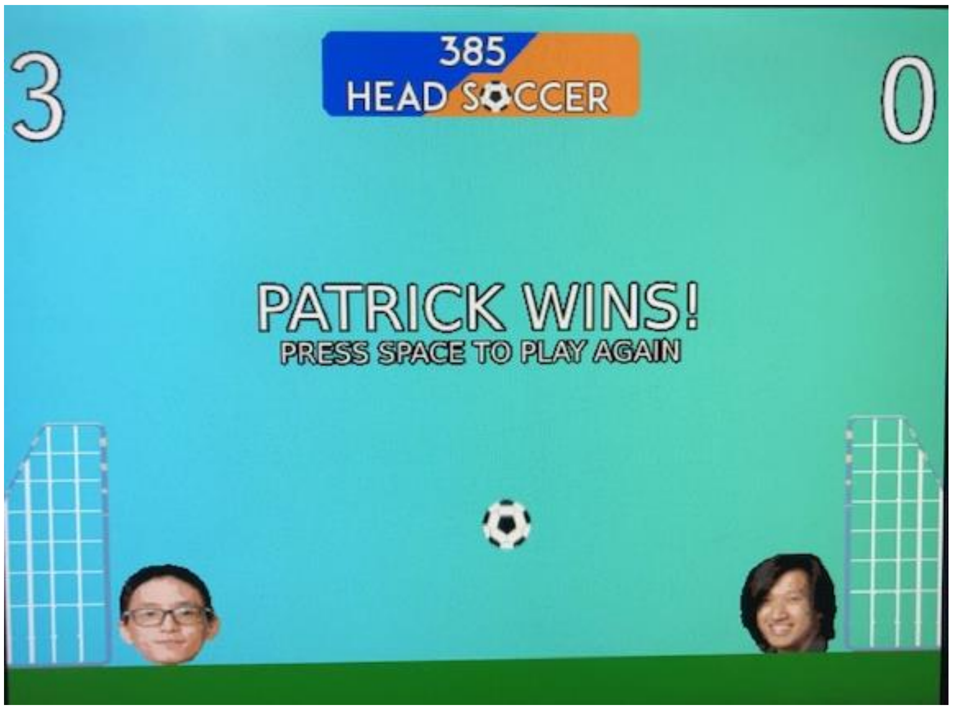
Starting Image



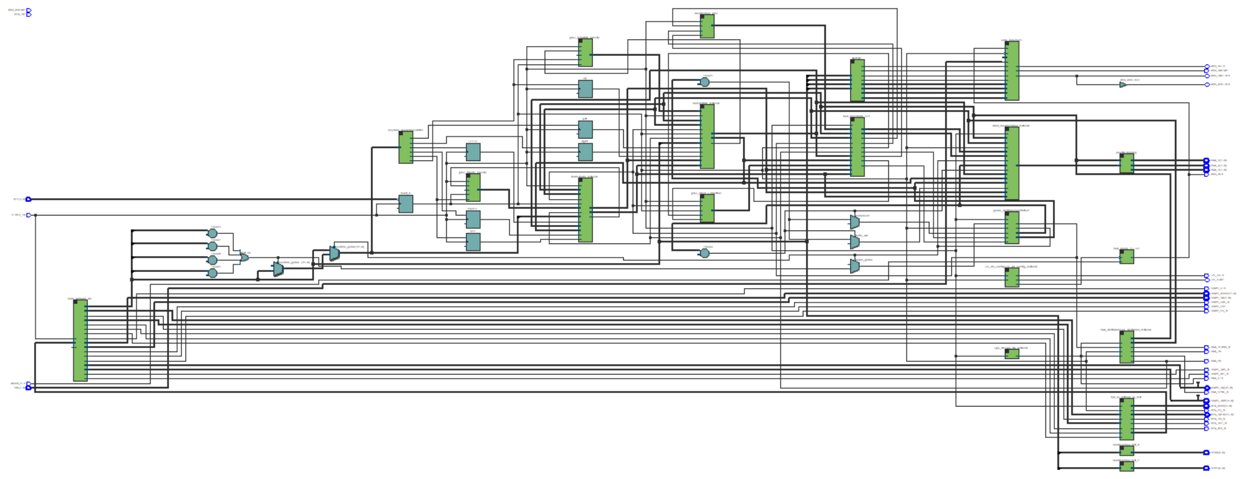
In-game Image



Endgame Image



RTL



Citations
Imaging scripts : https://github.com/Atrifex/ECE385-HelperTools
Audio driver and starting text : ALTERA DE2-115 educational board disc
Palette mapping software : https://www.piskelapp.com/