ECE 385

Spring 2023
Final Project Report
Akhil Bonela, Jay Nathan
Shitao Liu

Introduction

Our final project was a simple hardware recreation of the arcade game Astro Fighter. The player controls a spaceship at the bottom of the screen with a keyboard. The enemies attack in waves with the same type of ship. Each enemy ship type has different movement logic and color. The game ends when the player dies, the enemy wins, or the player defeats 10 waves.

Written Description

Astro Fighter has multiple modules that describe the behaviors of each game feature. Since the game uses a VGA controller with elements that move around, we used Lab 6 as a base for our code. The VGA controller is identical, and the color mapper is an expanded version of the one used in Lab 6. The modules that control the player and enemy movement also take inspiration from ball.sv. The characters and text that are shown on screen are custom-made and from the FONTROM.

The NIOS II processor establishes a connection with the MAX3421E USB chip through a USB interface. This connection facilitates the transfer of data between the processor and the USB chip, allowing the processor to control external USB devices like keyboards. Moreover, the NIOS II processor sends video signals via the VGA output port to VGA components, including connected monitors.

The state machine decides what screen to show, as well as the current enemy wave. State 0 is the start state and asks the player to press the enter key. States 1 to 10 represent the different enemy waves. In states 1,2,3, and 6, enemy A attacks; enemy B attacks in states 4,5, and 8; and enemy C attacks in states 7, 9, and 10. Each state has a different number of enemies positioned in different locations on the screen. Finally, states 11 and 12 are the ending states. State 11 will display the “you win” screen, and this only occurs when all enemies in wave 10 are destroyed. State 12 will display the “you lose” screen when the player dies.

State Machine:

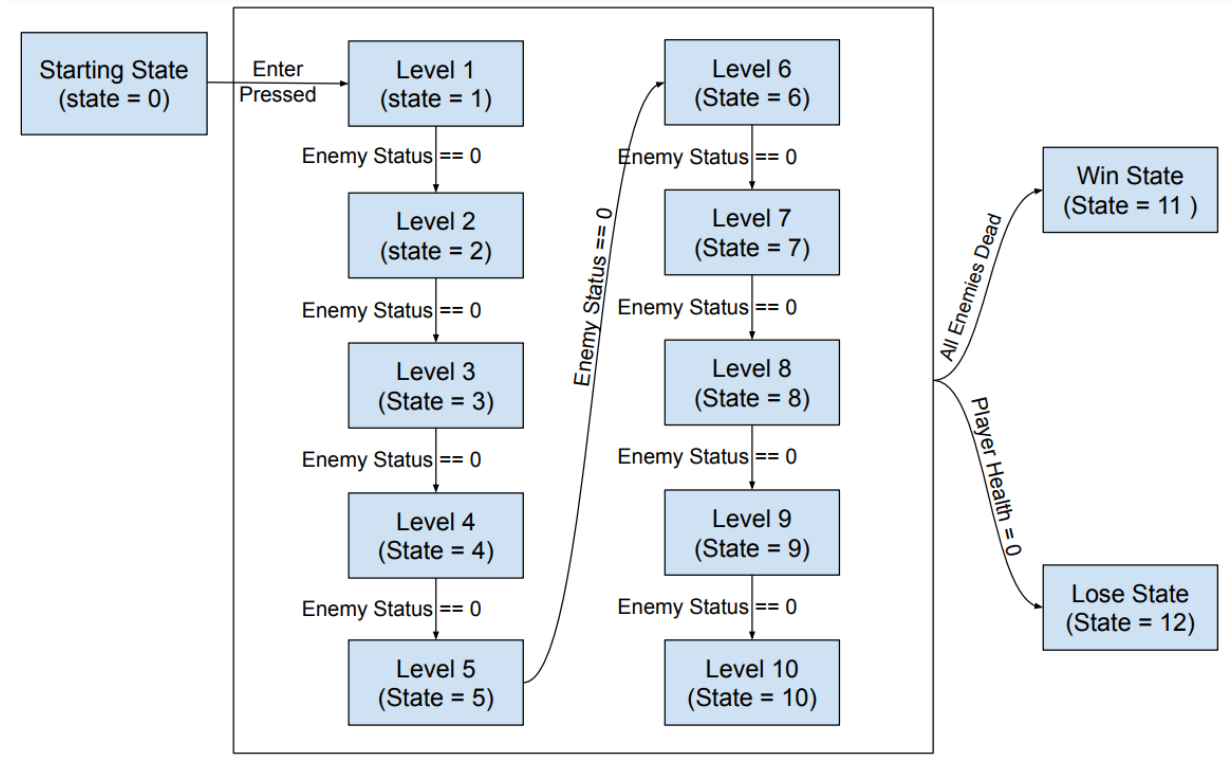
Enemies are placed in an array and move together as one. The array is filled with an assortment of 1s and 0s, where the 1s represent an alive ship. The enemy closest to the player will shoot a missile that will take 20 points from the player's health. Once all the enemies in the array are destroyed (the array is full of 0s), the state will transition to the next state and the array will be reinitialized with a new assortment of 1s and 0s.

