ECE 385

Spring 2024

Final Project Report

Multiplayer Ultimate Fighter Game on Spartan-7 FPGA

Ambika Mohapatra (ambikam2)

Ivan Ren (iren2)

TA: Alara TIN

Introduction

Overview and Motivation

The final project we decided to build on the Spartan-7 FPGA is titled “Ultimate Fighter”, which is our unique take on the famous “Street Fighter” SNES platform game first released in 1987 by Capcom. The characters in our fighter game are inspired from the popular anime series “Baki Hanma”, where player 1 can choose to play as “Doppo Orochi”, and player 2 can choose to play as “Baki Hanma”. The character sprites for Doppo Orochi are procured from the website “Spriters-Resource”,[1] and Baki Hanma character sprites are taken from “Deviantart”. [2]

On a high level, the game has 3 states - Welcome, Fighting, Game Over.

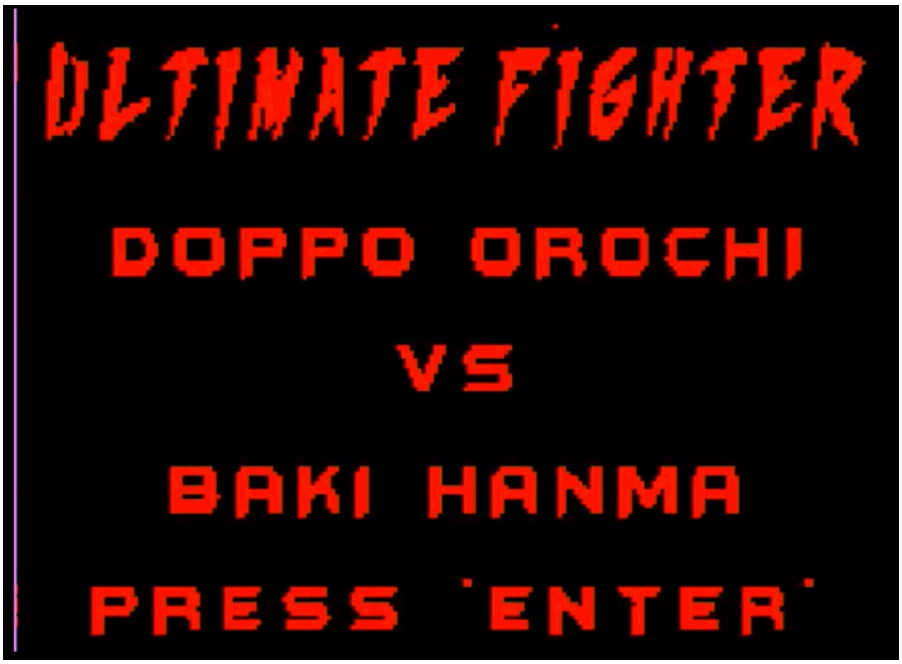

Figure 1. Welcome screen

The game initialises from the welcome screen to the fighting state based on the keyboard entry of the “enter” key. Once in fighting state, the 2 players can control the characters of Doppo (left) and

[1] Spriter’s resource, Alpha 3 Sagat Sprites. Link - www.spriters-resource.com/arcade/streetfighteralpha3

[2] Deviantart, Baki Hanma Sprites. Link - www.deviantart.com/srchimuelo/art/Baki-Hanma

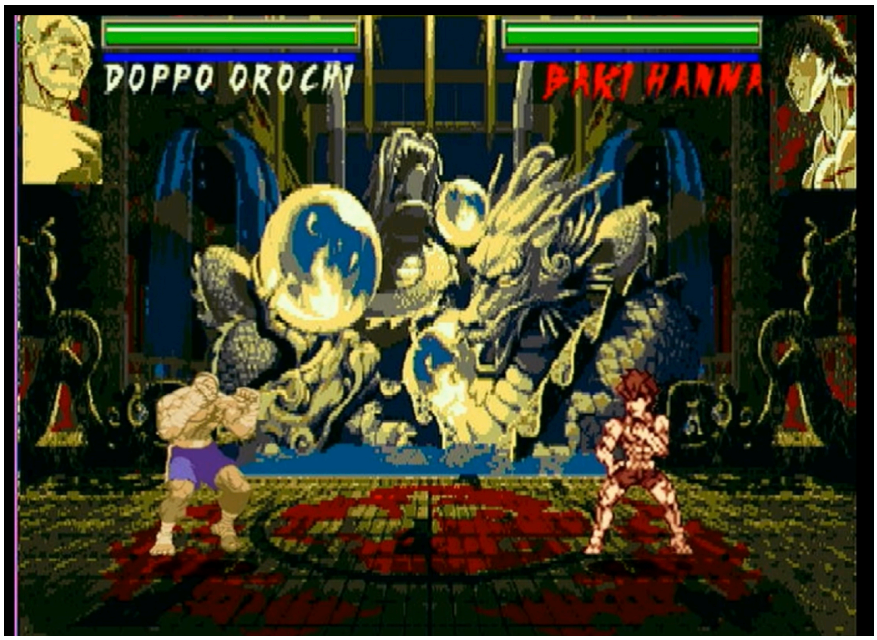

Figure 2. Game screen

Baki (right) via multiplayer-configured keyboard that allows up to 6 simultaneous keyboard inputs. Each player has 6 different possible moves - Walk (2 directions), Jump, Punch, Kick, Block.

The characters have configured idle animations such as breathing, and dynamic animations such as variable shadows, walking, jumping, block bars, “critical hit” sprites, variable color health bars etc. The objective of the game is to get within suitable range of the opponent, the characters can punch and kick their opponent (with kicks having a higher damage) to reduce their health bar’s value. Once a character’s health falls below 40%, they enter into rage mode where their kicks and punches can damage their opponent significantly more.

Figure 3. Game over screen

When the health of a character reaches 0, indicated by the health bar, the game moves into game over state where the winning player's character stands over the losing player's character's "death" sprite. The game is reset via the btn[0] on the FPGA to re-enter welcome screen state.

Project Highlight Features

- Entirely unique implementation of the popular anime "Baki Hanma" as a multiplayer fighting game, with absolutely no code used from online resources. All implemented features are inspired from pro fighting games.
- Players can choose to either play as character Doppo Orochi or Baki Hanma, original anime characters whose sprite sheets were modified by us in Adobe Photoshop. Game background is also unique and entirely self-made in Adobe photoshop.

- Dynamic character movement of breathing, walking, and jumping. Dynamic character shadows which change size on jumping and are transparent (not opaque) shaped in ovals to give an authentic shadow effect.
- Players can kick or punch, with punch having lower damage and range but executing quickly, kick having higher damage and range but executing slowly. At every punch and kick landed the character sprites flash in bright red and their health decreases. Also “critical hit!” text appears on every landed kick (but not punch).
- The hit logic is extremely range sensitive i.e. the punches and kicks only hit when within range. Also, if a player jumps then the other player’s punch/kick cannot hit them, but if they are both at the exact same vertical coordinates then even mid-air attacks can get registered. This was implemented with additional hit logic to check both vertical and horizontal player coordinates.

