**ECE 385**

Spring 2023

Final Project

# **Parody Arcade Game**

# **Street Fighter II**

Robert Dela Cruz and Daksh Kalley

Section: AL1

TA: Marwan Eladl

**Introduction**

For our final project, we designed and implemented a parody of Street Fighter II on the FPGA as a System-on-Chip using the NIOS II CPU. Two players use a USB keyboard plugged into the FPGA to control the characters' movement and attacks. The health of each player is tracked and displayed on the HEX display of the FPGA and once a player reaches zero health, the match ends. We display the game on screen using VGA signals.

****

Picture of the Street Fighter II Final Project on a display

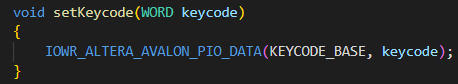
**Written Description of System**

Almost all of our system's design was done in hardware. The software portion just deals with creating a connection to a USB keyboard which was already done in Lab 6, and making a few adjustments. In fact, we began this project by copying over our files from Lab 6 and working off of this lab as a base, as it contained both the NIOS II interface for the USB keyboard, as well as a basic ball which we could control. Each player has their own "ball" instantiation to represent their character.

**Software Portion**

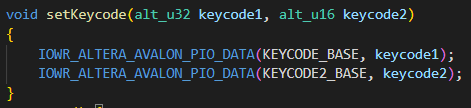
We needed to modify our code so that we could implement two player functionality. This required editing some of the functions in main.c so that simultaneous keycodes could be read.

The first function we edited was setKeycode. Below is the original function which only took a single word as a keycode and put it into our keycode PIO module for hardware use.



Original Lab 6 setKeycode function

We expanded the amount of keycodes we could take by changing the arguments of the function to alt\_u32 and alt\_u16 to receive a total of 6 different keycodes. The reason we split the arguments is because the PIO modules in the quartus platform designer can only go up to 32 bits, and therefore we put 4 keycodes into one PIO module, and the other 2 in another module.



Modified setKeycode function

The next thing we needed to do was edit the main function to set six keycodes rather than one.



Lab 6 calling of setKeycode function

Since the keycodes are stored in an array, the original code just took the first element of the array and set that keycode. In our final project though, we lumped together the keycodes through bit shifting and OR operations before passing them into setKeycode.



Modified main calling of setKeycode function

With these edits, we can take in six keycodes which is more than enough to check for actions from both players.

**Player Control**

We needed to edit the Lab 6 ball.sv in account for all six keycodes as well as changing some of the movement logic. To account for the six keycodes, we created a logic array for each key press that would set a bit in the array to 1 if the corresponding keycode (bit 0 corresponds to keycode[7:0]) in the input was that key. That way instead of checking every single keycode, we could just check to see if the array is all 0's or not. If it isn't then that key must've been pressed. Two separate but very similar ball modules were created, one for each player. This was done so that each character is controlled by separate keys, allowing for two player functionality. For player 1, this would be W/A/S/D for movement, and V for punching. For player 2, the arrow keys control movement and O is for punching.

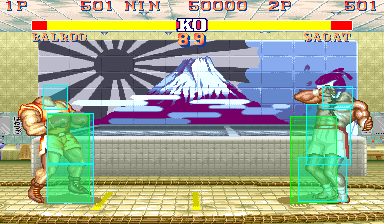
Before the player can move, there are several conditions that are checked in a large if/else if/else chain to make sure a player can only move or attack when it's valid. First are the edge cases where if the player reaches invalid boundaries (past the left/right/top edge of screen or below what's considered the ground of the stage), they should move back to a normal position. For the left/right/top edge cases, this was the same as Lab 6 in which the ball just bounces in the opposite direction to stop touching the edge. In the case of the bottom edge, the player vertically teleports back to the correct ground elevation. A teleport is required due to acceleration from jumping (the character may fall a few pixels past the floor rather than just touch it). After that we need to check if the player is getting knocked back or should start getting knocked back from an opponent's punch, if the player is currently attacking, if the two players' hitboxes are colliding with each other (they shouldn't overlap), and if the player is currently in the air.