The player logic involves movement, shoot, and health. The player moves horrizontaly at the bottom of the screen. Its positon is tracked and the player won't be able to leave the screen's bounds. The player controls are from the keyboard input: the A key will be for left movement, the D key will be for right movement, and the Space Bar will be shoot.

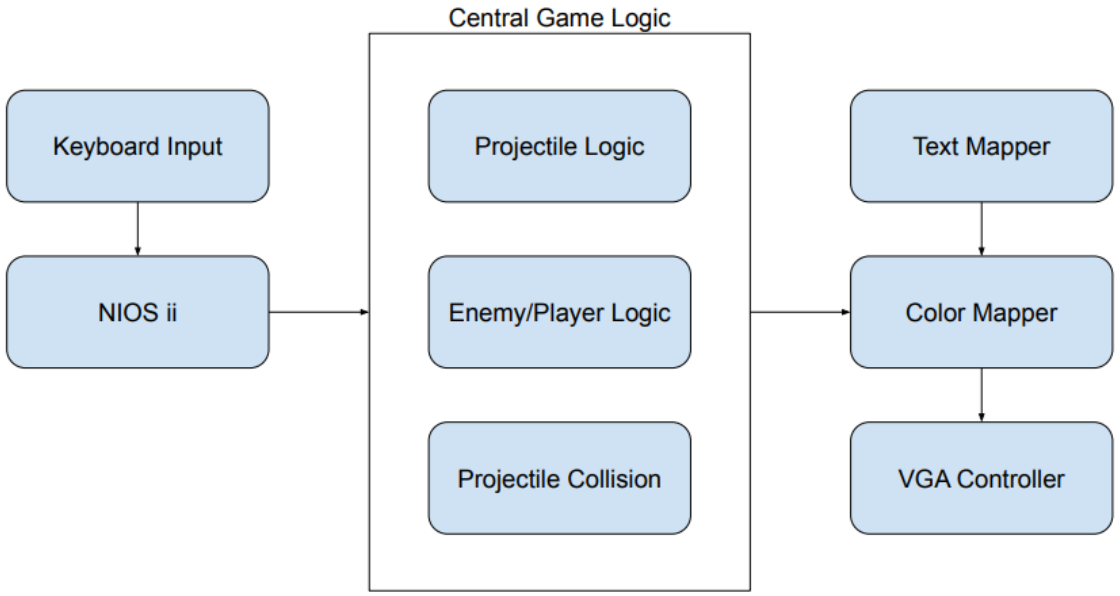
At the top of the screen, there is a yellow bar that displays the player's health and score with a numerical values. The health starts at 100 and decreases by 20 everytime the player gets hit by an enemy missile. Initialy the hundred's digit will be a 1, then it will become a 0. The tens digit starts as a 0, then an 8, and so on. When the tens digit of the health reaches 0, the player will die and the "you lose" screen will be shown. The player also becomes invinsible for a brief interval of time after the ship gets hit. At the top right of the screen, the score is shown. This initially shows the score of 0, but increments by 1 with every enemy that gets destroyed. When the player kills 10 enemies the ones digit will revert back to a 0 and the tens digit will become a 1, 2, 3 and so on.

The color mapper module assigns the appropriate VGA color code to the RGB values of the game. Concurrently, the VGA controller module generates timing signals essential for image display. The timing signals, namely the pixel clock, horizontal sync, and vertical sync, regulate the pixel display rate, the end of each line, and the final frame, respectively. The various modules generate the necessary images each clock cycle, which are subsequently conveyed to the color mapper module to allocate VGA color codes to the sprites and background. Afterward, the VGA controller presents the images on the monitor, ultimately forming the screen of the Astro Fighter game.

