ECE 385

Spring 2018

Final Project

Fighting Video Game in SystemVerilog

Jerry Chen & Noah Mathes

Section ABG (Wed 12:00 – 2:50pm)

Xinheng Liu

Introduction

The purpose of this lab is to create a fighting video game in system verilog and SOC. The software on the SOC interacts with the keyboard and controls the player1's motion. The hardware switches and buttons control the player2's motion and game-state transitions. The hardware is also responsible for hitbox detection, life counting, automatic transitions, and graphics.

Description of System

Description of Software Processes

From our lab8 report, Universal Serial Bus (USB) is a standard that defines the communication protocol between a computer and an electronic device. UsbWrite and UsbRead are functions to help NIOS and the USB interface with the memory. They make function calls to IO_write and IO_read to modify the HPI register's Data, Address, and Status. On a UsbWrite, the HPI_ADDR and HPI_DATA will be filled with provided data. On a UsbRead, the HPI_ADDR will be filled then it will signal the memory to read from the provided address to retrieve the data. This description of UsbWrite and UsbRead matches Figure 2.1.

To go into more detail, IO_write will fill the previously stated registers, select the chip, signal a write and wait for the process to finish. Then it will invert the signals it just sent, specifically, it will turn off the write operation and unselect the chip. The same process happens with IO_read with one slight modification. IO_read, after the operation, will copy the values in HPI_DATA into a local register. The purpose of this is to have the data persist since we don't know what will happen to the register after we turn off the read operation.

Description of Hardware Processes and Operations

Character Controls

Up: The character jumps at a fixed speed and comes back down at a max height. The rate at which the character moves in the y direction is twice as fast as the x direction.

Left: The character moves to the left in the x direction. For player1, the character can not move past the left side of the screen. For player2, the character cannot move past the current position of player1.

Right: The character moves to the right in the x direction. The same rules apply as the Left controls except in the opposite direction.

Shoot: The players can trigger an event to have the projectile become active and display on the screen. The projectile will travel in the x direction towards the opposing player until it hits or runs off the map.

Basic

Multi-State: The game operates in stages. In the beginning the game is on a start screen. The battle screen doesn't activate unless triggered to do so. The battle screen contains all the actions of the game. When player1 dies, the defeat screen displays. When player2 dies, the victory screen displays. This is with the assumption that player1 is the protagonist.

Score-Counter: The score counter for this game is based on the health bar. Unlike other arcade games, fighting games don't have a point system that can be racked up. Instead when a player's health goes to 0, they lose.

Physics

Projectiles: Regarding basic overlapping, the projectiles and character creates the basic overlapping when triggered or contacted. The sprites are deigned to have the character sprites appear on top of the projectile sprites.

Collision Detection: In this fighting game there's projectile collision and character to character collision. Both collision logics are basic. For projectile to character, the logic for mapping pixels was adopted and used for hitbox detection. If the projectile's coordinates are within the bounds of the character's rectangular hitbox then the contact bit will be turned on, triggering a couple events, one being deactivating the projectile to disappear from the screen, another event is reducing the character's hit points by 1.

Graphics

Bitmap Sprite: For this game, we have bitmap sprites for the fireball. The bitmap for this sprite includes a palettize design where index 0 represents empty/transparent pixels. All other indices will represent an existing pixel that needs to be drawn, which is also used to index the palette.

PNG to MIF: We adapted Rishi's github python script. Its original purpose was to read all the pixels in the PNG, and output a 24-bit hex representation of each pixels. Since we didn't want to deal with buggy software for the RAM memory, we decided to use on-chip memory, and there isn't much we can deal with, so instead of loading 24-bit pixels for the whole background and character sprite, we decided to change the purpose of the python script. We sampled the original image for 8-16 colors. These colors are inputted into the script. The script will read all the pixels from the original PNG then it will round all the pixels to one of the sample colors using a simple distance formula. The script will then output an index for each color including 0 for white/transparent pixels. The background was loaded from a background.txt file and the sprite was loaded from a char.txt file.

Text/Font Sprite: The text/font sprites are created manually similar to the bitmap sprite. We followed the font sprite tutorial on the final projects page. Since font colors are basic, we only used 0's for transparent and 1's for colored pixels.

<u>Description of Hardware/Software Interface</u>

Interface

Keyboard: The keyboard interface is the same exact functionality as the one from lab8. The keyboard software communicates with the hardware modules through a keycode array. When a key is pressed, the keycode is loaded with the 8-bit encoding of the keypress.