- Players can block attacks, with a dynamic block bar in blue that indicates the block amount left. The bar regenerates at half speed then it gets exhausted, to ensure players can’t block excessively. Once block bar is exhausted players can get damaged despite holding block.
- Once player health falls below 40%, the health bar color changes from orange to red and the player enters “RAGE” mode - where the sprites change significantly in stance and flames appear around them. In rage mode their speed and attack damage increases by 50%. “Rage” text appears besides character mugshots as well.
- Once either player’s health falls to 0, the game is over with a “fatality” screen and the death sprite of the losing character appearing exactly at the place where they died, and the winning character’s breathing sprite. All character movements are suspended once game is over.

- A hidden feature is to hold the “enter” key on keyboard after game over to glance into the game background after game over to see which player won by how much of their health remaining, which players had their rage modes activated etc.

Files Hierarchy Description

The Ultimate Fighter game uses a toplevel named “lab622xsa” since it utilises similar modules as Lab 6.2’s mb_usb_hdmi_top module. The modules in the toplevel connection include hex displays, vga controller, color mapper, ball1 (player 1), ball (player 2), states, and clocking wizard.

These modules work together to project the welcome screen from the game’s commencement. The “Enter” keycode is a required input in welcome screen state in order to move into fighting screen state. When the “Enter” keycode is input by the player(s), the states file selects the

the game background instead of the welcome screen background to be input into Color Mapper, and the game enters fight screen state. The 2 players, Doppo on the left and Baki on the right appear on screen with the fighting background. The sprites have an idle animation of breathing configured via switching between 2 standing sprites through the doppo_walk and baki_walk files.

When a player decides to move horizontally on screen, dynamic shadows follow the players which is achieved through shadow sprites configured in Color Mapper which track the player’s X-axis motion. Horizontal movement leads to the players’ walking animation as the sprites switch between standing and walking sprites. When the character’s jump, the Y-axis position of the character sprite decreases and the jump sprite appears on screen, while the dynamic shadow of the player changes size to become smaller.

The ball and ball1 modules configure Baki and Doppo’s characters respectively, where all the keycode movements, game bounds, in-game gravity and player collision logic is configured by intensive modification of Lab 6.2’s ball module. The characters can attack via punch or a kick, with punch having a lower damage but greater range and kick having a greater damage but lower range.

The range of the punch and/or kick being hit is monitored in Color Mapper, which then draws a red health bar onto the character healths which lower as they sustain more damage. The Y-axis coordinates of both players are also a factor in determining whether a kick or a punch is detected, so only when both players are at the same height an attack can be detected.

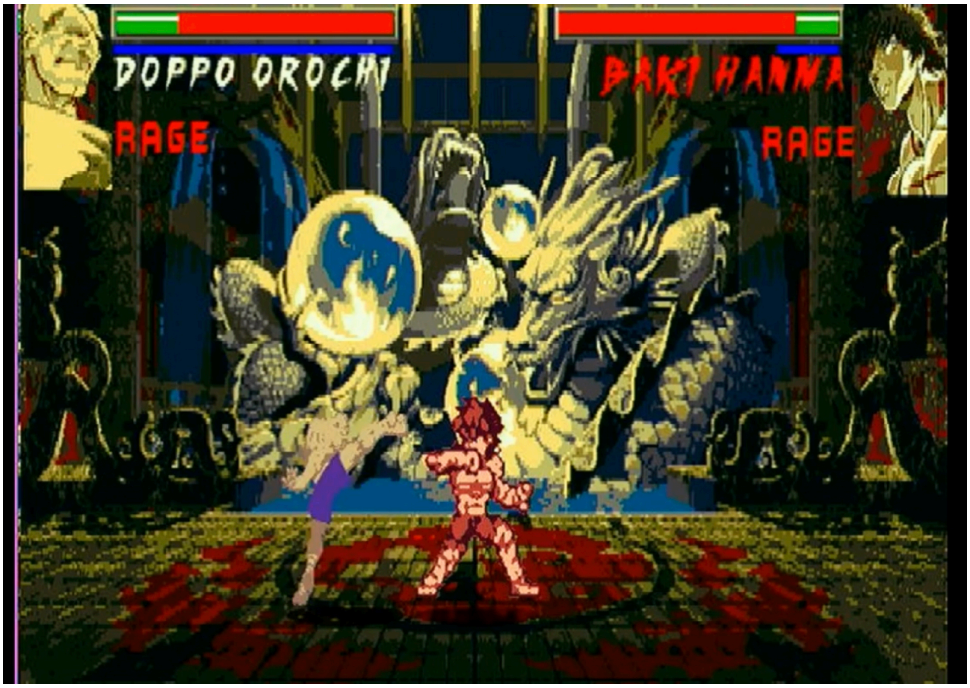

Figure 4. Block bar (right) enabling damage protection

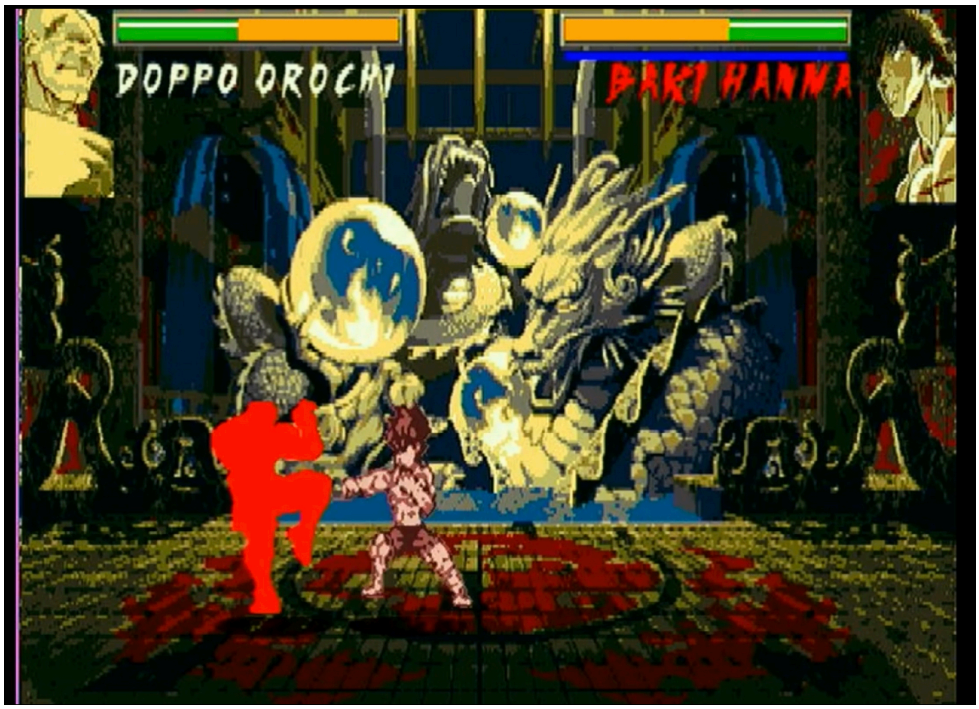

Figure 5. Exhausted block bar (left) not providing damage protection

The block logic is configured via Color Mapper, where the block can only be held for 3 seconds at a time before the block bar “runs out”. Then, attacks (punches and kicks) from the opponent get registered despite holding block. The block bar is coded to regenerate at half the speed of its exhaustion to ensure players cannot block excessively and hence utilise their blocks effectively.