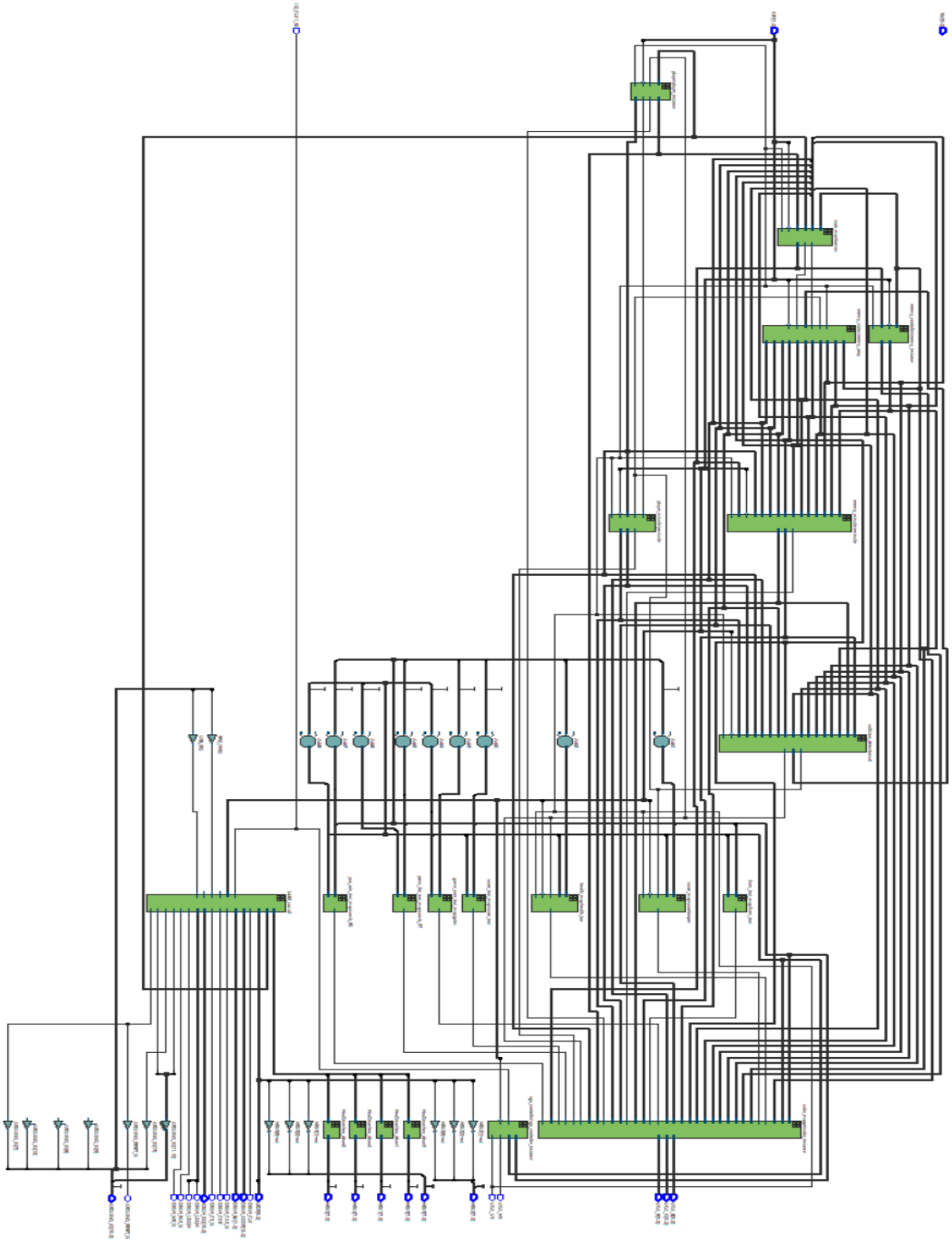
State Diagram



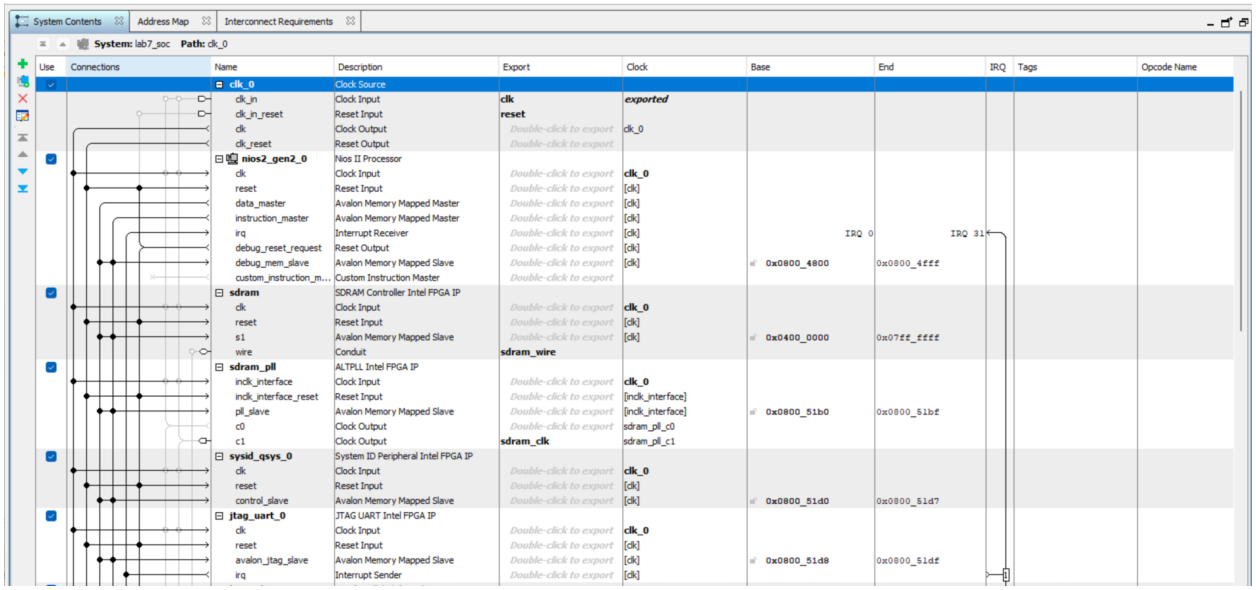
Block Diagram

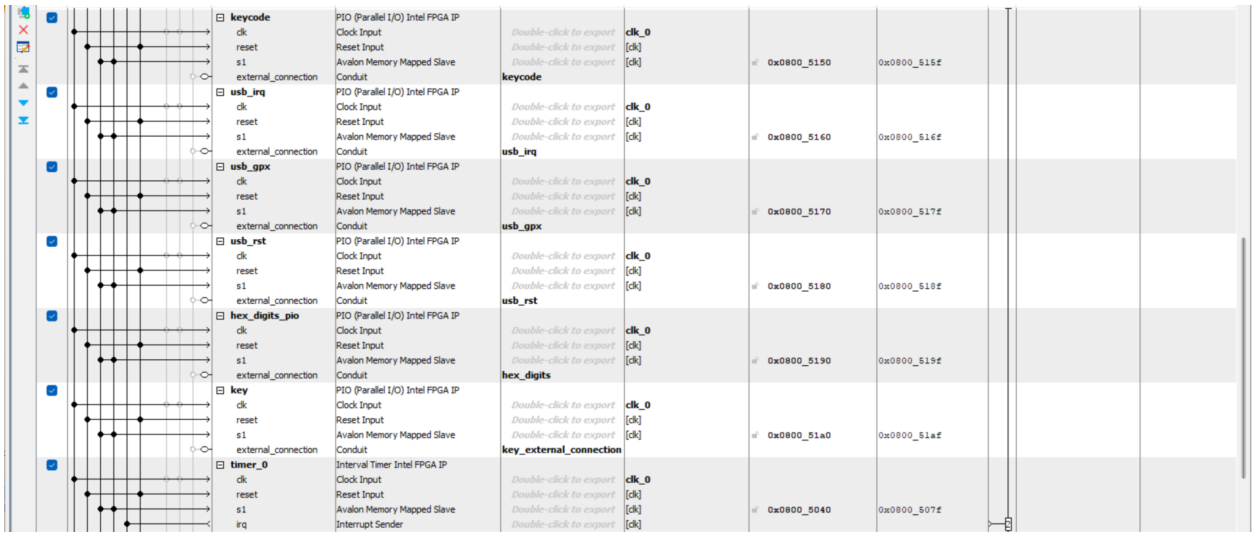


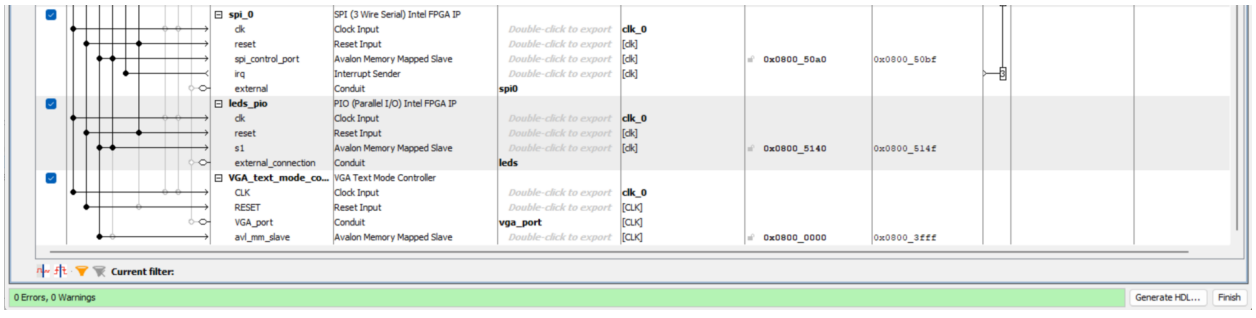
Top Level



Platform Designer







Description of core components

Nios2_gen2_0 - This is the NIOS processor used, here the instructions are sent to other components through the bus

Onchip_memory_2 - The onchip memory on the processor

Sysid_qsys_0 - This component helps to verify the code in our C & Verilog files to assure that the information is being transferred accordingly

Sdram - Offchip memory used by the NIOS, in case of on chip memory being overloaded. It is connected to the instruction and data bus connections

Sdram pll - This is the generated clock for sdram to stabalize, and is connected to sdram by the clk port

Led - This block is for the LED output

Jtag_uart_0 - This tells the processor to access the computers terminal for reading in input for the USB driver and outputting onto eclipse

