ECE 385

Spring 2018

Final Project

2-Player Fighting Game

Fite 4 The A

Omar Taha & Taha Anwar

ABL/ Thursday 8am - 11am

Vibhakar Vemulapati

Introduction

We implemented a 2-Player fighting game for our final project in ECE 385. The objective of our game was for two players to battle for an A in ECE385. Essentially, both players can run left or right and can also punch using certain keys on the keyboard. A combo was also implemented where one could output a fireball by pressing certain keys in succession. Both players have a health bar and when a player is hit, his health decreases. Once the health of one player reaches the end, he has lost. We decided to code our project entirely in System Verilog and work off our lab8 code that we implemented. We initially intended to make our game logic in C code but we did not want to deal with the many problems that come with interfacing between software and hardware. Although this increased the amount of code we had to write, we deemed it to be the best option. As for the user interface we decided to proceed with the PS/2 keyboard instead of the USB keyboard. We chose this so we do not have to deal with the NIOS processor which handled the USB protocol. Since PS/2 is interrupt driven we solely had to include the given driver in our project. There were certain aspects we had to keep track off in order to get the keypress to work as desired. These modifications will be explained later in the report. In order to implement our characters we used a sprite table which we converted from png to txt using Rishi's helper tools. His tools simply used a python script to convert any png into a txt file. We put these sprites in On Chip Memory and accessed them by reading into the addresses. More about this and other other details will be explained below. Some overall features we included in our project were: start screen, end screen, palitized sprites, bitmapped font sprites, scorekeeping, funny title, advanced wall boundaries, combo hits, basic sprites and others.

Written Description

❏ Written description of project

In this project we used the PS/2 keyboard. The keyboard outputs both the keyCode and the press signals to our keypress module. The keypress module then sets the correct data in order to support simultaneous key presses of the keyboard. From there the keypress data goes to both the Player and Combo modules. The Player and Combo modules control the positions of the objects on the screen based off which keys are being pressed. In specific, each player's coordinates are outputted to the Color Mapper, Collision Detection, and Combo Modules. The combo module takes the player coordinates in order to handle the combo collision detection and outputs a signal to the Player modules to signify that the player has been hit. The combo module also outputs the coordinates of the fireball to the colormapper. The Color Mapper is where all of the screen work takes place. Using the DrawX, DrawY, and object coordinates, the color mapper will set the proper ROM read address. This read address is set to the ROM containing all the sprites of the game. The ROM then outputs the data of the sprite back to the color mapper. The Color Mapper then sets the RBG output values based off of the palletized data. Finally, the VGA_Controller takes these RBG values and colors the screen accordingly. So this is a very extensive process that each signal travels before the final colors of the game are on the screen, but with the power of technology it is incredibly fast.

❏ Each action

There are three basic possible actions each player may make; Left, Right, Punch. Each move will cause a series of different sprites to cycle through as if the person is moving. Below I will describe exactly the process of each motion.

The Left or Right control keys cause for the motion and direction of the player to be set so long he is walking. The direction of the player is important as it is sent to the color mapper in order to tell whether or not the player should be drawn upright or flipped horizontally. The walking phase in total cycles through 9 sprites. This is a bit more than usual, however we chose to implement it this way in order to have the most realistic walk as possible. Meanwhile, the punching action cycles through 4 sprites. The punch is given preference over walking.

❑ Sprites:

Sprites are an essentially aspect to any project as graphics are the first thing you notice in a game. In our project we opted to have a brighter graphics game such as Mario style instead of a traditional dark fighting game such as street fighter. Our main sprite sheet contains 18 stick fighters. I have attached all sprites in their respective sections below. We used photoshop to adjust the original sprite sheet to account for even spaces in order to make each individual sprite more accessible. We also had sprites for the Mario pipes as well as the combo fireball. However, our largest sprite was the start screen which was a full 640x480 pixels. All these sprites were put onto On Chip Memory with a ram module using the readmemh synthesis tool.