Block Diagram

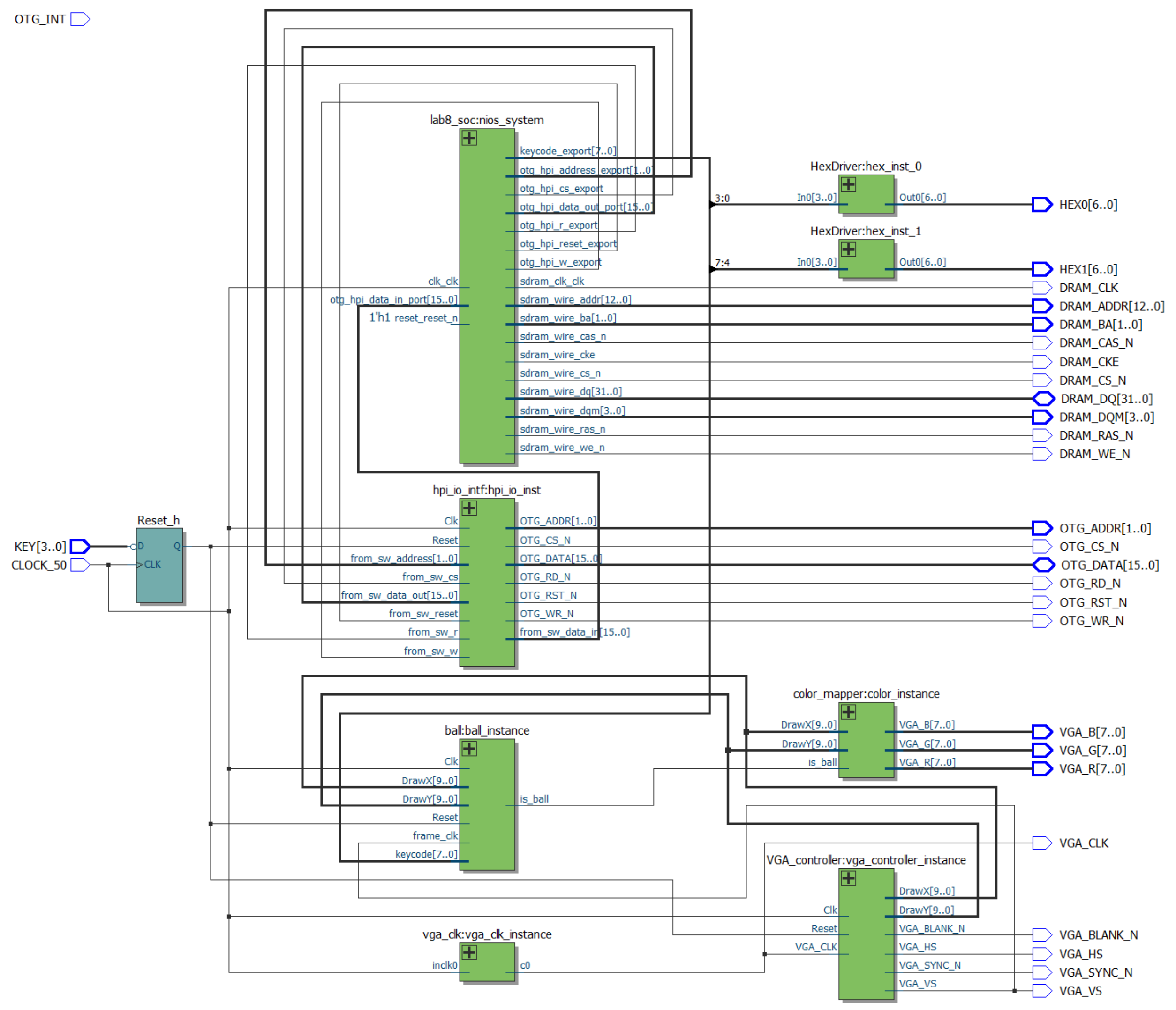


Figure 2.1: Lab 8 block diagram.

The lab8 diagram above is a simplified version of block diagram containing only the essential modules. All the addition modules shown in the below diagrams incorporates the graphic modules, projectils, and collision detection. As you can see the final block diagram is messy and difficult to read. With the Lab8 diagram it’s easy to gain a basic understanding of what the layout looks like.

