ECE 385

SP 2019

Final Project Report

FPGA Game:
I Wanna Be the Zuofu

Yuhang Ren & Ximin Chen
ABO/28 Feb 3:00-5:50 p.m.
TA: Patrick Wang, Xinbo Wu

1 Overview

In this lab, we implemented a system-on-chip that runs the game: I Wanna Be the Zuofu, where we spend the most efforts to design. The SoC includes a hardware part which implements System Bus, RAM, and Video Display. Additionally, we will use NIOS II CPU to interface with the USB keyboard which has been accomplished in lab 8. With the SoC, we demonstrated the I Wanna Be the Zuofu game, where one character having the look of Zuofu whose sprite is adapted from the game I Wanna Be the Guy, goes through a series of difficulties to beat the final boss, DE2-115 and got GPA 4.0.

2 Game Introduction

The game, I Wanna Be the Guy, has been known to people from all over the globe for many years, and has received many times of adaptation to the games I Wanna Be the XXX. In our game, I Wanna Be the Zuofu, the character goes through rough roads to get to his final goal: GPA = 4.0. Before his dream comes true, he would be dead if he touches any of the killers: spikes, apples and FPGA (boss). If he is dead, the player would press “R” to reset to the latest map he has been in. If he beats the final boss, FPGA, he would open the door to GPA = 4.0 and win. In each map, the character moves to the map end to get to the next level. Moves are done by pressing keyboard “A” to go left, “D” to go right, and “K” to jump. He can shoot bullets by pressing “J”.

3 Written Description

3.1 Overall Structure

The overall structure of our system is illustrated below. For all time, color mapper keeps drawing images to VGA, as has been implemented in lab 8. With the signals DrawX, DrawY telling which pixel VGA is currently drawing, color mapper decides which color (red, green, and blue) to map into the corresponding pixel. The module Zuofu tells the status of the character. It’s connected to keyboard by NIOS II system, which outputs keyboard values to Zuofu. The modules of MAP which contains the features of spikes, blocks, apples in each level. Boss interacts with Zuofu and color mapper to decides game logics such as death and win. Given DrawX and DrawY, the current drawing position of the block, the modules would decide whether their corresponding objects are at that location, and send out a signal, e.g. is_block and is_apple. Color mapper decides which object to draw with some priority. The color mapper also contains a timer and life counter of the player, tracking game time and death times of the player, and draw them on the screen.

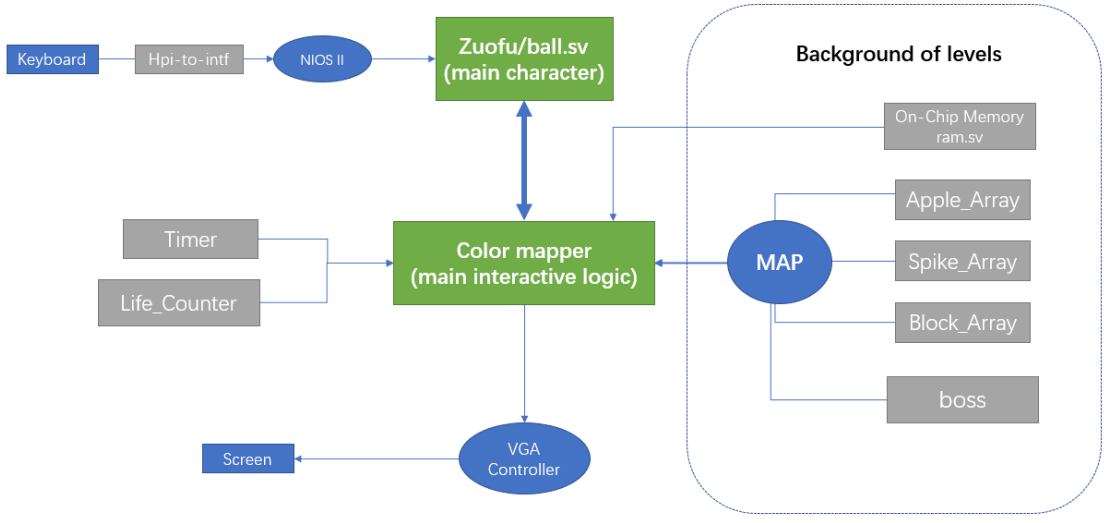

Figure 1: Block diagram of the overall structure of our design

3.2 Keyboard

In order to provide a smoother control of character, we modify the platform design comparing to it in lab8. It used to support reading only one keycode but two right now. In detail, we have added one more pointer and one more I/O peripheral unit and assign the value of the second keycode to the new pointer. After that, FPGA can read the second keycode's value.