An important task that allowed us to save space and fit all of these sprites onto On Chip Memory was the process of palletizing all of the sprites. We were able to select specific colors using Piskel.com and then use Rishi's ECE385 Tool to convert the sprites to a text file which contained the palletized RGB colors.

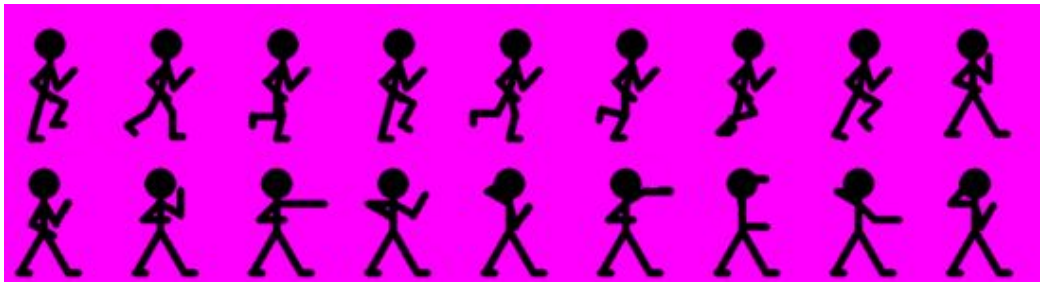
For our end screen we used font sprites. We had a end game signal in each of the player modules that signifies to the color mapper to color the "PlayerX Wins" end game message screen (X being either 1 or 2).

❑KeyPress

Because our game is a multiplayer game we struggled with managing the key presses from one keyboard. Players can be hitting multiple keys at a time, but the game is expected to be able tell exactly what is being held when. For example, Player1 may be moving left using keyCode (8h'1c), but when Player2 decides to punch with keyCode (8'h70) Player1's motion should not be affected by what key Player2 is pushing. To fix this we were able to use the fact that the PS/2 outputs the keyCode when the key is pressed **AND** released. We then were able to have a 8 bit data that is keeping track of all the keys being pressed by updating with the press value being outputted by the PS/2 when its keyCode is outputted. By doing this we were able to support 2-Player inputs from different keys at the same time.

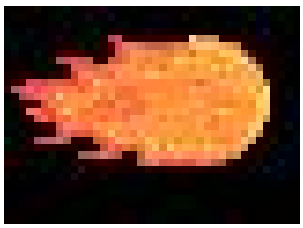
❏ Players

Our game is a multiplayer game. This means that there at at least 2 players on the screen at once. Each player module handles the motion and position of each player. There we also account for the health of each player which is then displayed on the top of the screen. Here we also handle the actions of each player. For example, if the “A” key is pressed for player1, the player will move left. As a result, this is also where the shifting of sprites for each action occurs. Below is the sprite sheet for the players.



❏ Combos

The more advanced attack mode of each player is the combo. A specific sequence of keyCodes will set off a fireball coming from the player and aims to damage the enemy player much more than a single punch does. In fact, each combo attack hit deals 15 damage, while each punch deals 2 damage. The combo sequence is Left, Right, Up, Punch, Direction to send the fireball. Once the combo is activated it starts from the players position and travels in the direction based on the last keyCode of the combo sequence. Once a combo is activated another combo cannot be activated from the same player until the original combo has been deactivated. There are 3 scenarios in which the combo is deactivated; combo hits the enemy player in which damage is dealt, combo hits Mario pipes (barrier) in which no damage is dealt, and combo is cancelled out with combo from enemy player in which no damage is dealt. Below is sprite for the combo (fireball) as well the combo in gameplay.