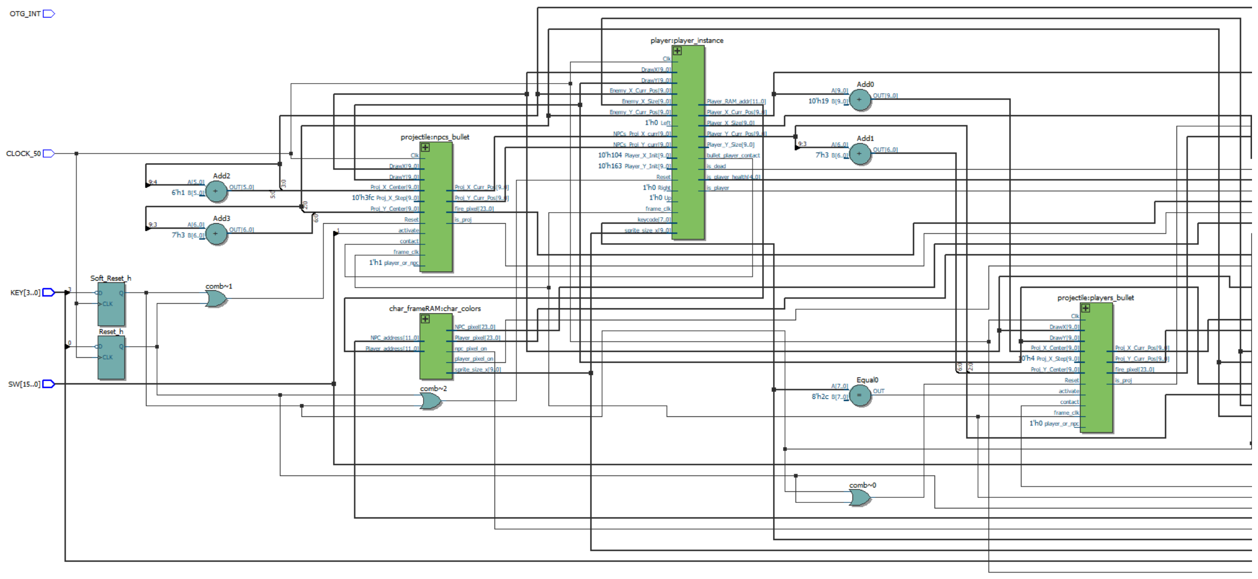


Figure 2.2: Left side of top level block diagram

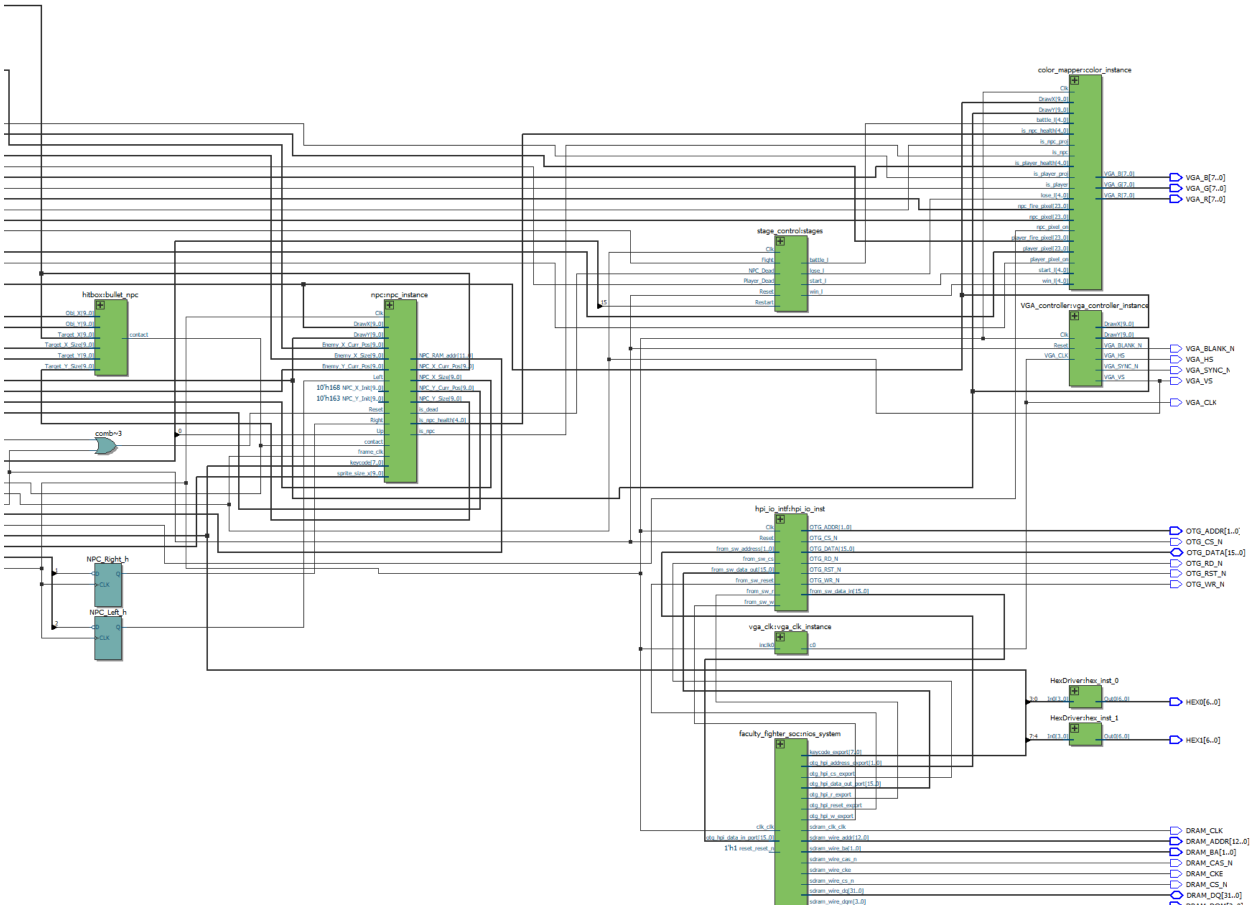


Figure 2.3: Right side of top level block diagram

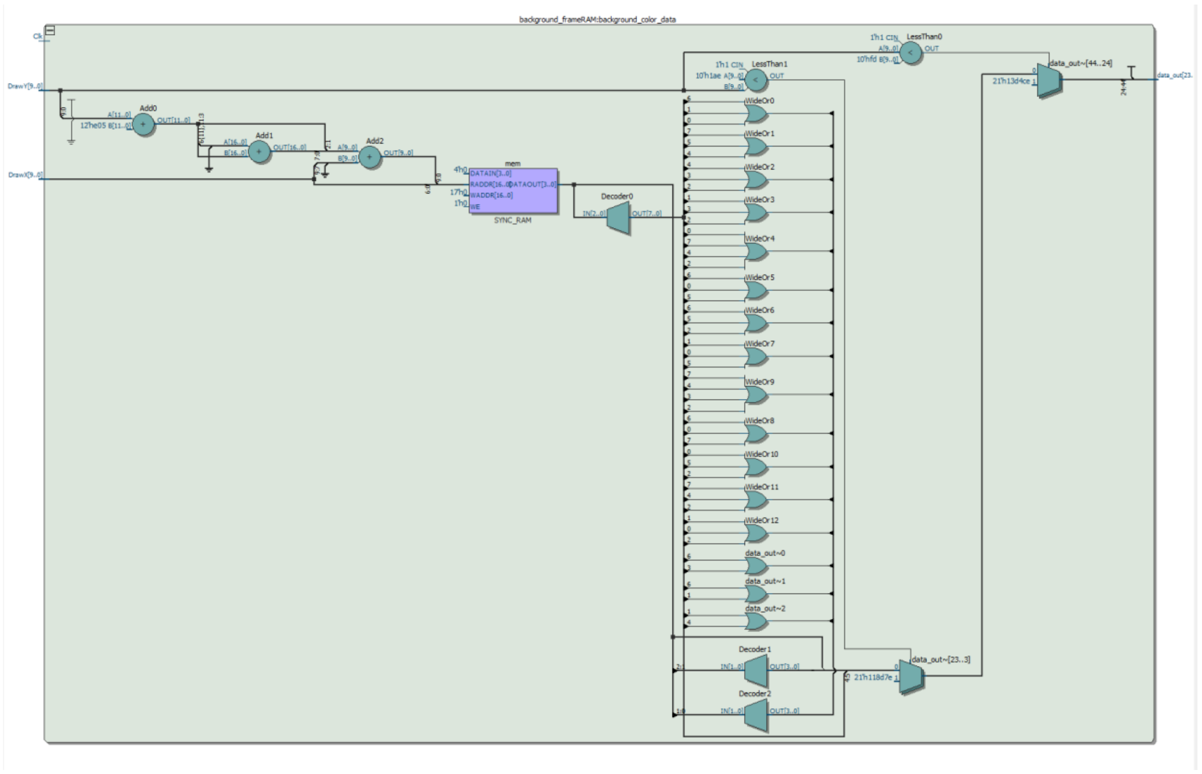


Figure 2.4: Block diagram for background graphics

Short annotation for Figure 2.4, the module read part of the screen's pixel data from a file. The rest of the screen was hardcoded. This saves a lot of time during compilation since Quartus doesn't need to synthesis a full 640x480 size block of on-chip memory. So we programmed the pixels above a y coordinate a shade of blue, and below another y coordinate a shade of green. We then filed the middle ground with the palette for the clouds, ocean, and shadows. The end result produces a pixel value for the current pixel coordinate.