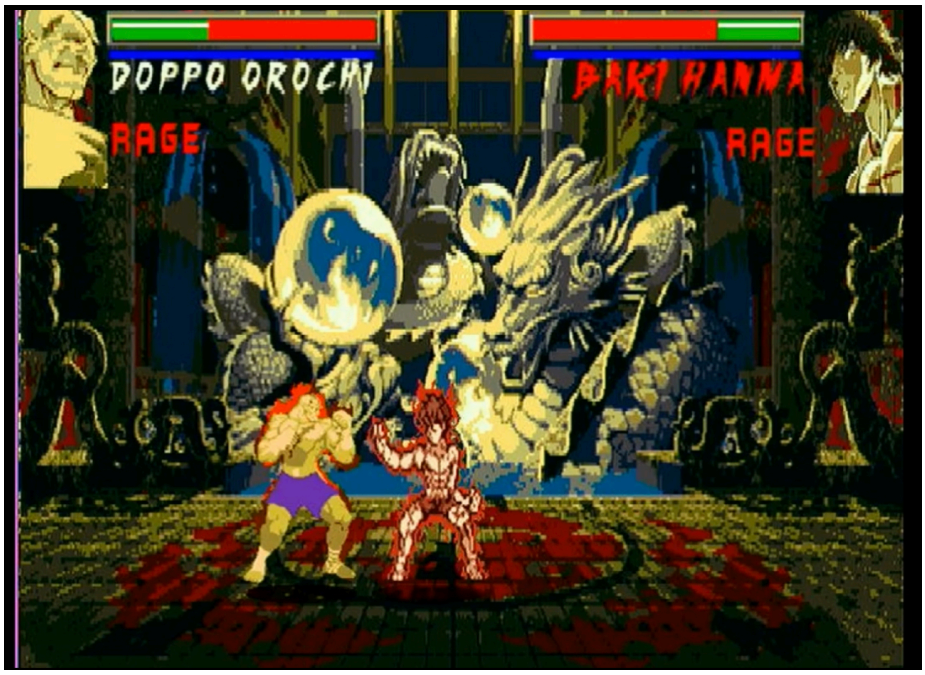

Figure 6. Rage mode

The highlight feature of this game is its rage mode. Once player health falls below 40%, the health bar color changes from orange to red and the player enters “RAGE” mode - where the sprites change significantly in stance and flames appear around them. In rage mode their speed and attack damage increases by 50%. “Rage” text appears besides characters’ health bars as well.

Once either player’s health falls to 0 by visual indication of the health bars, the game enters game over state (via the state machine). The player movement is suspended and the death sprite of the losing character is shown on screen.

Multiplayer Keyboard SPI

Player 1 (Doppo)		Player 2 (Baki)	
Keycode	**Function**	**Keycode**	**Function**
A	Move Left	Left Arrow	Move Left
D	Move Right	Right Arrow	Move Right
W	Jump	Up Arrow	Jump
S	Block	Down Arrow	Block
R	Punch	Numpad 1	Punch
F	Kick	Numpad 0	Kick

Figure 7. Player key codes and functions

As described in the table, the multiplayer keycode configurations for each player required using the MAX3421E SPI previously used in Lab 6.2. The single byte as well as multiple byte read and write configurations in MAX3421E.c remained unchanged since Lab 6.2 as these keyboard interactions were crucial for configuring multiplayer mode in the game, but were also sufficient so did not need modifications. The keycode GPIO in our Microblaze SoC were also sufficient for configuring multiplayer keycode recognition, with keycode0_gpio [23:0] being used for Player 1's movements and keycode1_gpio [23:0] being used for Player 2's movements. The [31:24] bits of both the keyboard GPIOs are left unused since the internal buffer of gaming keyboards does not allow recognition of more than 6 key presses at a time.

Multiplayer keycodes were majorly configured by modifying lw_usb_main.c from Lab 6.2. The `player1Buff` and `player2Buff` arrays are utilized to store keycodes received from a connected USB keyboard, processed through a USB-to-HID interface. The main function initializes the USB hardware using `MAX3421E_init` and `USB_init`, then enters an infinite loop to handle USB device connections and keyboard data.

In the loop, the `USB_Task` function monitors the USB state. Once the state reaches `USB_STATE_RUNNING`, the loop identifies the connected device type using

`GetDriverandReport`. If the detected device is a keyboard (indicated by `device == 1`), the `kbdPoll` function is used to fetch the latest keycodes into the `kbdbuf` structure.

The keycodes are extracted from `kbdbuf.keycode` and categorized based on their values. Keycodes less than or equal to 30 are classified for player 1 and stored in `player1Buff`, while higher values are for player 2, going into `player2Buff`. Each buffer is limited to three keycodes. If either buffer reaches its limit, additional keycodes are ignored until the next cycle.

After collecting keycodes, the function `printHex` converts the keycode arrays into hexadecimal format, outputting them to separate channels for display. Both player's keyboard inputs are displayed to Vitis serial terminals via xilprintf functions.

This function utilizes the `XGpio_DiscreteWrite` method to output the hexadecimal value based on the respective channel, channel 1 for `player1Buff` and channel 2 for `player2Buff`. The buffers are then cleared for the next round of input collection from the 2 player's keyboard inputs.

VGA Controller and Color Mapper Description

VGA, or Video Graphics Array structured our displays into a grid of pixels, 640 columns by 480 rows. Each pixel has red, green, and blue (RGB) values controlled by analog voltage signals. These signals also controlled the intensity of the electron beam, which drew the image on the screen. The current position of the electron beam is tracked by coordinates called DrawX and DrawY.

Deflection yokes in the monitor could adjust the beam's position to draw images in row-major order. The beam starts at the leftmost pixel of each row after finishing the previous row, triggered by a horizontal sync (hsync) signal. After completing the image, a vertical sync (vsync) signal resets the beam to the top-left corner for the next image. The screen typically refreshed at 60 Hz, requiring the vsync signal 60 times per second for a time period of 16.67ms.

In the described lab setup, various modules such as player movement (ball and ball1), color mapper, and VGA controller interact together to create a HDMI signal to be displayed onto the screen (VGA - HDMI IP converts the VGA signals into HDMI). DrawX and DrawY from the VGA controller inform the color mapper about pixel positions for drawing sprites or backgrounds. Player movement modules use keycodes from a GPIO block (from ball and ball1 modules) to update sprite positions based on player inputs.

The pixel_clk from the VGA controller drives the Color Mapper's timing, ensuring pixels are drawn correctly. The vsync signal, being slower at 60 Hz compared to the other clocks (25 MHz or 50 MHz), is used for timing-sensitive modules or counters, allowing for sprite switching or animation. The Color Mapper contains the information about drawing sprite animations, health bars, block bars, game backgrounds.