❏ Boundaries/Portals

In the early development of our game, we had opted for a hard wall boundaries, meaning the left and right side of the screen was the end of the game screen and you cannot go through. However we experienced a weird bug in that the players were able to go through the left side of the screen and come back in from the right. After we fixed the bug, we decided that being able to travel across the screen in more ways than one was better for game play. We added the Mario Pipes on either side and implement it as the Mario portals; in from one pipe leads out the other. Because the screen size is so small compared to the entirety of the game, this helped in being able to dodge your opponent more effectively. This functionality is handled in each of the player's module. Below is the sprite used for the Pipes.

❏ Collision Detection

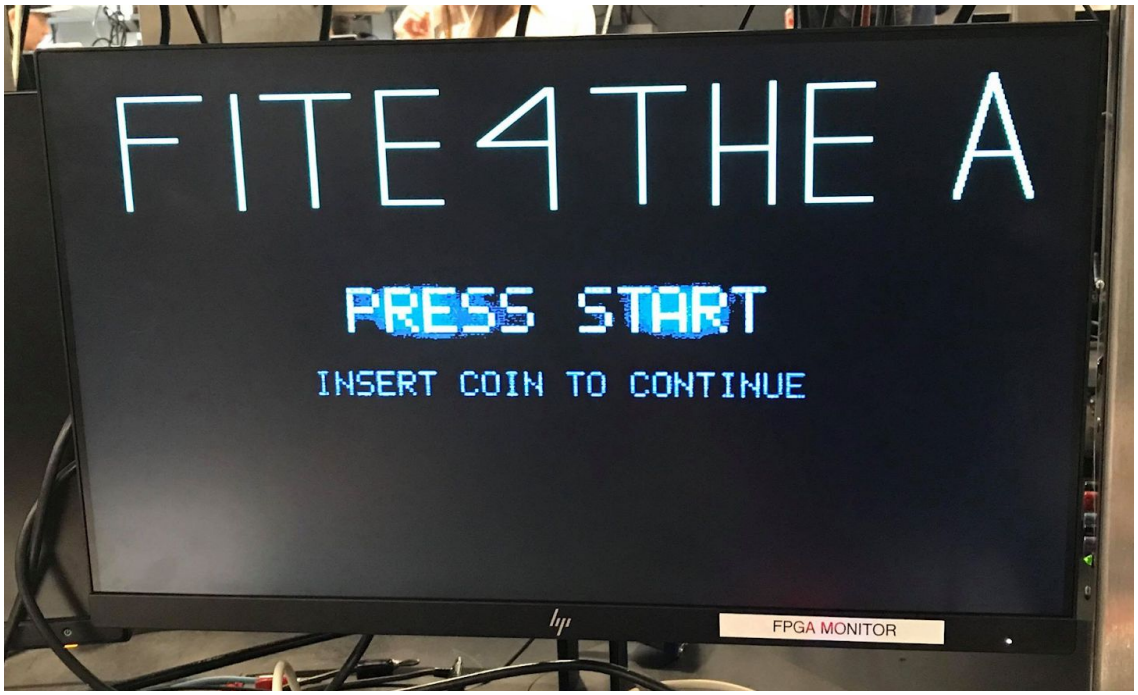
Collision detection is the most important part of any fighting game. Without collision detection we cannot tell whether or not a punch or attack was landed, if so, who wins the fighting game if players cannot fight each other? The difficulty of the collision detection is the amount of cases in which collisions occur. In particular there are 8 cases that count as a hit. I won't list each one as that will be too extensive, however I will say that each case depends on not only the positions of each player, but also their directions. This is because we must also account for the white space in the sprites as the positions may line up, but the background color of the sprite may make it seem as the player is punching *close to* the enemy player but not directly hitting him. Below I have included a photo showing a collision happening as well as a decreased health bar.

❏ Health

Each Player begins with 100 health, and a single punch deals 2 damage while a combo attack deals 15 damage. The health bars displayed on the top of the screen serve as score keepers. The health of each player is also important in ending the game and displaying the victor of the two players. Below I have included a photo showing the health bar.