Keycode - This pio block is how the processor uploads the key code values to the VGA for display logic

Usb_irq - This interrupt(USB) is used by the processor whenever it detects a key press.

Usb_gpx - Interacts with the MAX3421.c file

Usb_rst - Interacts with the MAX3421.c file

Hex_digits_pio - This memory-mapped output shows us the keycode pressed on the hex display

Timer_0 - This is our interrupt timer, where it records the operations run by the usb driver

Spi_0 - This is the SPI peripheral, where we implement the master for the slave file in MAX3421E.c

Description of all .sv modules

Astrofighters.sv:

- Inputs:

```
//////// Clocks ////////
input      MAX10_CLK1_50,

//////// KEY ////////
input    [ 1: 0]   KEY,

//////// SW ////////
input    [ 9: 0]   SW,
```

- Outputs:

```
//////// LEDR ////////
output   [ 9: 0]   LEDR,

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

//////// SDRAM ////////
output             DRAM_CLK,
output             DRAM_CKE,
output   [12: 0]   DRAM_ADDR,
output   [ 1: 0]   DRAM_BA,
inout    [15: 0]   DRAM_DQ,
output             DRAM_LDQM,
output             DRAM_UDQM,
output             DRAM_CS_N,
output             DRAM_WE_N,
output             DRAM_CAS_N,
output             DRAM_RAS_N,

//////// VGA ////////
output             VGA_HS,
output             VGA_VS,
output   [ 3: 0]   VGA_R,
output   [ 3: 0]   VGA_G,
output   [ 3: 0]   VGA_B,


//////// ARDUINO ////////
inout    [15: 0]   ARDUINO_IO,
inout              ARDUINO_RESET_N
```

- Summary: This is the toplevel module that defines various input and output ports for different components such as clocks, keys, switches, LEDs, HEX displays, SDRAM, VGA, Arduino. These ports are used for communication with other

modules or external devices. Inside the module, there are assignments and instantiations of other modules such as "HexDriver," "vga_controller," "enemy_controller," "player," and others. These modules are used for controlling different aspects of the project, such as displaying hexadecimal numbers on the HEX displays, controlling the VGA output, managing enemy entities, player control, collision detection, scorekeeping, etc.