State Machines

Start Screen, Fighting Background, and Game Over Screen

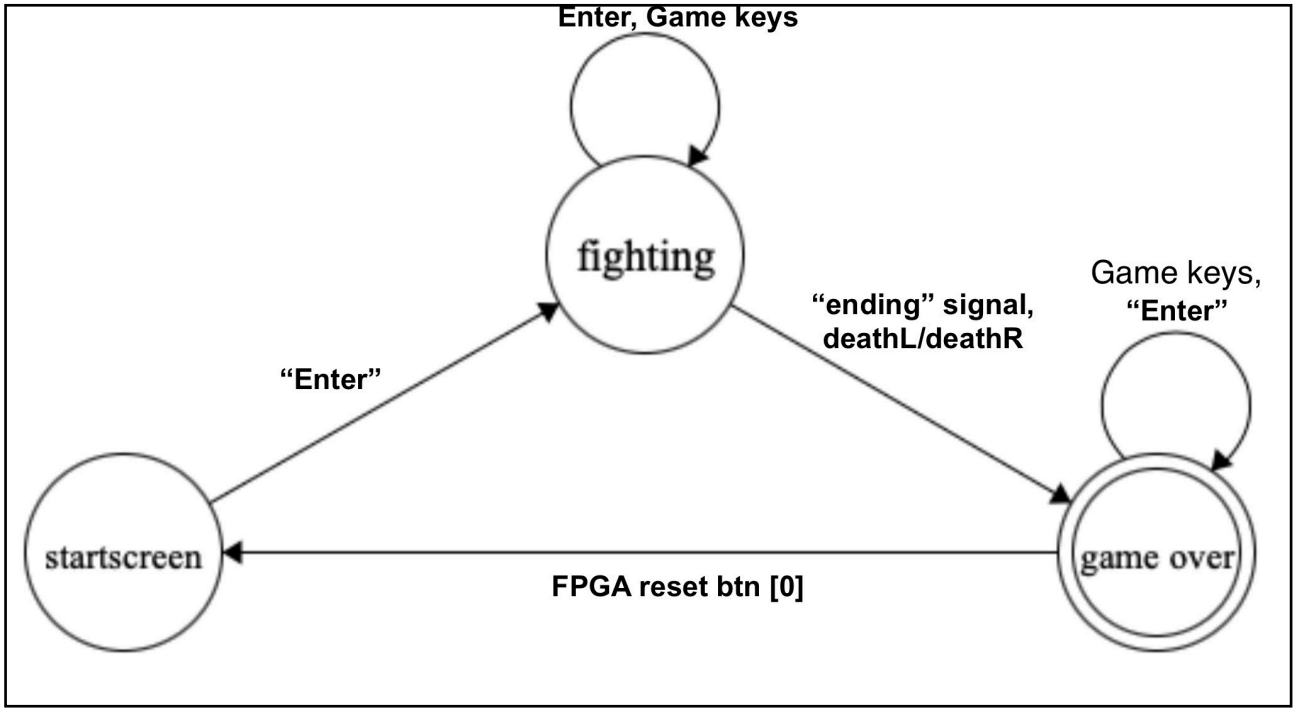

Figure 8. Block bar (right) enabling damage protection

The state machines used in this game are encapsulated in states.sv file, wherein the state machine chooses which background among “Startscreen, Fighting, and Game Over” should be chosen and sent as an output to be drawn by Color Mapper. The “enter” key is an accepted input from start screen to enter into game screen (fighting state). In fighting state, the player’s movement control keys are used to “play” the game and their interaction allows their health to decrease upon attack impact. When either of the player’s health is 0, the “ending” signal is high in Color Mapper which signifies the state machine to switch from fighting to game over state. Here, the character movements

Sprite Drawing

The sprites for this game were drawn using the Image-to-COE converter tool by CA Arnav Seth (inspired by the tool made by CA Ian Dailis).[3] The scripts in this tool utilises a k-means clustering method to configure images into a palette and generates a COE file which the Block Memory (BRAM) IP then accepts as input to store the image’s color and resolution information.

```
PS C:\Users\Ambika\Downloads> cd Image_to_COE-master
PS C:\Users\Ambika\Downloads\Image_to_COE-master> cd Image_to_COE-master
PS C:\Users\Ambika\Downloads\Image_to_COE-master\Image_to_COE-master> python main.py
Input image (eg: waterfall.jpeg): doppo.png
Number of bits per pixel to store: 3
Desired output horizontal resolution: 90
Desired output vertical resolution: 150
Using 40500 / 1490944 available M9k memory bits
Design may still not fit. M9k block usage is weird.
Resizing image... Done
Palettizing image... Done
Generating coe (Memory Instantiation File)... Done
Generating ROM module... Done
Generating palette module... Done
Generating example module... Done
Generating .qip file... Done
Generating output image... Done
Output files are in ./doppo/
PS C:\Users\Ambika\Downloads\Image_to_COE-master\Image_to_COE-master>
```

Figure 9. Python scripts to create sprite

[3] Image to COE converter by Arnav Sheth, link - https://github.com/amsheth/Image_to_COE

The python scripts configure jpeg/png images into SystemVerilog modules where the user can input the desired color bits, horizontal, and vertical resolutions for each sprite. The palletized module of the image (named ____palette.sv) and an example drawing tool (named ____example.sv) are generated. The background sprites of welcome screen, fighting screen, and game over screen were drawn as 320x240 resolution (1 bit color for welcome screen and game over screen, 4 bit color for the fighting screen) and stretched across the 640x480 monitor screen. The character sprites of Doppo and Baki were generated as 90x150 resolution and used 3 bit color configurations. A hot pink background was utilised as the sprite backgrounds which was then used to filter out the sprite from its background using color mapper logic to achieve transparency. Critical Hit!" text sprites are also configured to appear on screen every time a player lands a kick and "Rage" text sprite is configured to appear during the rage mode of characters.

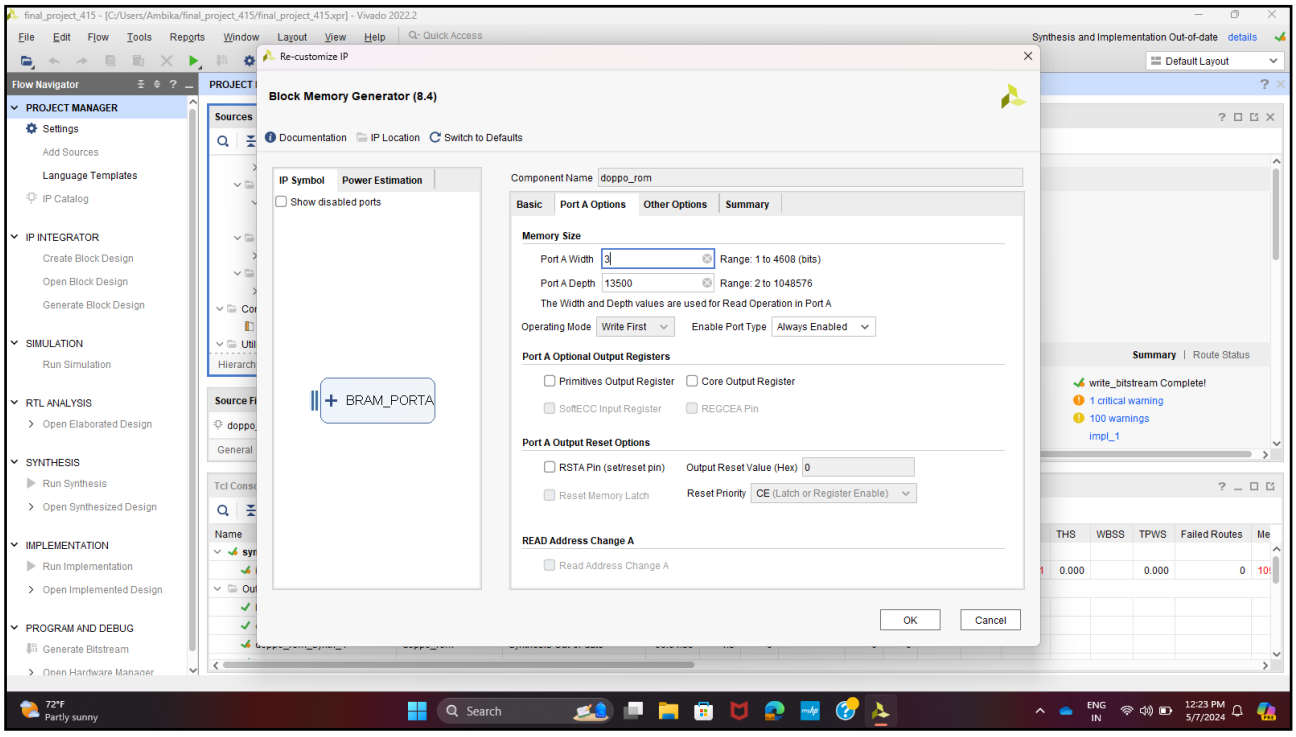

Figure 10. IP Block configurations (for sprites)