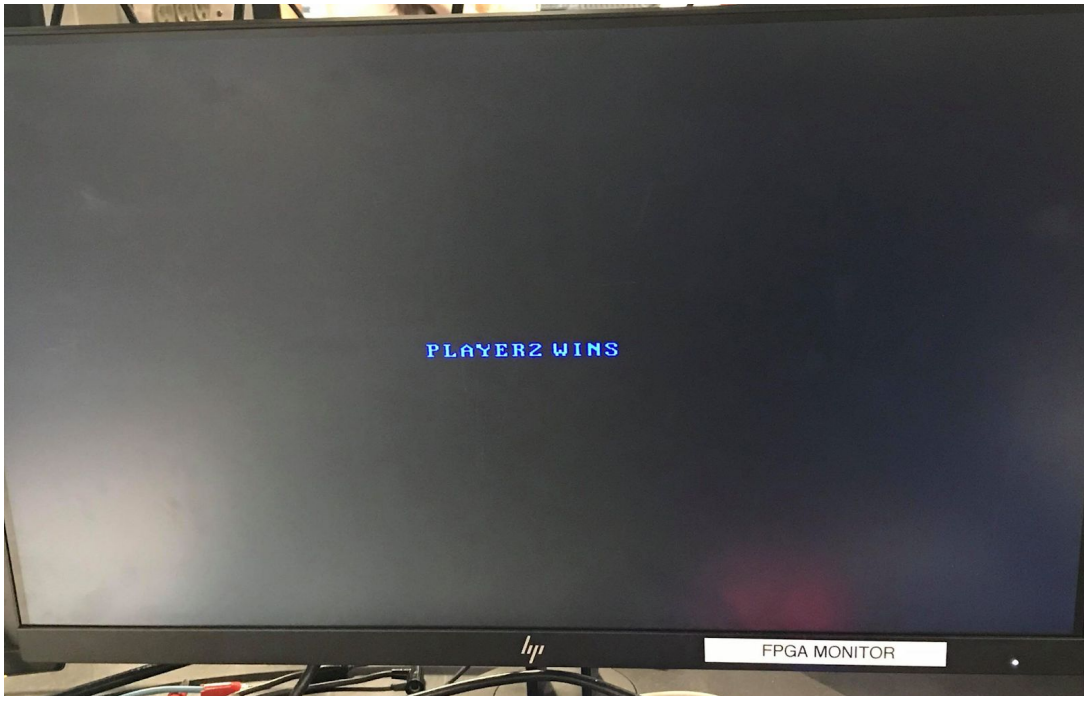
❑Start Screen

By default, we start at the start screen and must hit Enter on the keyboard to continue to the game. On the start screen we included an objective “Fite 4 The A” because a little humor never hurts. I have included an image showing the start screen.