If none of these cases are occurring, the player can control the character with the USB keyboard. Whenever none of the movement keys are being pressed, the character stands still. Left and right movement are very simple, the motion of the character is set in that direction. For the up direction, the character jumps (Given an initial velocity of -15 which means going up on the screen) and sets a flag that indicates they are in the air, leading back to the air condition on the next clock cycle. In that condition, the motion of the ball is incremented to create a downwards acceleration. Players can jump forwards and backwards if they move left or right before jumping. When the down direction is pressed, the character starts crouching and can't move until they stop crouching. Pressing the attacking key sets a flag for punching which is used to check if the opponent's been hit.

In the case that the game has ended (a player's health has reached zero) players get set back to their starting position and can no longer move until the game is reset.

**Hitboxes**

In a fighting game, while the characters may be visually dynamic or complex, the game ultimately decides when attacks land or not through the use of hitboxes, which can either be a shape (usually a square or rectangle) or group of shapes to represent the actual space a character or attack takes up. Below is an example screenshot showing these hitboxes. In this case the blue boxes represent what can be hit to land an attack while the green box is used as a means of preventing characters from overlapping.



Street Fighter II Hitbox Visualization

For this final project, we condensed these hitboxes into one large rectangle that merges the functionality of the green (collision detection) and blue (hit detection) hitboxes in the screenshot above. The rectangle also serves as a reserved space for drawing the sprites. The hitboxes were implemented inside of the ball.sv file by resizing the initial ball from Lab 6 and treating it as a rectangle instead.

**Collision Detection**

In order to implement collision detection, the player receives the BallX and BallY outputs (these mark the center of a hitbox) of their opponent that the opponent's location is known. From there, we compare the boundaries of each players' hitboxes and push them away from each other by setting their motion to a direction opposite from the opponent if they begin to overlap.

**Hit Detection**

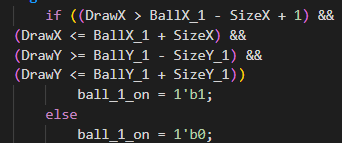
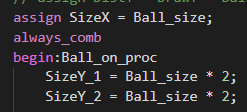
In an actually refined fighting game, there would usually be another separate hitbox that appears when a character attacks, however this idea poses a lot of trouble with our single rectangle design. We could get around this though because our design was a simplified version of the actual Street Fighter II. Since we only got around to basic punching by the end of the project, hit detection could be implemented in a very similar manner to collision detection. Rather than just checking if the boxes were overlapping though, we "extended" the hitbox by checking an area past the edge of the hitbox to emulate the punch hitting the opponent's body. There were additional conditions checking whether the player was punching and if the opponent was crouching to see if the punch should connect or not. When the player does get hit, they get knocked back a short distance. Additionally, if two players punch each other at the same time, they're both pushed away from each other.

**Health**

In order for the game to end, a player's health must reach zero. Each player has an integer keeping track of their health which decrements each time they get hit. The logic for decrementing the health goes together with the knockback logic for hit detection since both should occur when a player is hit.

**Drawing**

As mentioned before, the hitbox rectangle is also used as a space for drawing the sprites. In our color mapper, a representation of the hitboxes was created through the following code (this is a modification of the ball\_on logic used in Lab 6):



ball\_on logic to draw player hitbox (note: same applies to player 2)

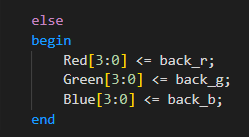
Before implementing the sprite drawings, this would show two solid rectangles on the screen that could be controlled with the USB keyboard. In order to draw a sprite onto the screen, we stored the sprites/images we planned to use into on-chip memory. This was done with "Ian's Helper Tools" which was provided to us on the ECE385 webpage. The conversion gave us a rom, example color mapper, and palette .sv file which we could utilize by adding the generated .qip file to the quartus project.

For our project both characters were fully animated. This was done by using a state machine which cycles through states that represent a different sprite for an animation after a few clock cycles. Each animation had a set of different sprites, for example the idle animation cycles through five different sprites as shown below.



Collection of sprites for idle animation. Each sprite was assigned a different state (Idle1, Idle2, etc.)

In "Ian's Helper Tools" the color mapper .sv file was called (image name)\_example.sv which takes the sprite and stretches it out onto a 640x480 screen. In order to draw the background, we instantiated this example module for the background in our color mapper and wrote the following condition at the end of our drawing if/else chain, meaning that the background would be drawn if nothing else is being drawn.