- Purpose: The purpose of this code is to interface between connections from different components/modules of the AstroFighters project. Further enabling their collaboration to create the desired functionality of the game.

vga_controller.sv:

- Inputs:

```
input         Clk,       // 50 MHz clock
              Reset,     // reset signal
```

- Outputs:

```
output logic hs,        // Horizontal sync pulse.  Active low
             vs,        // Vertical sync pulse.  Active low
             pixel_clk, // 25 MHz pixel clock output
             blank,     // Blanking interval indicator.  Active low.
             sync,      // Composite Sync signal.  Active low.  We don't use it in this lab,
                        //   but the video DAC on the DE2 board requires an input for it.
output [9:0] DrawX,     // horizontal coordinate
             DrawY );   // vertical coordinate
```

- Summary: This file the VGA controller that manages timing signals and coordinates for displaying graphics on a VGA monitor(640x480). It utilizes a 50 MHz clock, a 25 MHz pixel clock, and sync pulses (hsync & vsync) to synchronize the display.
- Purpose: This file generates timing signals and coordinates to display graphics on a VGA monitor. It enables easy integration of VGA-based graphics functionalities into this project.

Color_Mapper.sv:

- Inputs:

```
input        [9:0] DrawX, DrawY,
input logic  [3:0] state,count,
input logic        game_over_pixel,
input logic        you_win_pixel,
input logic        press_space_pixel,
input logic        player_unkillable,

input logic  [9:0] enemy_posX,
input logic  [9:0] enemy_posY,
input logic  [9:0] player_pos,
input logic        player_flash,

input logic        missile_exists,
input logic  [9:0] missileX,
input logic  [9:0] missileY,

input logic        pmissile_exists,
input logic  [9:0] pMissileX,
input logic  [9:0] pMissileY,

input logic  [2:0] player_lives,

input logic current_health_pixel,
input logic current_score_pixel,

input logic score_text_pixel,
input logic health_block_pixel,

input logic  [9:0][5:0] enemy_status,
```

- Outputs:

```
output logic [3:0] VGA_R, VGA_G, VGA_B ,
output logic current_sprite_pixel
```

- Summary: This module creates RGB values for each pixel of a VGA display based on the current state and various input signals, such as game status, player and enemy positions, missiles, health, and score. It assigns specific colors to different elements of the game, such as rendering player and enemy sprites, drawing health and score information, and highlighting player and enemy missiles. The module uses a ship-rom to store sprite and ship data, and based

on the current pixel coordinates, it determines the appropriate addresses to retrieve the corresponding color information and sets the RGB values accordingly.

- Purpose: The purpose of Color_Mapper.sv is to mainly handle various visual elements like rendering sprites, messages, etc… By utilizing rom's to retrieve color information and applying it to specific pixels, the module allows for the visual representation of game elements on the VGA display.

Statemachine.sv:

- Inputs:

```
input logic           vs,
input logic [59:0]    enemy_array,
input logic [2:0]     player_lives,
input logic [7:0]     key_pressed,
input logic           reset,
input logic [3:0]     count,
```

- Outputs:

```
output logic [3:0]   state,
output logic       pulse,co
```

- Summary: The module manages the game's state transitions based on various input signals. It includes a state register that determines the current game state, such as the initial "press space" state or different level states. The module also handles special states like "you win" and "game over" based on specific conditions, providing control over the game flow and progression.
- Purpose: The state machine module that controls the state transitions. It takes input signals such as key presses, enemy array status, player lives, and a reset signal to determine the current state of the game. The module assigns different states based on specific conditions, allowing for the progression of levels, handling game over situations, and signaling a win state.

player.sv:

- Inputs:

```
input logic reset,
input logic vs,
input logic [7:0] keycode,
input logic pcollision,
```

- Outputs:

```
output logic pmissile_create,
output logic player_flash,
output logic [9:0] x_pos,
output logic [2:0] health,
output logic player_unkillable
```

- Summary: The module represents a player entity in a game. The module tracks the missile's existence, position, and movement based on user input, as well as collision with enemy projectiles.
- Purpose: The module is responsible for managing the behavior of a player entity in a game. It handles player movement, health, invincibility, missile firing, and position tracking based on input signals and game logic.

player_missile.sv:

- Inputs:

```
input    logic       reset,
input    logic [9:0] playerx,
input    logic       v,
input    logic       create,
input    logic       has_collided,
```

- Outputs:

```
output    logic exists,
output    logic [9:0] playerMissileX,
output    logic [9:0] playerMissileY
```