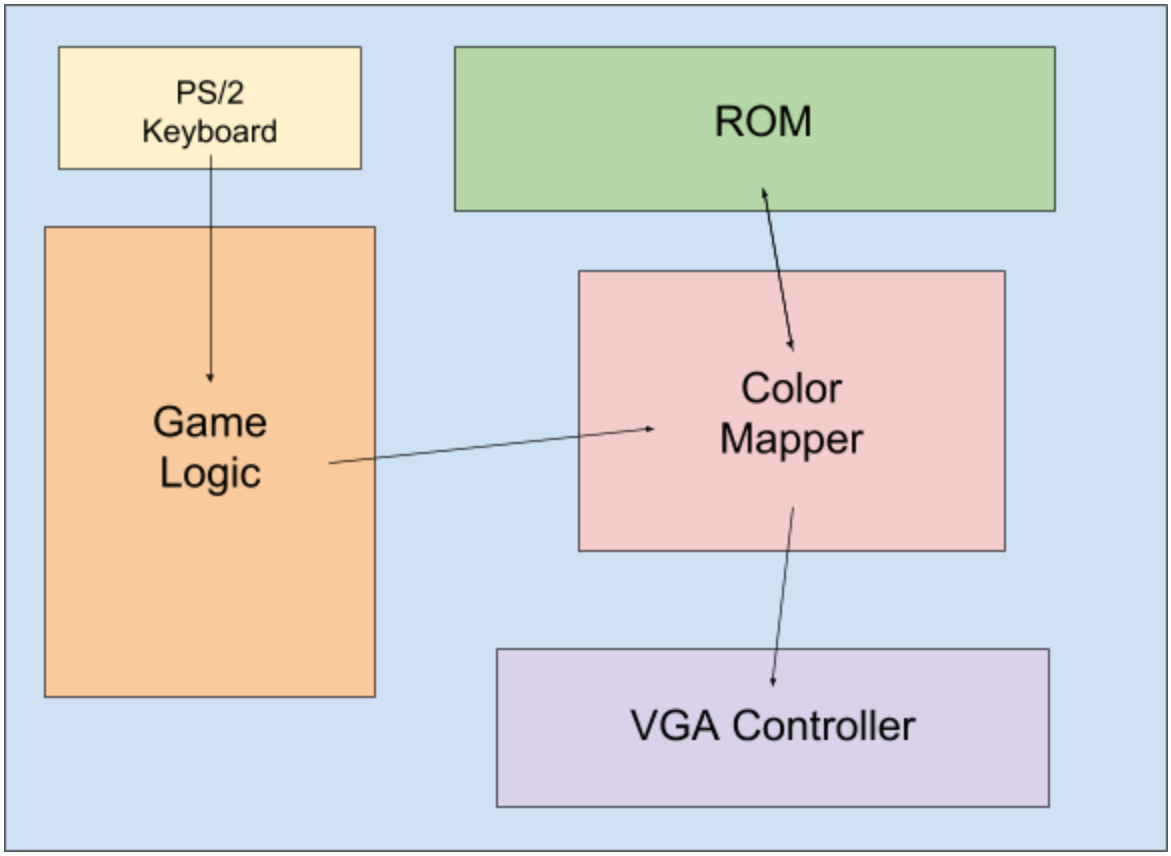


❑End Screen

When the health of a player goes to 0, the game ends and displays the victor, as it should. Unlike the start screen, the end screen is not a sprite on On Chip Memory. The End Screen is created using font bitmapped sprites. We preferred to do this way for the end screen to save memory and use a variety of sprite designs in our game. I have included a photo showing the End Screen when Player2 Wins.



❏ Block diagram



❏ Module Descriptions

Module: player1
Inputs: frame_clk, Reset, Clk, press, hit1, hit2, draw_combo2, combo_hit1, [7:0] keycode, [7:0] keypress, [9:0] DrawX, DrawY,
Outputs:
is_player1, p2_won,
[9:0] p1x, p1y, action1, direction1, health1
Description: This module contains many variables to keep track of what is happening. The action variable keeps track of the current motion/sprite that the player has based off the kecodes being pressed. The direction variable keeps track of whether the player is facing right or left. This will allow us to flip the sprite we use whenever the direction is different from the standard. The health variable keeps track of the players health and decrements it when player 2 lands a hit on player one. This collision detection is done in a different module.
Purpose: The purpose of this module is to compute all of the attributes/actions that player1 is doing. Such as moving, punching, keeping track of his health, updating his position and motion, selecting whether it is the correct time and more.

Module: player2
Inputs: frame_clk, Reset, Clk, press, hit1, hit2, draw_combo1, combo_hit2, [7:0] keycode, [7:0] keypress, [9:0] DrawX, DrawY,
Outputs:
is_player2, p1_won,
[9:0] p2x, p2y, action2, direction2, health2
Description: This module contains many variables to keep track of what is happening. The action variable keeps track of the current motion/sprite that the player has based off the kecodes being pressed. The direction variable keeps track of whether the player is facing right or left. This will allow us to flip the sprite we use whenever the direction is different from the standard. The health variable keeps track of the players health and decrements it when player 1 ands a hit on player 2. This collision detection is done in a different module.
Purpose: The purpose of this module is to compute all of the attributes/actions that player2 is doing. Such as moving, punching, keeping track of his health, updating his position and motion, selecting whether it is the correct time and more.