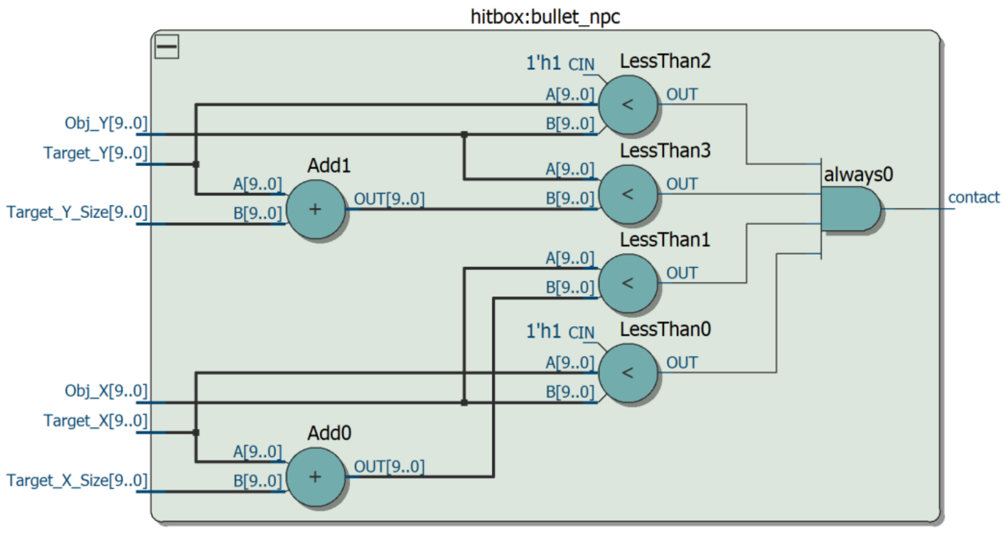


Figure 2.5: Hitbox block diagram

State Diagram

Some annotations for the state diagrams starting with the stage control. Stage control starts in the START state. This state signals the Color Mapper to display the start screen. From any state, whenever the player signals a restart/reset, it will return to this START state. When the player signals the battle/fight signal. During the BATTLE state, the actual game play will occur. When player1 dies, the BATTLE state will transition to the LOSE state. When player2 dies, the BATTLE state will transition to the WIN state. From both states, when the player triggers the restart signal, which happens to be the battle/fight signal, they will transition to an intermediary state, which immediately transitions to the START state after the restart signal is released. This allows the button to be reused as a battle/fight signal.

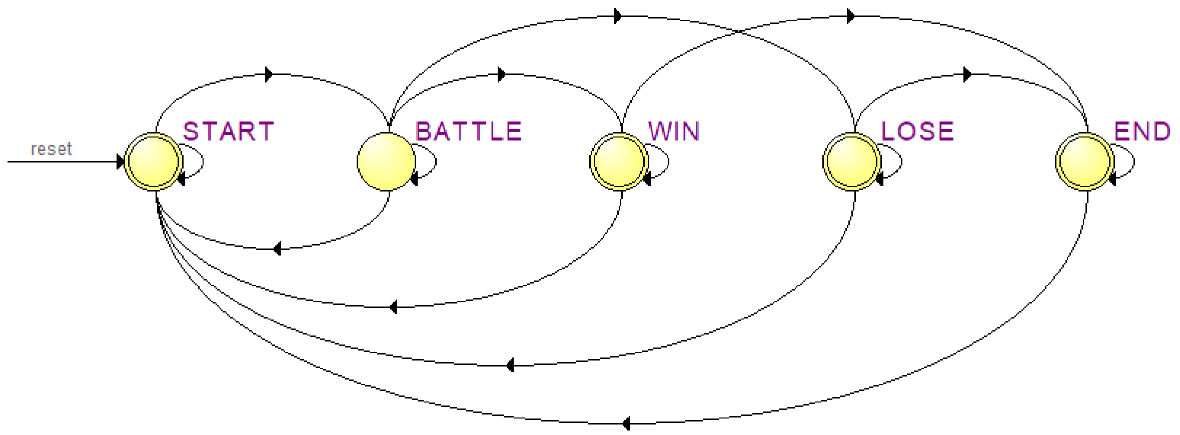
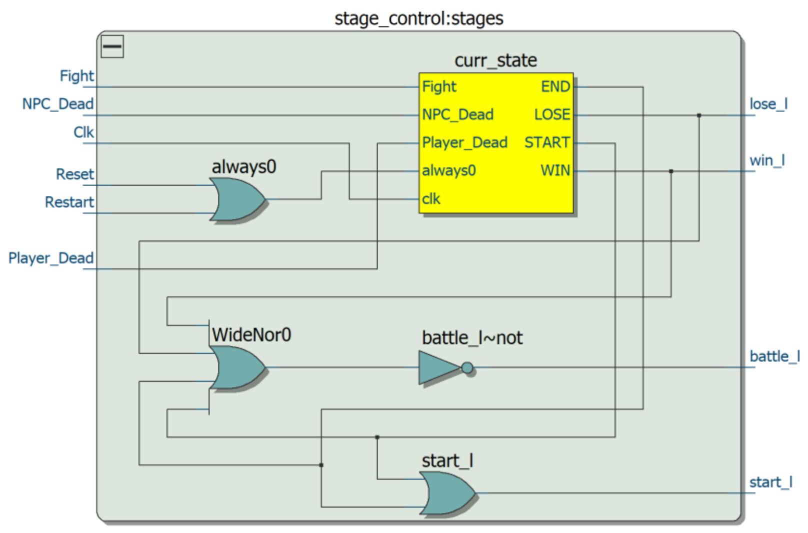


Figure 2. : Stage Control State Diagram



Module Descriptions

Module: faculty_fighter_top_level.sv

Inputs: CLOCK_50, [3:0] KEY, [15:0] SW, OTG_INT

Outputs: [6:0] HEX0, [6:0] HEX1, [7:0] VGA_R, [7:0] VGA_G, [7:0] VGA_B, VGA_CLK, VGA_SYNC_N, VGA_BLANK_N, VGA_VS, VGA_HS, [1:0] OTG_ADDR, OTG_CS_N, OTG_RD_N, OTG_WR_N, OTG_RST_N, [12:0] DRAM_ADDR, [1:0] DRAM_BA, [3:0] DRAM_DQM, DRAM_RAS_N, DRAM_CAS_N, DRAM_CKE, DRAM_WE_N, DRAM_CS_N, DRAM_CLK

Inout: [15:0] OTG_DATA, [31:0] DRAM_DQ

Description: Top level module for the project.