- Summary: The module is another instantiates and implements the missiles fired by the player. It creates signals to check the existence of the missile, its position, and handles collision detection, missile creation requests(pressing space bar), and movement.
- Purpose: The purpose of the module is to control the behavior of the missiles fired by the player in a video game. It handles the creation of missiles, tracks their existence and position, and updates their movement. Additionally, it manages collision detection with other game objects, ensuring that the missile is destroyed upon collision.

enemy_status.sv:

- Inputs:

```
input     logic reset,
input     logic clk,
input     logic [6:0] enemy_hit,
input     logic collision,
input     logic [3:0] state,
input     logic pulse, co,
```

- Outputs:

```
output    logic [9:0][5:0]  enemy_status,
output    logic [3:0]       count
```

- Summary: The enemy_status module implements an enemy status array that manages enemy units in a game. It updates the status of enemies based on collisions and progresses through different waves of enemies by updating the enemy_status array.

- Purpose: The purpose of the module is to keep track of the status of enemy units in a game. It maintains a column-major array called enemy_status to record which enemies are still alive. The code also includes logic to update the enemy status based on collisions and to progress through different waves of enemies.

enemy_controller.sv:

- Inputs:

```
input logic          vs,
input logic          reset,
input logic  [3:0]   state, count,
```

- Outputs:

```
output logic [9:0]   enemy_posX,
output logic [9:0]   enemy_posY
```

- Summary: The module implements our enemy controller, that is responsible for controlling the overall enemy array in the game. It takes inputs v_sync, reset, state, and count, and provides outputs for the X and Y offsets of the enemy array.
- Purpose: The purpose of the controller module is to determine the movement of the enemy array based on the current state and count values. The code determines the enemy array's movement based on state and count values, using subpixel offset variables for horizontal movement. It updates the position by incrementing or decrementing the x_pos value, handles boundary conditions, and assigns the resulting offsets to output.

enemy_missile.sv:

- Inputs:

```
input    logic              reset,
input    logic [9:0]        playerX,
input    logic [9:0][5:0]   enemy_status,
input    logic              vs,
input    logic [9:0]        enemy_posX,
input    logic [3:0]        state,
```

- Outputs:

```
output   logic          exists,
output   logic [9:0]    missileX,
output   logic [9:0]    missileY
```

- Summary: The module handles the creation, movement, and collision detection of the enemy missile. It creates and tracks a timer to control the rate at which enemies can fire at the player once in range. It sets the missile's X position based on the enemy array's position and a column index, and sets the missile's Y position based on the lowest alive enemy in the column. Additionally it updates the missile's position, and checks if the missile has collided with the player or reached the bottom of the screen.
- Purpose: The purpose of the code is to control the behavior of enemy missiles in a game. It handles the creation of missiles, determines their initial position based on the enemy array and player position, and updates their position while checking for collisions. The module also manages the timing between missile shots and ensures the existence of a single active enemy missile.

collision_detection.sv:

- Inputs:

```
input logic vs,

input logic [9:0] pmissileX,
input logic [9:0] emissileX,
input logic [9:0] pmissileY,
input logic [9:0] emissileY,

input logic [9:0] playerX,
input logic [9:0] enemy_posX,

input logic [9:0][5:0]  enemy_status,
```

- Outputs:

```
output logic [6:0] enemy_hit,
output logic pcollision,
output logic ecollision
```

- Summary: This module takes in various inputs such as missile coordinates, player & enemy positions, and enemy status. It includes internal variables to store calculated values for collision detection. The module uses an always_comb block to calculate and update these variables based on the inputs. It also uses an always_ff block triggered by the vsync signal to check for collisions between player and enemy missiles, as well as between enemy missiles and the player. The module sets the appropriate collision flags based on the detected collisions.
- Purpose: The purpose is to detect collisions between missiles hitting either the player or an enemy. When a collision occurs, the module sets the pcollision flag high for one clock cycle if the player is hit, or sets the ecollision flag high if an enemy is hit.

health_text_rom.sv:

- Inputs:

```
input [7:0] address,
```

- Outputs:

```
output [7:0] data
```

- Summary: This module stores the word “health” in fontrom of text characters. It takes an address input and outputs the corresponding text data.
- Purpose: The purpose of the module is to map the text data from the fontrom to the corresponding pixels on the display. It determines the correct address based on the X-coordinate input and retrieves the text slice for that address.

health.sv:

- Inputs:

```
input logic        vs,
input logic        reset,
input logic [5:0] X,
input logic [3:0] Y,
input logic player_collision,
```