Code for drawing the background onto the screen

In order to handle drawing the sprites onto the characters' hitboxes, we edited the given example module for each sprite so that instead of stretching it onto a 640x480 screen, we resized it fit inside of an 80x160 box which is the same size as our hitbox. The logic for resizing involves editing the rom address which is shown below:



Multiply DrawX and DrawY by sprite width and height respectively, then divide by hitbox width and height respectively. DrawY part is multiplied by sprite width for row-major order indexing.

In our case, rather than taking a large sheet of sprites and figuring out the exact address for a given sprite, we cropped out each individual sprite and converted each of them individually. This meant we could reuse this code for each sprite as long as we specified the same resolution while converting each one, which is what we did. Each example module takes in a DrawX and DrawY input which determines the address (row-major order) of the rom for drawing the respective sprite. To make sure that the sprite image aligns with the hitbox, we input DrawX and DrawY like this:



Inputs to each "example" module for drawing sprites

BallX/Y - SizeX/Y gives the top left corner of the hitbox, which means that if DrawX and DrawY refer to the top left corner of the hitbox, it will access rom address index 0, the top left corner of the sprite, which is what we want to happen.

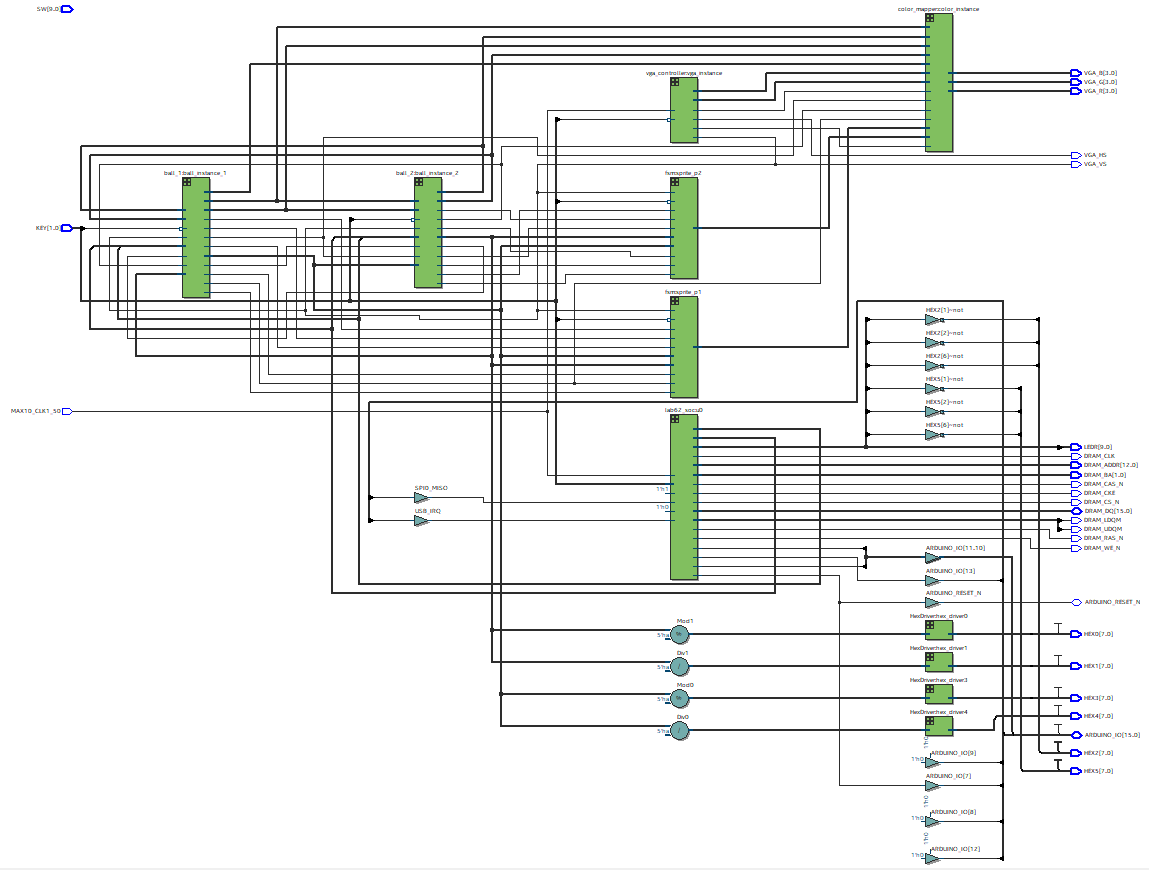
Every single one of these modified example modules were instantiated in our color mapper. Since we had a state machine, figuring out which specific sprite to draw was a matter of checking the current state in a series of if/else if statements. Each sprite has a blue-green background which we can omit by tacking on the following condition in each if statement:



Checks that the RGB values do not match the background color of the sprite

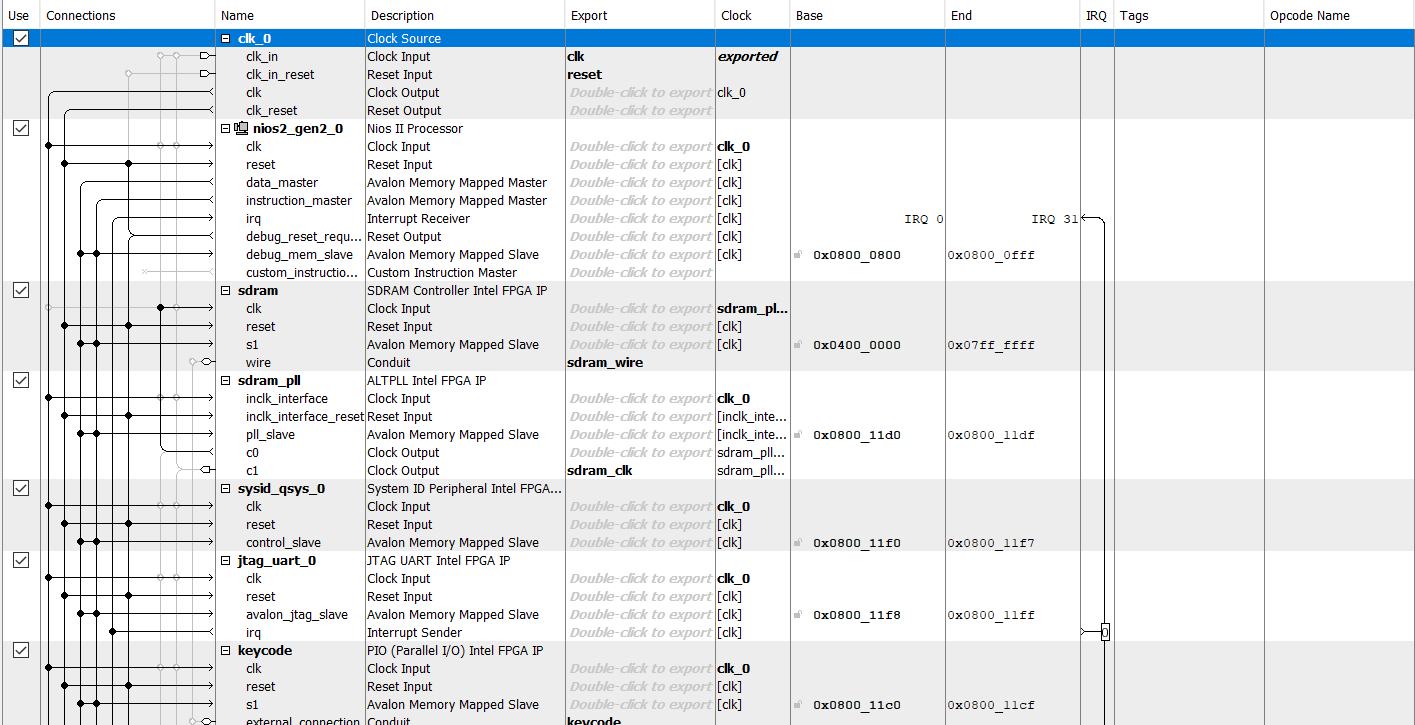
If this wasn't the case, then the RGB outputs would match the RGB outputs of the respective sprite, drawing the character onto the screen.