The Block Memory Generator IP is configured for each sprite used in this project. A Single Port ROM configuration is used with the “Always Enabled” pin is used. The depth of the ROM is selected as the number of color bits of the sprite (3 bits = 8 colors) and the width is configured by the product of its horizontal and vertical resolutions (for example, the width of the character is 90 and vertical height is 150, hence the ROM’s port depth is 90 x 150 = 13500).

Top Level RTL Diagram

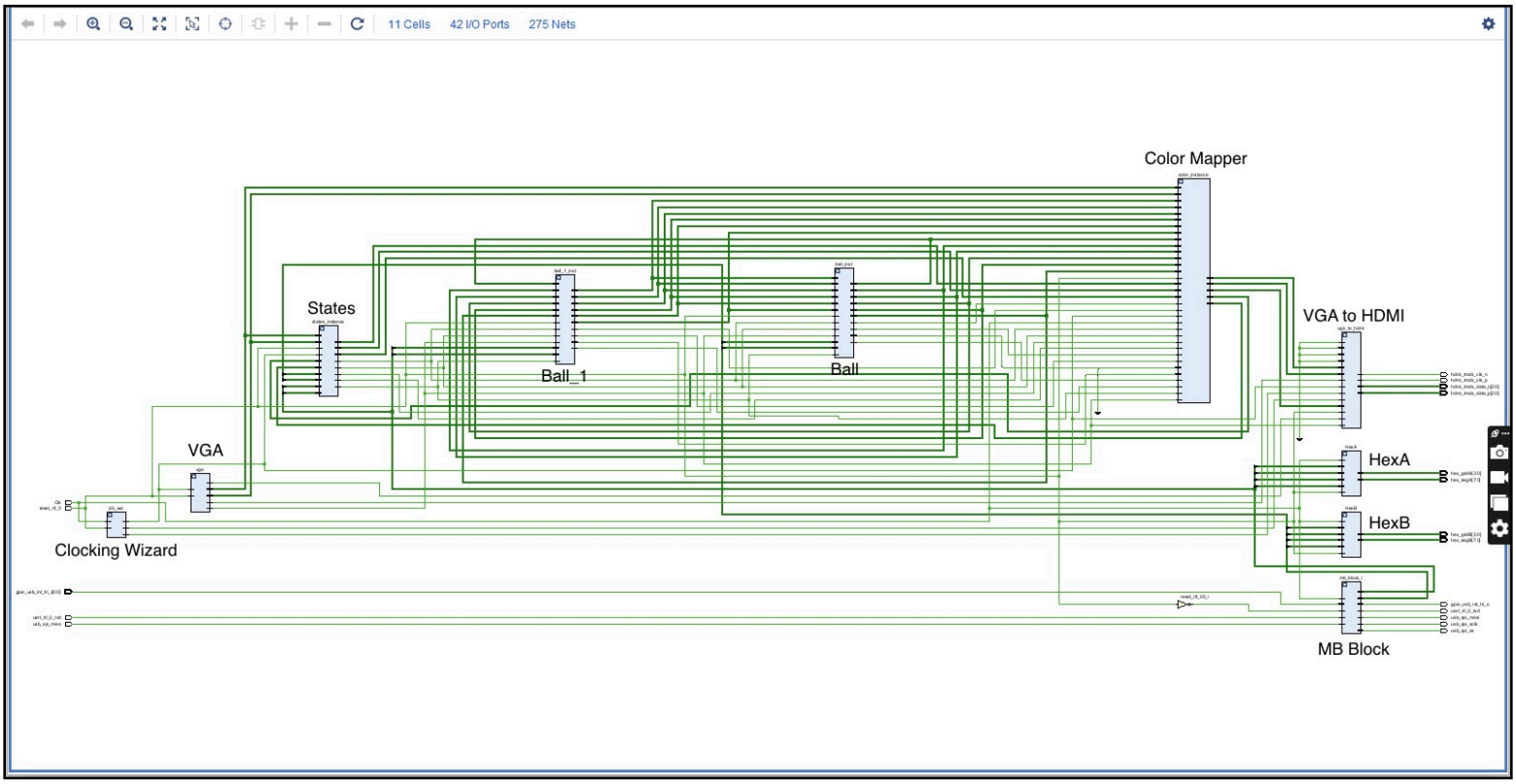

Figure 11. Annotated Top Level RTL Schematic of Project

Written Descriptions of Microblaze System on Chip

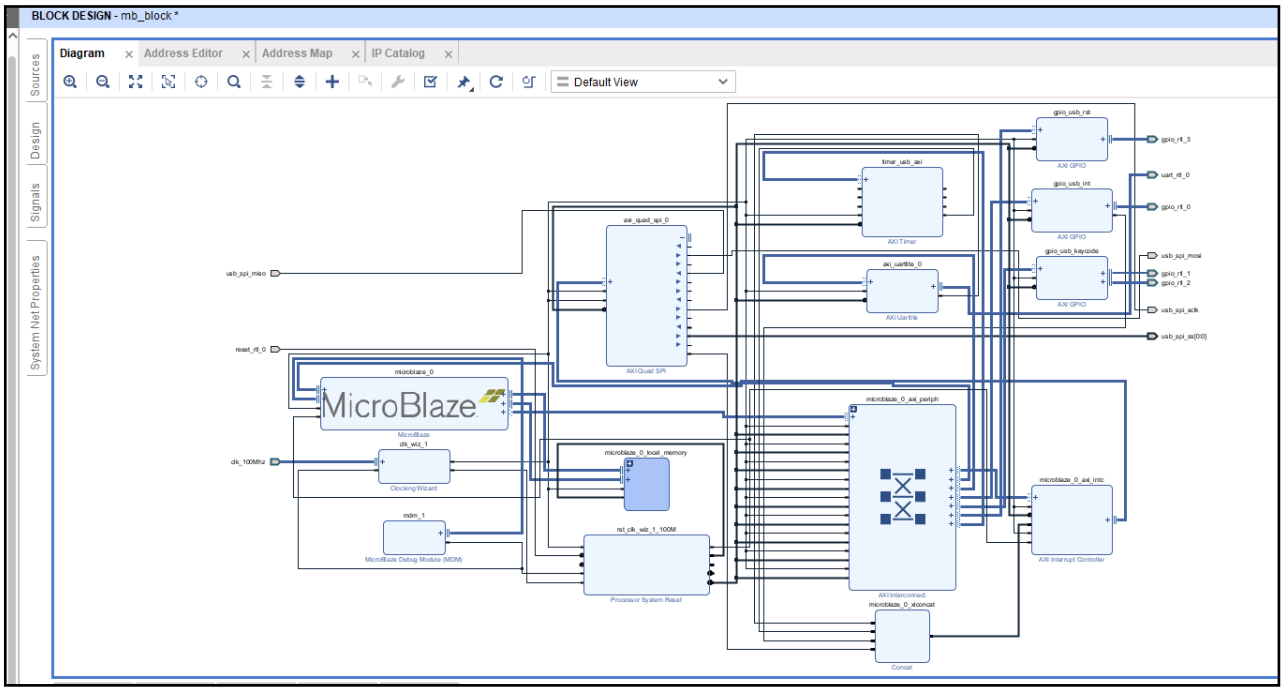

Figure 12. Block design of the configured Microblaze CPU SoC for Final Project

he Microblaze SoC used in this project is identical to that used in Lab 6. The following descriptions explain the Microblaze modules and IPs (which are the same as described in the Lab 6 Report.).