- Outputs:

```
output logic pixel
```

- Summary: The module takes in signals: player collision, X and Y coordinates, and reset. It then uses an always_ff block to decrease the health based on the enemy missile colliding with the player (player collision). Additionally an always_comb block to display the player's health as a number on the screen. The module also includes a number rom to access number symbols to display the current health.
- Purpose: The main purpose of the module is to implement a health map to track the player's health and displays it as a number on a screen. After the player is hit by an enemy projectile, the collision is detected and the player health will decrease.

score_text.sv:

- Inputs:

```
input [7:0] address,
```

- Outputs:

```
output [7:0] data
```

- Summary: This module stores the word “score” in a fontrom of text characters. It takes an address input and outputs the corresponding text data.
- Purpose: The purpose of the module is to map the text data from the fontrom to the corresponding pixels on the display. It determines the correct address based on the X-coordinate input and retrieves the text slice for that address.

score.sv:

- Inputs:

```
input logic       vs,
input logic       reset,
input logic [5:0] X,
input logic [3:0] Y,
input logic enemy_collision,
```

- Outputs:

```
output logic pixel
```

- Summary: The module takes in signals: enemy collision, X and Y coordinates, and reset. It then uses an always_ff block to increment the score based on the player missile colliding with enemies(enemy collision). Additionally, an always_comb block to display the score as a number on the screen. The module also includes a number rom to access number symbols to display the current score.
- Purpose: The main purpose of the module is to implement a score map to track the current score to track how many enemies have been eliminated, and displays it as a number on a screen. After the enemy is hit by an player projectile, the collision is detected and the score will increase.

ship.sv:

- Inputs:

```
input [7:0] address,
```

- Outputs:

```
output [7:0] data
```

- Summary: This module contains parameterized roms that store separate sprites representing different enemies & the player ship; and assigning the address to a ship rom.
- Purpose: This module is a convenient way to store the player and enemy ship models for different levels in the game.

start.sv:

- Inputs:

```
input logic [5:0] X,
input logic [4:0] Y,
input [7:0] address,
```

- Outputs:

```
output [7:0] data
output logic pixel
```

- Summary: The module takes input coordinates X and Y and generates an output pixel based on the value stored in the fire_rom at the corresponding address.
- Purpose: The purpose of the module is to map the input coordinates X and Y to a specific address in the fire_rom and retrieve the corresponding data stored at that address. This data is then used to assign the value of the output pixel. In here the fire_rom is used to store the words Press Enter.

win_map.sv:

- Inputs:

```
input logic [5:0] X,
input logic [4:0] Y,
input [7:0] address,
```

- Outputs:

```
output [7:0] data
output logic pixel
```

- Summary: This module uses a rom to retrieve the words “You Win” based on the input coordinates (X and Y). The rom’s address is determined based on the input coordinates, and the corresponding data is retrieved from the rom and assigned to the pixel output.
- Purpose: This is so once the player wins, after eliminating all the enemies that on the screen it will display you win in green text.

lose_map.sv:

- Inputs:

```
input logic [5:0] X,
input logic [4:0] Y,
input [7:0] address,
```

- Outputs:

```
output [7:0] data
output logic pixel
```

- Summary: This module uses a rom to retrieve the words “You Lose” based on the input coordinates (X and Y). The rom’s address is determined based on the input coordinates, and the corresponding data is retrieved from the rom and assigned to the pixel output.
- Purpose: This is so once the player looses, after loosing all of its health, on the screen it will display you lose in red text.

Design Resources and Statistics

LUT	5614
DSP	0
BRAM	64,512
Flip flop	2,951
Frequency	95.12 MHz
Static Power	96.18 mW
Dynamic Power	1.09 mW
Total Power	106.58 mW

Conclusion

This final project was a challenging and enjoyable experience. While we successfully implemented all the baseline features, however, we regrettably had to forego some of the additional ones due to time constraints. Originally, we were working out on a different project that involved placing large resolution images on SDRAM to create an 3D game. Unfortunately, certain elements were taking longer than expected and we had to turn in an new proposal. A significant takeaway from this project was the importance of properly planning ahead. We should have realized the obstacles in our first project ahead of time and tuned in our new proposal in sooner. Our second project also included certain parts that geve use issues such as state transition and VGA color. While these were fixed, the time we spent debugging took away from the time that we could have implemented additional features. Aside from this lesson, this project allowed us to showcase our knowledge and skills gained from ECE 385, culminating in a satisfying conclusion to our journey in this course.