3.3 Picture Drawing (OCM)

In our project, pictures are stored in On-chip Memory (OCM) in the form of palette indices. The use of OCM and palette is demonstrated clearly on ECE385 Helper Tools, so we don't talk about it much in our report. One difference we have from the online helper tools is that we did not use only one sprite. To save OCM as well as to simplify address calculation, we use separate sprites for different objects. With the helper tools, it would be easy for us to draw a single image, as well as multiples, or a region of an image. For example, we draw a lot of blocks in our game.

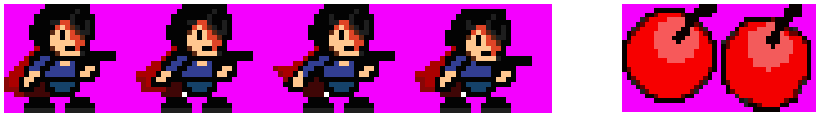

Figure 2: Sprite of idle Zuofu and sprite of apple. Notice that they are separate sprites rather than one sprite containing everything

It's easy to store a block into OCM, but it's impossible and would be a great waste of memory to store many identical blocks into memory just in order to draw multiple blocks. In this case, we just need to store one copy of block into memory, and calculate and refine the address in memory for a pixel. If block_array tells the color mapper that is_block is high in the range DrawX∈[0,399], DrawY∈[0, 19], the address for block ram would be refined into the size of a block using modular operations:

Block_X_Addr = (DrawX – Block_X_Pos) % Block_Size;

Block_Y_Addr = (DrawY – Block_Y_Pos) % Block_Size;

*Block_Address = Block_X_Addr + Block_Y_Addr * Block_Size;*

Sometimes we need to draw an image in different directions, e.g. spikes in all four directions. In this case, we still don't need to add new sprites. We just need to write a case statement selecting different cases for different directions of a spike. For different cases, we calculate the memory address differently to obtain the direction we want.

3.4 Animation

To make our game look more fun, it would be important to add animation to some elements, in our case, apple and Zuofu. We want Zuofu to breath up and down when he is standing still, and flap his feet when he's walking. First, we need some logic tell which status Zuofu is in, either walking or idle, and use different sprites for different status. To add animation to each status, we just need to alter the sprites used to draw Zuofu. Circularly switching among the set of images in the sprite results in animation. To simplify calculation, we make the images arrange closely in one row, each image having the same size. Then, an index selecting which image to draw in the current cycle is needed when calculating the drawing address. We call the index Walking_Counter, since it adds 1 repetitively, and clears to zero when maximum is reached. We still need something to trigger that addition, which determines how fast the animation runs. Hence we can create a clock triggering animation, which is fairly simple. The memory address for a walking Zuofu is calculated as follows:

*Walking_Address = Walking_X_Addr + Walking_Counter * Walking_X_Size + Walking_Sprite_Size * Walking_Y_Addr*

3.5 Map Design

The drawing space of our game is 640*480, and the size of a typical block or spike is 20 * 20 pixels. Hence it would be perfect to simplify the drawing of a map to a (640/20) * (480/20) = 32 * 24 bitmap. We first design the map using the 32 * 24 space, and put the blocks into 640*480 space. Because it's way more hard work to put the blocks one-by-one, we use one block entity or spike entity to refer to a range of blocks and spikes, for example, DrawX∈[0,399] and DrawY∈[0, 19], as has been discussed before. In this way, we can view the map as an array of large blocks and spikes, which simplifies map definition.