MicroBlaze Processor (microblaze_0)

This block is the processor block, which as its name suggests, handles all of the processing of data and runs the actual programs that are written in C. By itself, it has limited use but can be hooked up with many peripherals that can allow to take in input from external devices and output data to other external devices. There are a few different presets that the MicroBlaze processor can be set up with, the specifics of which will be explained in one of the post lab questions; for our purposes, the "Microcontroller" preset gives us all the required functionality.

MicroBlaze Local Memory (microblaze_0_local_memory)

This is the local memory for the MicroBlaze processor, which is used to store both data and programs, and follows the modified Harvard computer architecture, further explained in the post lab questions. The size of the local memory is initially determined when setting up the processor and can be between 4 KB and 128 KB.

Clocking Wizard (clk_wiz_1)

The clocking wizard IP is a block that takes care of producing the required clock signal, or signals, in our case. It can have up to 7 output clocks as well as 2 input clocks and has options to adjust the buffering, feedback, and timing of the clocking network. There are two instantiations of the clocking wizard in our lab, for 6.2, a clocking wizard is used in the MicroBlaze block design with USB peripherals and connected to the local memory, processor, AXI Interconnect, GPIO and AXI Quad SPI. Another wizard is instantiated to produce the 2 outputs of 25 MHz and 125 MHz to provide the correct screen frame rate for the HDMI module, which requires a TMDS clock and a normal clock, with the TMDS clock being 5 times faster.

Processor System Reset (rst_clk_wiz_1_100M)

This module is responsible for producing all the required reset signals for all of the other modules in the block design. This module has the ability to produce external and auxiliary reset signals and modify whether they are active low or high, and how they are synchronized with the clock; these customized reset signals give the design much more flexibility when it comes to enabling/disabling features.

MicroBlaze Debug Module (mdm_1)

This module is enables debugging for this processor with support for a JTAG software debug tools and support for debugging 32 MicroBlaze processor concurrently. This module can be made to communicate through UART, and can be configured to enable different levels of debug functionality, allowing for advanced tracing both externally and programmatically.

AXI Interconnect (microblaze_0_axi_intc)

The AXI Interconnect module is vital to connecting GPIO modules like the UART, and AXI GPIO to the master module, which in this case is the MicroBlaze processor. The Interconnect module is also responsible for handling the clock signals and reset signals of each of the GPIO modules by taking in the output Reset signals from the Processor System Reset and the Clock signals from the Clocking Wizard and then distributing them to corresponding GPIOs.

AXI GPIO (axi_gpio and gpio_usb)

The AXI GPIO is the main GPIO used in this experiment and as its name suggests, is a general purpose interface for inputs and outputs. It has either a single or dual channel GPIO channel and a channel width from 1 to 32 bits, with each bit being able to be programmed as either an input or output. It’s used primarily in our case as either the block that takes in inputs from the switches and then displays the output to the LEDs, or for 6.2, acts as the USB Reset, Interrupt, and Keycode_0 and Keycode_1 outputs, letting us communicate with external peripherals.

AXI UART (axi_uartlite_0)

This AXI UART module is an interface that provides UART functionality with our MicroBlaze chip. Namely, outputs from our program can be directly displayed in the Serial Monitor built into Vitis using the print function. This allows for debugging when determining whether or not our code was properly receiving values from the keyboard by displaying the key code that was currently being read.

AXI Interrupt Controller (microblaze_0_axi_intc)

This module ensures that the various interrupts enabled in the design from different devices can be consolidated into a single interrupt output that can then be sent to the INTERRUPT input on the MicroBlaze processor block. In Lab 6.2's case, due to increased number of interrupts, a concatenation block is used in conjunction with the AXI Interrupt Controller to streamline the process of combining all interrupts into a single interrupt signal by taking the various interrupts and then combining them into a [3:0] array that is inputted into the AXI Interrupt Controller.

Concat (microblaze_0_xlconcat)

This turns various signals of different widths into a single signal of a fixed width. As mentioned previously, this module is used to combine all of the different interrupt signals from QUAD SPI, GPIO_usb_intr, and axi_uartlite_0 into a single [3:0] output that can be used as the input for the AXI Interrupt Controller.

AXI Quad SPI (spi_usb)

This module is an interface that connects all of the SPI slave devices used in our lab, like the various different USB modules and the MISO (Master In Slave Out) signal from the USB peripheral MAX3421E chip and then connects the signals from the Master, which is the MicroBlaze processor in our case, to be outputted into to the slaves, or the MOSI (Master Out Slave In) signal. This module also handles all the clock signals, and produces its own interrupt signal that is sent to concatenate module and subsequently sent to the AXI Interrupt Controller.

VGA-HDMI IP (HDMI/DVI Encoder)

This module is a provided module that is responsible for taking the produced values from our VGA_Controller.sv file and then transforming it to a HDMI output that can then be used to display the ball bouncing on a monitor. Since this is a provided file, we imported it into our IP Catalog by adding a directory and then double clicking the resulting IP to create an instantiation. For our uses, the Blue, Green, and Red channels all have a data width of 4, allowing a total of 4096 colors.

Module Descriptions of SystemVerilog Files (Appendix)

Top Level (lab622xsa.sv)

Inputs: input logic Clk, reset_rtl_0, usb_spi_miso, uart_rtl_0_rxd

input logic [0:0] gpio_usb_int_tri_i

Outputs: output logic gpio_usb_tri_o, usb_spi_mosi, usb_spi_sclk, usb_spi_ss,

uart_rtl_0_txd, hdmi_tmds_clk_n, hdmi_tmds_clk_p

output logic [2:0] hdmi_tmds_data_n, hdmi_tmds_data_p,

output logic [7:0] hex_segA, hex_gridA, hex_segB, hex_gridB

Description: This is the top level that contained all of the inputs from our IPs, as well as the outputs required to communicate with external HDMI and USB modules. It also contained all of the internal wiring that linked modules to each other.