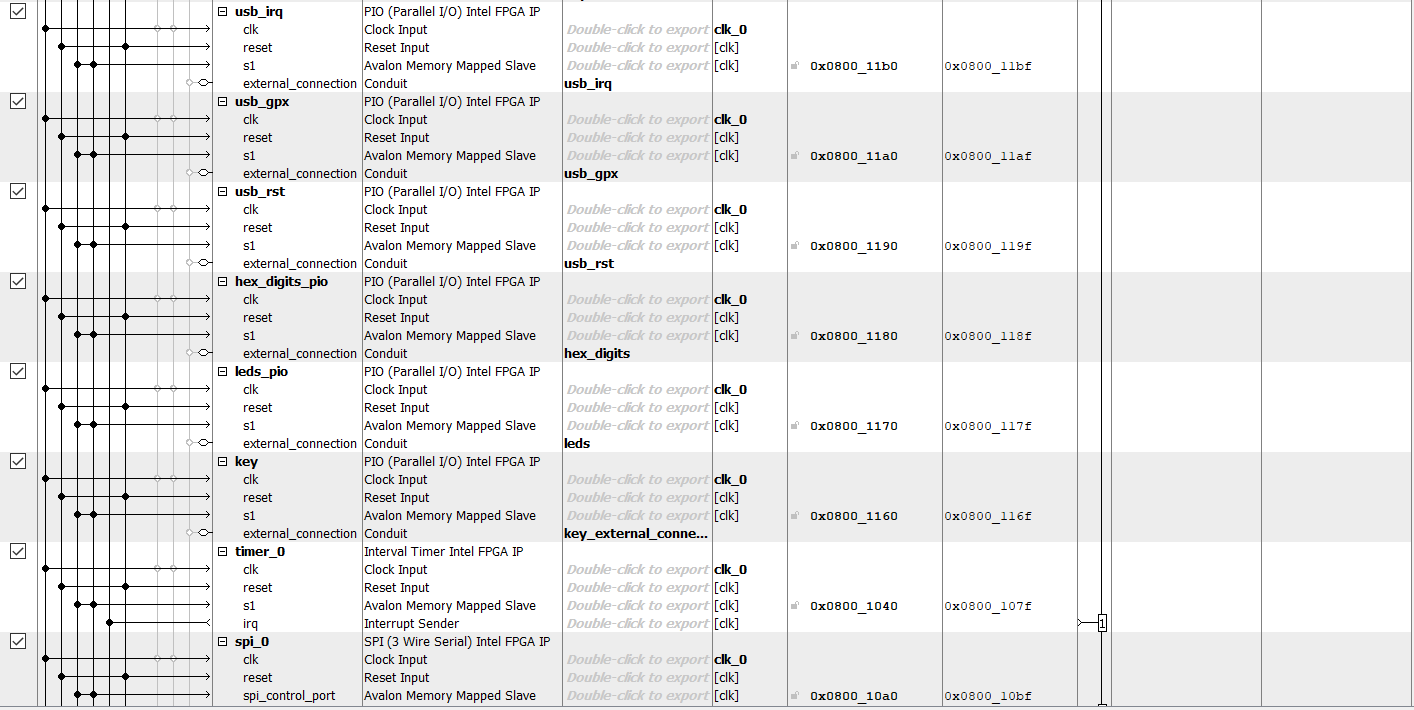
**Block Diagram**

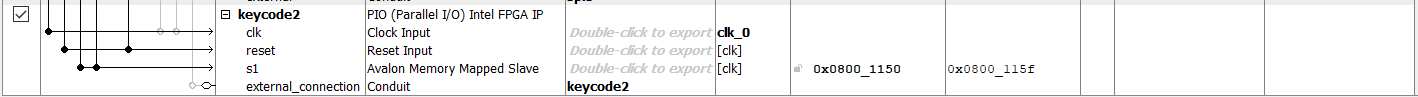
****

Top Level Block Diagram of Final Project

**Platform Designer**



****

****

clk\_0: Clock source generates clock for all the other modules.

nios2\_gen2\_0: 32-bit NIOS II/e processor that handles C code compilation and programming.

sdram: Off chip memory that stores the software program

sdram\_pll: Provides the clock signal for the SDRAM chip and performs a delay first in order to give the sdram a better window of time to read in data.

sysid\_qsys\_0: System id checker makes sure configurations are compatible between the software and hardware before the program can be run.

jtag\_uart\_0: Allows use of the host computer terminal to communicate with the NIOS II (can use print and scan statements in C)

keycode: PIO module that connects to the USB keyboard and reads four of the keycodes from the keyboard.