Purpose: Connects all the modules that make up the fighting game.

Module: Color_Mapper.sv

Inputs: Clk, is_player, is_npc, is_player_proj, is_npc_proj, [23:0] player_fire_pixel, [23:0] npc_fire_pixel, [4:0] is_player_health, [4:0] is_npc_health, [4:0] start_l, [4:0] battle_l, [4:0] win_l, [4:0] lose_l, player_pixel_on, npc_pixel_on, [23:0] player_pixel, [23:0] npc_pixel, [9:0] DrawX, [9:0] DrawY

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

Description: This module takes in bits stating whether the current pixel exists, and colors it a provided or predetermined color.

Purpose: This module handles all coloring of the VGA display.

Module: HexDriver.sv

Inputs: [3:0] In0

Outputs: [6:0] Out0

Description: This module converts 4-bit hex value into its 7-segment hex representation.

Purpose: This module is used to display hex values on the board hex LEDs.

Module: VGA_controller.sv

Inputs: Clk, Reset, VGA_CLK,

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

Description: This module produces the coordinates of the current pixel.

Purpose: This module controls the refreshing of the VGA display for vertical and horizontal sync.

Module: background_frameRAM.sv

Inputs: [9:0] DrawX, [9:0] DrawY, Clk

Outputs: [23:0] data_out

Description: This module reads a third of the pixel mapping logic from a text file to create the static frame buffer.

Purpose: This module is used to color the background for the current pixel during battle.

Module: box.sv

Inputs: [9:0] DrawX, [9:0] DrawY, [9:0] Pos_X, [9:0] Pos_Y

Outputs: is_box

Description: This module contains the heart sprite ROM and provides whether the current pixel needs to be drawn for the heart.

Purpose: This module is used with the health bar to interface it with the user.

Module: char_frameRAM.sv

Inputs: [11:0] Player_address, [11:0] NPC_address

Outputs: [9:0] sprite_size_x, player_pixel_on, npc_pixel_on, [23:0] Player_pixel, [23:0] NPC_pixel

Description: This module reads the character sprite from a text file that contains palette indexing data. The text file is generated from a python script that rounds all the pixels to the closest color on the palette.

Purpose: This module is used to map the player onto the screen.

Module: fireball_animation_control.sv

Inputs: Clk, Reset, is_active

Outputs: [3:0] idx_office

Description: This module is a state machine that is synced with the frame clk to offset the indexing of the fireball sprite.

Purpose: This module is used to animate the fireball.

Module: fireball_sprite.sv

Inputs: [9:0] DrawX, [9:0] DrawY, frame_clk, Reset, player_or_npc, [9:0] proj_x_curr, [9:0] proj_y_curr, fire_active,

Outputs: [9:0] sprite_size_x, [9:0] sprite_size_y, fire_pixel_on, [23:0] fire_pixel

Description: This module contains the fireball sprite ROM data.

Purpose: This module is addressed by the projectile module to retrieve the colors of the current pixel, which is fed into colormapper.

Module: font_rom.sv

Inputs: [9:0] addr

Outputs: [7:0] data

Description: This module contains the font sprite ROM data for alphanumeric characters for 0-9, A-Z, and a-z.

Purpose: This module is addressed by colormapper to retrieve the bitmap data for each character.

Module: health.sv

Inputs: [9:0] DrawX, [9:0] DrawY, [9:0] Health_Pos_X, [9:0] Health_Pos_Y

Outputs: [4:0] is_health

Description: This module provides positioning and coloring data of the health points.

Purpose: This module is used with ColorMapper to display hearts on the screen. To interface how much health the players have.

Module: hit_once_control.sv

Inputs: Clk, Reset, contact, triggered

Outputs: hit

Description: This module is a state machine that clocks contact.

Purpose: This module is used to make sure that contact from an object and a character's hitbox is only triggered once.

Module: hitbox.sv

Inputs: [9:0] Obj_X, [9:0] Obj_Y, [9:0] Target_X, [9:0] Target_Y, [9:0] Target_X_Size, [9:0] Target_Y_Size

Outputs: contact

Description: This module calculates whether the object is within the frame of the target.

Purpose: This module is used to create a collision element between any two things.