Color Mapper (color_mapper.sv)

Inputs: input logic [9:0] DrawX, DrawY, RectX, RectY, RectL, RectR, RectU, RectD,
Rect2X, Rect2Y, Rect2L, Rect2R, Rect2U, Rect2D, motion, motionx1,
input logic [3:0] bg_red, bg_green, bg_blue,
input logic clk_25MHz, clk_100MHz, vsync, Reset, punch, punch1, kick, kick1, block,
block1, start screen, fighting, ending, deathL, deathR

Outputs: output logic [7:0] healthR, healthL,
output logic [3:0] Red, Green, Blue

Description: This module was more than just a module that decided what color was used; it also handled the block bar, rage mechanics, critical hits, and health bar logic. To keep the description brief, this module operated using a ton of internal flags depending on what needed to be drawn when at a DrawX and DrawY value. For example, when coloring over the locations of the the health bars, a flag would be set depending on the location as well as the current health of the players. Then, this flag would be used to draw the bar by using a huge if-else chain that checked each flag to determine whether to draw a sprite, health bar, block bar, etc. And if none of the foreground flags were high, it would then draw the background. The health logic was made using a counter that kept track of the hits taken, the type of hits and whether or not the character was enraged; this would then be used to determine the health bar flags. The block bar had a similar counter that would decrement to a minimum value when block was held and then increase when not held. Sprites for

critical hits, animations, and for all character moves were instantiated in this module and then were used depending on what movement was made and which flags were set high.

State Control (states.sv)

Inputs: input logic clk_25MHz, reset_ah, startscreen, fighting, ending, deathR, deathL,

input logic [7:0] keycode, keycode1, keycode2, keycode3, healthR, healthL

input logic [9:0] DrawX, DrawY

Outputs: output logic [3:0] bg_red, bg_green, bg_blue

Description: This module handled all of the logic regarding start screens, end screens and the background. Depending on if the key codes to start the game were pressed, the module would shift from the first state (start state) to the fighting state, which set the background to the fighting arena. But if either of the characters lost all of their health, the state machine would then move to the end screen state, which would output the end screen as the background.

Control for Left Player (ball.sv)

Inputs: input logic Reset, frame_clk, deathR, deathL, startscreen

input logic [7:0] keycode, keycode2

input logic [9:0] Rect1X, Rect1Y, Rect1WidthLeft, Rect1WidthRight,

Rect1HeightUp, Rect1HeightDown

Outputs: output logic signed [9:0] RectX, RectY, RectWidthLeft, RectWidthRight, RectHeightUp,

RectHeightDown, motion

output logic punch_flag, kick_flag

Description: This module handled all of the position, motion, and collision logic for the left character. The basis of this module was an combinational section and an always_ff section that updated the character position depending on the motion on the clock edge. Depending on the key codes entered, this module would assign a X and Y motion value that would be applied to the position, while gravity directly updated the Y motion so that it would decrement if the object was still in the air. This module also handled all of the boundary logic to make sure that the sprites stayed within bounds, and also implemented collision logic features that allowed us to make it so that characters would vibrate when colliding with each other and moving opposite ways. This module also kept note of kick and punch keycodes and outputted them to color mapper to be used to animate punches and kicks.

Control for Right Player (ball1.sv)

Inputs: input logic Reset, frame_clk, deathR, deathL, startscreen

input logic [7:0] keycode, keycode2

input logic [9:0] Rect2X, Rect2Y, Rect2WidthLeft, Rect2WidthRight,

Rect2HeightUp, Rect2HeightDown

Outputs: output logic signed [9:0] RectX, RectY, RectWidthLeft, RectWidthRight, RectHeightUp,

RectHeightDown, motionx1

output logic punch_flag, kick_flag

Description: This module was similar to the ball.sv module, except it checked different key codes since the right player responded to different inputs. The overall idea is the same, it controls all of the X and Y motion, while recording all of the positions of both rectangles in order to calculate the next positions depending on the current motion in the X and Y directions. This module also allowed

us to build gravity physics into our game by setting a constant value to decrement any Y motion by while also helping account for collisions between characters.

Shadows (createshadows.sv)

Inputs: input logic [9:0] DrawX, DrawY, RectX, RectY, RectWidthLeft, RectWidthRight, RectHeightUp, RectHeightDown, Rect2X, Rect2Y, Rect2WidthLeft, Rect2WidthRight, Rect2HeightUp, Rect2HeightDown,

input logic clk_25MHz

Outputs: output logic [3:0] shadowL_red, shadowL_green, shadowL_blue, shadowR_red, shadowR_green, shadowR_blue

Description: This module took advantage of the fact that we can shade the actual background colors using logic; by subtracting 4'h2 from the RGB values, we can actually dim an area. However, the shape had to be achieved by using a sprite; two 1 bit sprites were used, one for when the character was jumping, the other when the character was on the ground. Sprite positions were input to the module in order for the shadows to track the characters movements in the X direction and any movements in the Y directions would immediately change the shadow to the smaller one. We built the shape by checking for the background color in the sprite COE; if the color was the color we set as background (FFF), the color mapper would ignore this and set the RGB output to be the background, else, it wold subtract 2 from each RGB color to dim it. Of course, logic to ensure that the color values didn't go negative were also in place to make sure the dimming didn't go too far.

Walking for Left Player (d_walking.sv)

Inputs: input logic [9:0] motionx1, DrawX, DrawY, spritX, spriteY, spriteWidthL, spriteHeightU,

input logic [7:0] healthL,

input logic vga_clk, vsync

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

Description: In order to achieve the animation of walking, we decided to use just one extra frame in order to save on BRAM. The idea behind this was the same for both left and right players, but the sprite instantiations were different in each walking module, since different sprites had to be used for different players. Basically, we had several different sprites, one for standing still, the other for mid-frame standing, then another for mid-frame walking, and one more that replaced the standing still sprite with a raged version. Then, depending on a counter value, the RGB values outputted by this module would change between the still frame and the mid-frame. We also had to use health in our logic in order to decide when to move into rage mode, and motion information to decide when to use the walking vs the standing sprite. The clock used for animation was important, since we couldn't use the vga_clk because it was too fast; instead we synced up the counter to the vsync clock in order to count frames. This module was for the left side, for Doppo.

Walking for Right Player (b_walking.sv)

Inputs: input logic [9:0] motionx, DrawX, DrawY, spriteX, spriteY, spriteWidthL, spriteHeightU,

input logic [7:0] healthR,

input logic vga_clk, vsync

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

Description: In order to achieve the animation of walking, we decided to use just one extra frame in order to save on BRAM. The idea behind this was the same for both left and right players, but the sprite instantiations were different in each walking module, since different sprites had to be used for different players. Basically, we had several different sprites, one for standing still, the other for mid-frame standing, then another for mid-frame walking, and one more that replaced the standing still sprite with a raged version. Then, depending on a counter value, the RGB values outputted by this module would change between the still frame and the mid-frame. We also had to use health in our logic in order to decide when to move into rage mode, and motion information to decide when to use the walking vs the standing sprite. The clock used for animation was important, since we couldn't use the vga_clk because it was too fast; instead we synced up the counter to the vsync clock in order to count frames. This module was for the right side, for Baki.

Example Module for Generic Sprite (sprite_example.sv)

Inputs: input logic [9:0] DrawX, DrawY, spriteX, spriteY, spriteWidthL, spriteHeightU

input logic vga_clk, blank

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

Description: This module handled all of the color calculations to decide which RGB values to output when the foreground was chosen; instead of mapping the COE file to the entirety of the screen like the background module though, we had to tweak the indexing for this module to fit the 90 x 150 sprites we were using. The overall concept was that the example file would take in drawX and drawY information, as well as the position of the sprite and the dimensions, and then calculate the current location in memory of the pixel we needed to choose. Then, the example module communicated with the palette to convert the index stored at the memory location to a RGB value

to be outputted to color mapper. We had multiple different sprite BRAM instantiations with different COE files, but the code for the example file was the same throughout.