Module: Frame_control
Inputs: [9:0] p1x, p1y, p2x, p2y, action1, action2, direction1, direction2,
Outputs: hit1, hit2
Description: This module is the collision detection module which takes in as inputs both player1 and player 2's actions, positions, and directions. It outputs two variables indicating whether a hit was successful by player 1 or player 2.
Purpose: The purpose of this module is to determine if player1 hit player2 and vice versa. This is one of the crucial modules in our project since it detects the collisions between the two players.

Module: Start
Inputs: Clk, Reset,
[7:0] keyCode,
Outputs: is_start
Description: This module simply outputs and sets the is_start variable to 1 when the enter key is pressed.
Purpose: This module takes care of our start screen. We set the start screen sprite's address in color mapper and display it once reset is hit. Then once enter is hit we continue to the game.

Module: Combo1
Inputs:
frame_clk, Clk, Reset, make_combo2,
[7:0] keypress,
[9:0] p2x, p2y, p1x, p1y, player_direction, DrawX, DrawY, combo2_x, combo2_y,
Outputs:
[9:0] combo_x, combo_y, combo_direction
draw_combo, is_combo, combo_hit2
Description: This module controls the functionality of player1's combo. It uses the position, direction, and keypress of player1 in order to output the combo output's (fireball's) location and motion. It also considers when the fireball hits the other player, hits a boundary and when it hits player 2's fireball. The combination pattern needed to perform the fireball attack is left, right, up, punch, then either left or right depending on which direction you want the fireball to go in.
Purpose: The purpose of this module is to check whether a combination sequence of key presses of player1 is met and then to display the fireball (combo attack) in the appropriate location.

Module: Combo2
Inputs:
frame_clk, Clk, Reset, make_combo1,
[7:0] keypress,
[9:0] p2x, p2y, p1x, p1y, player_direction, DrawX, DrawY, combo1_x, combo1_y,
Outputs:
[9:0] combo_x, combo_y, combo_direction
draw_combo, is_combo, combo_hit1
Description: This module controls the functionality of player2's combo. It uses the position, direction, and keypress of player2 in order to output the combo output's (fireball's) location and motion. It also considers when the fireball hits the other player, hits a boundary and when it hits player 1's fireball. The combination pattern needed to perform the fireball attack is left, right, up, punch, then either left or right depending on which direction you want the fireball to go in.
Purpose: The purpose of this module is to check whether a combination sequence of key presses of player2 is met and then to display the fireball (combo attack) in the appropriate location.

Module: keyboard.sv
Inputs: Clk, psClk, psData, reset,
Outputs: [7:0] keyCode, press
Description: This is the PS/2 keyboard interface driver which allows us to use the keyboard in our game. It outputs the keycode being pressed and a press variable which is 1 when a key has been pressed and is 0 when a key is released.
Purpose: The purpose is to allows us to use a keyboard for our game.

Module: Dreg
Inputs: Clk, Load, Reset, D,
Outputs: Q
Description: 1 bit register that is used called in keyboard.sv
Purpose: To store a value (a flip flop)

Module: reg_11
Inputs: Clk, Reset, Shift_In, Load, Shift_En, [10:0] D
Outputs: Shift_Out, [10:0] Data_Out
Description: Is an 11 bit shift register which is called in keyboard.sv
Purpose: To either do a parallel load, hold data, or shift data out.

Module: keypress
Inputs: [7:0] keycode, press, Clk,
Outputs: [7:0] keypress
Description: Takes in the keycode and press variables that we receive from keyboard.sv and outputs an array which is updated with the press variable. Each element in the array corresponds to a key that we are using.
Purpose: The main purpose of this module is to allow us to deal with multiple keypresses.