Module: hpi_io_intf.sv

Inputs: Clk, Reset, [1:0] from_sw_address, [15:0] from_sw_data_out, from_sw_r, from_sw_w, from_sw_cs, from_sw_reset

Inout: [15:0] OTG_DATA

Outputs: [15:0] from_sw_data_in, [1:0] OTG_ADDR, OTG_RD_N, OTG_WR_N, OTG_CS_N, OTG_RST_N

Description: This module interfaces data with HPI.

Purpose: This module assists with the logic for USB.

Module: npc.sv

Inputs: Clk, Reset, frame_clk, [9:0] NPC_X_Init, [9:0] NPC_Y_Init, [9:0] Enemy_X_Curr_Pos, [9:0] Enemy_Y_Curr_Pos, Enemy_X_Size, Up, Left, Right, contact, [7:0] keycode, [9:0] DrawX, [9:0] DrawY, [9:0] sprite_size_x,

Outputs: [9:0] NPC_X_Curr_Pos, [9:0] NPC_Y_Curr_Pos, [9:0] NPC_X_Size, [9:0] NPC_Y_Size, [4:0] is_npc_health, [11:0] NPC_RAM_addr, is_npc, is_dead

Description: This module contains all the controls and logical position and coloring data for the npc character.

Purpose: This module is used to create an adversary for the player to battle against.

Module: player.sv

Inputs: Clk, Reset, frame_clk, [9:0] PLAYER_X_Init, [9:0] PLAYER_Y_Init, [9:0] Enemy_X_Curr_Pos, [9:0] Enemy_Y_Curr_Pos, Enemy_X_Size, Up, Left, Right, contact, [7:0] keycode, [9:0] DrawX, [9:0] DrawY, [9:0] sprite_size_x, [9:0] NPCs_Proj_X_curr, [9:0] NPCx_Proj_Y_curr

Outputs: [9:0] Player_X_Curr_Pos, [9:0] Player_Y_Curr_Pos, [9:0] Player_X_Size, [9:0] Player_Y_Size, [4:0] is_player_health, [11:0] Player_RAM_addr, is_player, is_dead, bullet_player_contact

Description: This module contains all the controls and logical position and coloring data for the player character.

Purpose: This module is used to create a character that player can interact with and fight the npc.

Module: projectile.sv

Inputs: Clk, Reset, frame_clk, player_or_npc, [9:0] Proj_X_Center, [9:0] Proj_Y_Center, [9:0] Proj_X_Step, activate, contact, [9:0] DrawX, [9:0] DrawY,

Outputs: [9:0] Proj_X_Curr_Pos, [9:0] Proj_Y_Curr_Pos, is_proj, [23:0] fire_pixel

Description: This module contains all the controls and logical position and coloring data for the projectiles.

Purpose: This module instantiates a projectile from the perspective of either the player or the npc.

Module: stage_control.sv

Inputs: Clk, Reset, Fight, Restart, NPC_Dead, Player_Dead

Outputs: start_l, battle_l, win_l, lose_l

Description: This module is a state machine that controls the stages of the game based on game data.

Purpose: This module is used to create explicit start screen, battle screen, win screen, and lose screen.

Module: word_FIGHT

Inputs: [9:0] DrawX, [9:0] DrawY, active,

Outputs: [9:0] start_x, [9:0] start_y, [9:0] n, is_word

Description: This module outputs n values that index the start of the desired character in the font sprite ROM.

Purpose: This module is used to spell out the word FIGHT in sequence starting from its start pixel coordinates.

Module: word_DEFEAT

Inputs: [9:0] DrawX, [9:0] DrawY, active,

Outputs: [9:0] start_x, [9:0] start_y, [9:0] n, is_word

Description: This module outputs n values that index the start of the desired character in the font sprite ROM.

Purpose: This module is used to spell out the word DEFEAT in sequence starting from its start pixel coordinates.

Module: word_VICTORY

Inputs: [9:0] DrawX, [9:0] DrawY, active,

Outputs: [9:0] start_x, [9:0] start_y, [9:0] n, is_word

Description: This module outputs n values that index the start of the desired character in the font sprite ROM.

Purpose: This module is used to spell out the word VICTORY in sequence starting from its start pixel coordinates.

Module: word_player1

Inputs: [9:0] DrawX, [9:0] DrawY, active,

Outputs: [9:0] start_x, [9:0] start_y, [9:0] n, is_word

Description: This module outputs n values that index the start of the desired character in the font sprite ROM.

Purpose: This module is used to spell out the word PLAYER1 in sequence starting from its start pixel coordinates.

Module: word_player2

Inputs: [9:0] DrawX, [9:0] DrawY, active,

Outputs: [9:0] start_x, [9:0] start_y, [9:0] n, is_word

Description: This module outputs n values that index the start of the desired character in the font sprite ROM.

Purpose: This module is used to spell out the word PLAYER2 in sequence starting from its start pixel coordinates.