Palette Module for Generic Sprite (sprite_palette.sv)

Inputs: input logic [2:0] index

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

Description: Since we had BRAM constraints, the sprite palette only consisted of 8 bit colors, half of the background. The concept is the same as the background palette, each index corresponds to a different color that has already been defined using the Image to COE tool and is outputted from the example.sv module to the color mapper module.

Example Module for Background (bg_example.sv)

Inputs: input logic [9:0] DrawX, DrawY,

input logic vga_clk, blank

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

Description: This module handled all of the calculations for which color was to be used when displaying the background. The reason this is a separate module is because the size of the background is different from the size of the sprites, and as such, the indexing had to be changed in order to accommodate the change in dimension, this way we could map the entirety of the screen (640 x 480) using the COE file.

Palette Module for Background (bg_palette.sv)

Inputs: input logic [3:0] index

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

Description: This palette had a 4 bit index to support the use of 16 colors, and was specifically used with the background example module in order to provide the correct colors depending on the index, this way allowing for much reduced memory usage instead of using the full 12 bits of color.

Hex Driver (hex driver.sv)

Inputs: input logic [3:0] in[4],

input logic clk, reset

Outputs: output logic [7:0] hex_seg,

output logic [3:0] hex_grid

Description: The HexDriver module is designed to convert 4-bit inputs into hexadecimal display outputs for a 7-segment display. It handles four inputs, encoding them into segment patterns for digits 0-9 and letters A-F. The module cycles through each input based on a counter, displaying one at a time by activating the corresponding segment of the hex display. This design ensures that all four inputs can be visually represented on a single 7-segment display through multiplexing. This is facilitated by clock and reset signals. It also utilizes a submodule for nibble-to-segment encoding and a counter for display multiplexing, ensuring seamless digit representation. For the final project, we mainly used this module to display the key codes when we were debugging the key code logic, and we also used it to display health when debugging health calculations.

VGA Controller (vga_controller.sv)

Inputs: pixel_clk, reset

Outputs: output logic [9:0] drawX, drawY,

output logic hs, vs, active_nblank, sync

Description: This modules coordinates all of the horizontal, vertical and clocking data to display the right dimension of image at the correct frame rate. The output of this module is routed into a given IP module, vga_to_hdmi, which converts the VGA outputs into HDMI outputs that can be displayed through monitor. The hs and vs outputs refer to the horizontal and vertical synchronization pulse that determines how fast the vertical and horizontal pixels are updated to ensure clean output waveforms, while the active_nblank output determines when the display is cleared blank. The vs output is especially important since our animations ran at the frame rate speed to have the smoothest and best synchronized animations; this output was linked to the walking modules for use with the walking animation and idle animation. The drawX and drawY outputs determine the horizontal and vertical coordinates of our current pixel and is also used extensively in color mapper to determine which color to color a pixel.

Module Descriptions of Software (C) Files (Appendix)

lw_usb_main.c

This was the main software file that handled all of the communication and information from the MAX3421E chip on the Urbana board. It used defined functions to read information from the chip that came from the keyboard and store them in key code buffers to be used; the only module that we changed here was the main file in order to allow for multiple key codes to be used at once. To do so,

we used both channels that were available instead of the singular one (which we would use in the labs). This way, we were able to use 6 key codes all at once, so that both characters could do multiple moves at the same time, allowing for a much more fluid game overall. This was done by using the printHex function and assigning the keyboard buffer values partially to channel one and channel two as well.

Design Statistics

Design Statistic	Ultimate Fighter Design's Usage
LUT	4908
DSP	38
Memory (BRAM)	74.5
Flip-Flop	3271
Latches	0
WNS	-1.446
Frequency ($\frac{1}{T - WNS}$)	87.4 MHz
Static Power	0.079 W
Dynamic Power	0.434 W
Total Power	0.513 W

Figure 13. Final Project design statistics table

Errors and Corrections

Errors Encountered	Correction Measures
Player Movement Physics (Collisions and Gravity)	After figuring the color and control logic, when we moved onto building the physics of the game by instantiating a gravity setting to control the motion of the rectangles, and working through a tedious process of debugging that we would perform throughout the project when our collision logic broke, which happened at multiple points during our feature implementation process. Eventually, instantiating and tracking the other players' DrawX and DrawY coordinates helped in solving collision logic. Gravity logic was implemented as a constant downward force in Y direction.
Health Bar Logic	The most difficult logic part of the project was the health bar logic; though simple in theory, with us only needing to overlay a rectangle over a hardcoded health bar, the actual process of storing the health information and updating it took a lot of time to perfect. It required extensive if statements, with the else statements being extremely important as well, since we had a lot of issues with edge cases and unexpected cases not being accounted for. Also, since the clock used here was so high, we needed a counter system to only reduce health by a limited amount when the characters were hit. Once the basics for health recording and hit detection were in place, we could move onto also implementing kicks, which did more damage, as well as a rage mode that did more damage once a certain amount of health was reached.

State Machine glitches	The state machines we implemented achieved to provide logic for a start screen and an end screen, which used a state machine that switched between 3 states depending on the key codes entered. Initially, the Color Mapper logic of drawing the health bars, character sprites etc overlapped onto the game start screen. This issue was solved via individual 1-bit signals such as “start screen”,”endscreen”, “fighting” had to be implemented. Further, “deathL” and “deathR” are 2 signals for the death of the left and right characters respectively, implemented in order to understand which character has been defeated and accordingly display their death sprite.
BRAM usage	Throughout this project, BRAM usage was the most crucial design aspect we struggled to comply with. Having only 75 BRAM blocks available on the Spartan-7 FPGA, we had to ration our BRAM by numerous downscaling of resolution and color grading of the character sprites and background. In order to accomplish the additional features of breathing animations, shadows, kicking, rage mode sprites etc effective and efficient usage of on-chip memory was the most frequent design limitation we faced and had to deal with.

Figure 14. Errors and corrections table

Conclusion

During the process of coding and debugging our final project, we were introduced to what Vivado and industry standard FPGAs can really accomplish. In retrospect, the ability for us to build an entire arcade game, and not just the logic dictating the possible moves and overall gaming experience but all of the underlying code that controls background and foreground generation, collision mechanics, and state machines that allow the game to work was an astounding result.
We first prioritised to get to baseline difficulty to ensure a steady project progress within the limited time of 3 weeks.

Once we achieved game start, fighting and game over logic with health bars implemented, we implemented extra features with idle animations, rage animations, and variable-sized shadows for the characters. Through trial and error, we were also able to build an efficient work system that catered to both of our schedules and skillsets, with a slow but methodical process of building our project through alternating processes of writing new code and debugging while catering to each of our individual skillsets. We ensured to keep Git backups of our project to ensure the ability of reversion to a previously working model throughout the project.

The freedom of having full autonomy over our project's design meant that we were free to explore whatever features we wanted and implement them, which let us truly understand and appreciate the significance of our labs in this course and ECE courses as well.

Overall, this final project gave us an avenue to utilise the full potential of SystemVerilog and Digital Programming, allowing us to not only understand the reasoning behind what we had learnt, but also get a practical understanding of the industries we could potentially work in the future.