Module: frameROM
Inputs: [18:0] read_address1, read_address2, start_address, pipe_address, combo1_address, combo2_address,
Clk
Outputs: [3:0] start_data, pipe_data, combo1_data, combo2_data,
[23:0] data_Out1, data_Out2
Description: This module takes as inputs the addresses for different sprites and then outputs the data corresponding to those addresses.
Purpose: The purpose of this module is to store our sprite txt files onto On chip memory and allows us to read data for different sprites based on the addresses inputted.

Module: font_rom
Inputs: [10:0] addr
Outputs: [7:0] data
Description: module that holds arrays for many different font sprites
Purpose: We used this file in order to display the end game screen using font sprites

Module: VGA_controller.sv

Inputs: Clk, Reset, VGA_CLK,

Outputs: [9:0] DrawX, [9:0] DrawY, VGA_BLANK_N, VGA_SYNC_N, VGA_VS, VGA_HS

Description: Sets vertical and horizontal counter, uses the horizontal and vertical pulse and VGA_clk

Purpose: To keep track of location on VGA and to deal with setting the appropriate condition codes in order to display the desired pixel.

Module: Color_Mapper.sv

Inputs: is_player1, is_player2, hit1, hit2, p1_won, p2_won, is_start, is_combo1, is_combo2, draw_combo1, draw_combo2,

[9:0] DrawX, DrawY, p1_h, p1_w, p1x, p1y, p2x, p2y, health1, health2, combo1_x, combo1_y, combo2_x, combo2_y,

[23:0] data1, data2,

[9:0] action1, action2, direction1, direction2, combo1_direction, combo2_direction,

[7:0] font_data,

[3:0] start_data, pipe_data, combo1_data, combo2_data,

Outputs: [7:0] VGA_R, VGA_G, VGA_B

[10:0] font_addr,

[18:0] read_address1, read_address2, start_address, pipe_address, combo1_address, combo2_address

Description: Outputs Red, Green and/or Blue to the VGA, the addresses of different sprites is set in this module. The module also inputs data based of those addresses and then uses different conditions to output different sprites.

Purpose: To determine what color needs to be displayed based off the data from our many different sprites. Also takes into account when to display what and gives precedence sprites over others.

Module: hex driver
Inputs: [3:0] In
Outputs: [6:0] Out
Description: Consists of a conversion from hex to hex display values
Purpose: This module display out to the Hexdisplays

Module: lab8.sv
Inputs: PS2_CLK, PS2_DAT, CLOCK_50, [3:0] KEY,
Outputs:
[6:0] HEX0, HEX1,HEX2, HEX3, HEX4, HEX5, HEX6,
[7:0] VGA_R, VGA_G, VGA_B,
VGA_CLK, VGA_SYNC_N, VGA_BLANK_N, VGA_VS, VGA_HS,
Description: This is the top level module which instantiates all the other modules
Purpose: To make the necessary internal connections from one module to the other in order to output correctly to the hex displays among other outputs.

Post-Lab Questions

LUT	**6345**
DSP	**2**
Memory (BRAM)	**3,048,000**
Flip-Flop	**362**
Frequency	**44.51 MHz**
Static Power	**102.85 mW**
Dynamic Power	**135.97 mW**
Total Power	**312.92 mW**

Conclusion

Looking back at the final project, there were a few things that caused an ample amount of difficulty. Some of these things include using multiple sprites to express one action, handling multiple key presses, dealing with collision detection between two players, adding a combo, using the pipe to have advanced boundaries, implementing a start screen, implementing an end screen, bitmapped font sprites, palletized sprites, and scorekeeping.

This final project was a chance to use the knowledge we gathered all semester to create something enjoyable. Throughout the project we were really able to add what we want and make our own design decisions. Throughout the difficulties of this project, we were able to work through it and achieve the goal which is the most important aspect of any project. We were able to successfully complete the basic functionality of the game as laid out in project proposal as well as implement the more advanced functions.