usb\_irq: PIO module that handles interrupt requests. Necessary for utilizing the USB keyboard.

usb\_gpx: General-purpose multiplexed output. Selects between 4 output signals: OPERATE, VBUS\_DET, BUSACT, and SOF. Necessary for utilizing the USB keyboard.

usb\_rst: PIO module that deals with resets. Necessary for utilizing the USB keyboard.

hex\_digits\_pio: PIO module that connects to the hex display of the FPGA. Displays the health of each player.

leds\_pio: PIO module that connects to the LEDs of the FPGA.

key: PIO module that connects to the key buttons of the FPGA.

timer\_0: Interval timer IP that helps the NIOS keep track of passed time. Needed to keep track of time-outs required by the USB.

spi\_0: SPI port peripheral used in conjunction with the written four functions to read and write registers to the MAX3421E chip through the SPI. Allows data transfer between master and slave.

keycode2: Essentially a copy of the keycode module but reads the other two of the six keycodes because a single module can only hold up to 4 keycodes.

**Module Descriptions**

**Module: HexDriver in HexDriver.sv**

Inputs: [3:0] In0

Outputs: [6:0] Out0

Description: The HexDriver module inputs a 4-bit binary digit and outputs a 7-bit value corresponding to the hex value to display the digit on the 7-segment LED FPGA display.

Purpose: This module describes the logic required to output a hex value on the 7-segment LED FPGA display. Four instantiations of this module are utilized to display the health of each player in the game. The health goes from 10 to 0, and the health resets when a new game starts.

**Module: fsm in control.sv**

Inputs: w\_in, a\_in, s\_in, d\_in, v\_in, air, Clk, Reset,

Int health, opphp

Outputs: int sprite

Description: Finite state machine with wait-states and states for each sprite animation’s frame. A state-label package is imported with enumerated logic for each state name. The module takes in keyboard inputs such as “WASD” or arrow keys, and also inputs flags such as ‘air’ and the health of each player. Outputs a single sprite state integer so the color\_mapper module knows which sprite to draw. Module also changes animation after reading the health and opponent’s health to indicate a winner and loser.

Purpose: This module describes our finite state machine which decides which sprite to draw and cycles through sprites to complete animations for each player’s actions.

**Module: color\_mapper in Color\_Mapper.sv**

Inputs: [9:0] BallX\_1, BallY\_1, BallX\_2, BallY\_2, DrawX, Draw, Ball\_size,

vga\_clk, blank, crouch1, crouch2,

int sprite\_state1, sprite\_state2

Outputs: [7:0] Red, Green, Blue

Description: Color Mapper module has logic for rgb values of each sprite. Also has drawing logic for each player’s sprite on the screen. Inputs the coordinates of the players, the size of their hitboxes, flags for crouching, and the sprite-state values.

Purpose: Instantiates sprites in on-chip memory using Ian’s sprite-helper tool. Draws each player’s sprite, draws the player animations and background. Determines which sprite to draw by looking at the sprite-state output values from the finite state machine of each player.

**Module: ball\_1 in ball.sv**

Inputs: Reset, frame\_clk,

[47:0] keycode,

[9:0] OppX, OppY

knockback, opp\_crouch

Int opp\_hp

Outputs: [9:0] BallX, BallY, BallS,

int health,

crouch, w\_out, a\_out, s\_out, d\_out, v\_out, air, hitopp

Description: Creates logic for the position and motion of the leftmost (player 1) character. Checks each section of the keycode logic to determine which keys are pressed (movement or punch). Has counter to indicate health, opponent’s health, and delay when punched (knockback). Has logic for jumping, crouching, moving left/right, and punching. Has logic for hitbox detection, preventing players from crossing and pushes players against each other. Instantiated in top level and sends air and crouch flags to initiate jumping animation and changing the sprite hitbox when crouching.

Purpose: To analyze keycode input and control player movement and actions such as jumping, crouching, punching, and taking/inflicting damage.