Annotated Waveforms

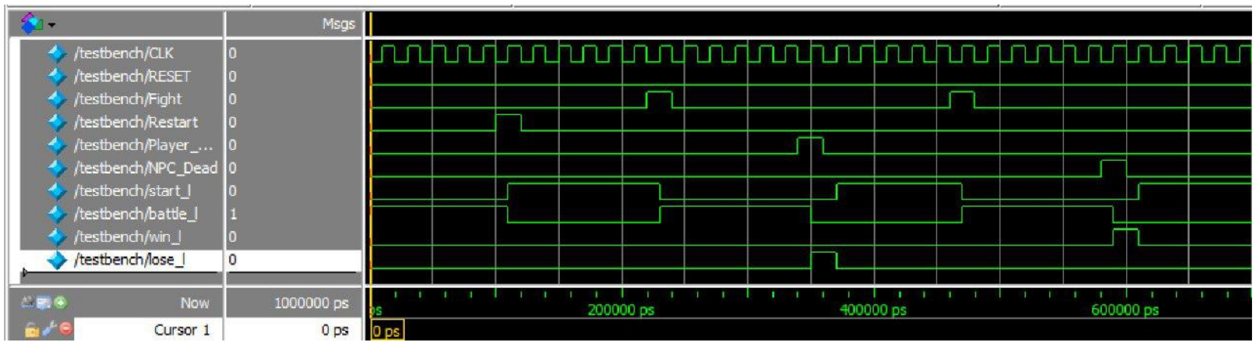


Figure 3. : Waveform for stage_control state machine

This waveform shows that our transitions between game states are working properly. It outputs a high battle_l when the fight activity is displayed, start_l when the start screen is displayed, win_l when the victory screen is displayed, and lose_l when the defeat screen is displayed. As seen in the waveform, automatic transitions to win_l and lose_l occur when the player dies or the npc dies, indicated by the signal names.

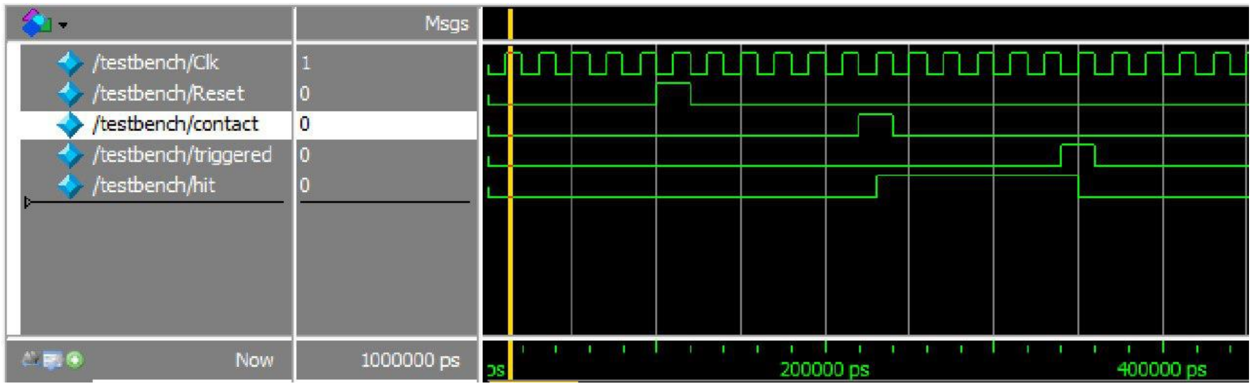


Figure 3. : Waveform for hit_once state machine

This waveform shows that our hit detection state machine is working properly. We faced the issue of hits registering multiple times. Notice the hit signal only becomes high once contact is toggled and remains high until triggered is toggled

Resources and Statistics

LUT	13,921
DSP	N/A

Memory (BRAM)	55,296
Flip-Flop	2,338
Frequency	148.81 MHz
Static Power	105.94 mW
Dynamic Power	.86 mW
Total Power	173.74 mW

Conclusion

In the end although we didn't get a chance to implement all the functionalities we want to create a full-fledge hardware game, we fulfilled most of the features we proposed in the project proposal. We proposed the features for projectiles, hitbox/collision, player health, melee action, combo moves, audio, animations, powerups, character selection, scrolling screen, etc. We focused on the basic ideas of a fighter game and implemented them. The other ones such as powerups and scrolling screen, were disregarded since they don't fit with the theme of the fighting game. Since the sprite that we chose didn't have different forms of the same sprite, we were unable to craft an animation for our character. This also affected our ability to create melee actions.

After this experience, we learned a lot about how the we can build systems from smaller hardware components. While building the game, we tried to modulate as much as we could. We broke the features into smaller parts and then we tested it after integrating it with the current state. This made sure that we had as little errors as possible throughout the process. For future students, I highly recommend this approach since keeping the modules simple also removed the necessity of test benches for most of the modules we designed.

We recommend all the tutorials that have been provided in the final project. Rishi's github was useful in understanding how we can load data from a file into the on-chip memory.