**Module: ball\_1 in ball.sv**

Inputs: Reset, frame\_clk,

[47:0] keycode,

[9:0] OppX, OppY

knockback, opp\_crouch

Int opp\_hp

Outputs: [9:0] BallX, BallY, BallS,

int health,

crouch, up\_out, left\_out, down\_out, right\_out, o\_out, air, hitopp

Description: Creates logic for the position and motion of the rightmost (player 2) character. Checks each section of the keycode logic to determine which keys are pressed (movement or punch). Has counter to indicate health, opponent’s health, and delay when punched (knockback). Has logic for jumping, crouching, moving left/right, and punching. Has logic for hitbox detection, preventing players from crossing and pushes players against each other. Instantiated in top level and sends air and crouch flags to initiate jumping animation and changing the sprite hitbox when crouching.

Purpose: To analyze keycode input and control player movement and actions such as jumping, crouching, punching, and taking/inflicting damage.

**Module: lab62 from lab62.sv**

Inputs: MAX10\_CLK1\_50,

[ 1: 0] KEY,

[ 9: 0] SW,

Outputs: [ 9: 0] LEDR,

[ 7: 0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5,

DRAM\_CLK, DRAM\_CKE, DRAM\_LDQM, DRAM\_UDQM, DRAM\_CS\_N,

DRAM\_WE\_N, DRAM\_CAS\_N, DRAM\_RAS\_N

[12:0] DRAM\_ADDR

[1:0] DRAM\_BA

[15:0] DRAM\_DQ, ARDUINO\_IO

[3:0] VGA\_R, VGA\_B, VGA\_G

VGA\_HS, VGA\_VS, ARDUINO\_RESET

Description: Top level module utilized in program. Instantiates lab62\_soc. Instantiate our vga\_controller, ball\_1/ball\_2 player movement modules, color\_mapper, and two fsm modules for each player’s sprite animations.

Purpose: Describes our system on chip module and connects modules such as LEDs, Hex displays, and the VGA display. Integrates the NIOS II system with the rest of the hardware. Connects signals from the movement modules to our finite state machine and color mapper to draw the players in the game.

**Module: (Image Name)\_example (Applies to every sprite)**

Inputs: vga\_clk, blank

[9:0] DrawX, DrawY

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

Description: Sprite image module that instantiates the rom and palette of a sprite. Calculates the rom address by finding the row-major order index based on DrawX and DrawY. Outputs the RGB signal for that given address.

Purpose: Obtain the correct RGB signal to draw the pixels of the character. Resize the image by calculating the correct pixel to map onto.

**Sprite Modules Follow Pattern Below:**

**26 Sprites per Player, 1 Sprite for Background, 53 Total**

**Module: (Image Name)\_rom (Applies to every sprite)**

Inputs: clock

[12:0] address

Outputs: [2:0] q

Description: Uses the rom address as an index for the memory pertaining to the sprite. Outputs the color palette index corresponding to the specific pixel given by the address.

Purpose: Access the sprite in the on-chip memory and figure out which colors to choose in the palette.

**Module: (Image Name)\_palette (Applies to every sprite)**

Inputs: [2:0] index

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

Description: Takes the index from the sprite rom and finds the RGB signals in the palette corresponding to that index.

Purpose: Return the RGB signals of the pixel.

**Design Resources and Statistics**

| LUT | 10935 elements |
| --- | --- |
| DSP | N/A |
| Memory (BRAM) | 1,640,448 bits |
| Flip-Flop | 3450 registers |
| Frequency | 132.73 MHz |
| Static Power | 96.18 mW |
| Dynamic Power | 0.71 mW |
| Total Power | 106.20 mW |

**Conclusion**

With the help and inspiration from many CAs, we were able to create a functioning game of Street Fighter II in the end. Although it's simplified in terms of attacks, the game works as intended. In the proposal we listed round tracking as a necessary feature which we unfortunately did not get around to, although the game still functions without this.

Throughout the four weeks spent on this project, we encountered many bugs that required time and prevented us from adding additional features or polishing the game. Granted more time, we would want to implement different attacks such as projectiles and kicking. Additionally, we had reached maximum capacity on memory implementation bits which meant we could not fit any more sprites into our on-chip memory. One solution to this is optimizing our screen resolution for each sprite and the background however we had already converted and started working on our sprite images. We also noticed artifacts around our characters which we concluded to be a timing issue. Our fix to this bug allowed us to remove glitching pixels and allow for smoother animations yet the implementation would not allow us to compile in a reasonable amount of time. Overall, our work from previous labs and experience debugging course assignments helped us customize and build a final project that we are satisfied with.