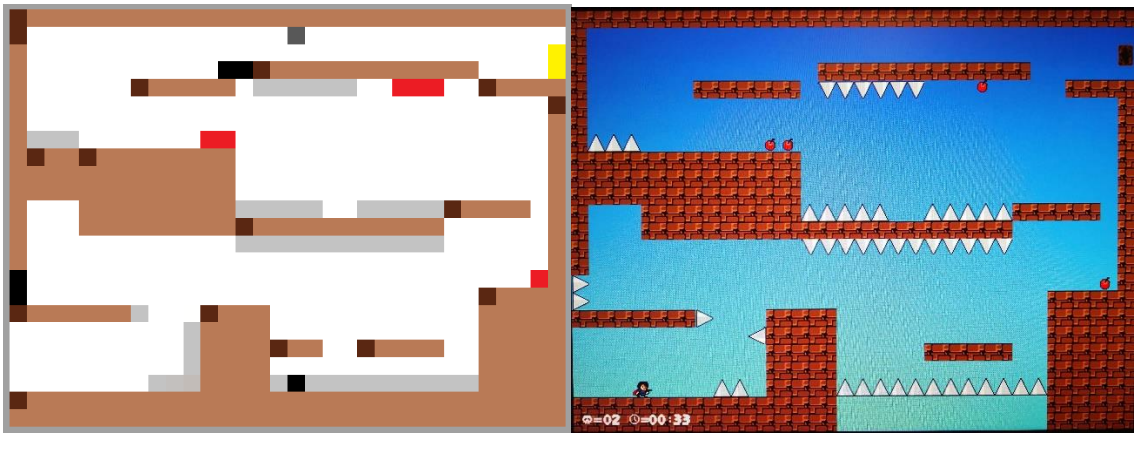

Figure 3 Bitmap used to design the map.

3.6 Jumping and Gravity

Basically, we use the DrawX and DrawY to detect if zuofu is in air. As it shows on the figure. If there is nothing exist in the pixel, we will simply use logical method due to the current location to make the gradually changing sky. The only parameter changes in RGB is Green. Therefore, going through the screen to video memory, when the DrawX, DrawY are locating the exact pixel below zuofu. We will check if this pixel is sky or any kind of trap, which is showing the code fragment.

*If (Green != 8'h51+(6*DrawY)/25 && is_MapEnd == 0 && is_apple == 0 && is_spike==0)*

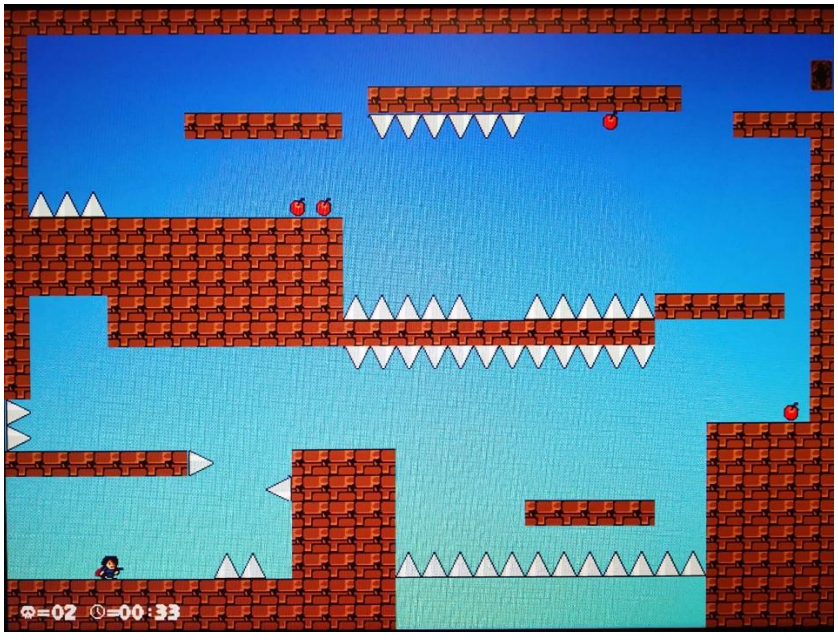

Figure 4: Map of the second level. Whether the character is in air is determined by color right under the character's foot.

If so, the variable in_air becomes 1 and will be output to module of zuofu to help implement the gravity.

In ball.sv, we use in_air and other parameter to check if we need to add a gravity acceleration in Y_motion. Simply explaining, jump_count equals to zero when zuofu start jumping and air_reset become one when we find zuofu is in the air.

else if (jump_count != 0 && in_air) begin

Ball_Y_Motion_in = Ball_Y_Motion + gravity;

There is a variable air_reset is used to avoid race competition between jump up and landing just above the block which will be explain in the following section.

Also, powerful zuofu can jump twice in the air. To implement this feature, we set a jump_counter which increase by one when it jumps and reset to zero when in_air is zero too. If the counter is larger than 2, it would disable the jump operation so that it can only jump twice.

3.7 Block Detection

To avoid uncomfortable overlapping between zuofu and blocks. We designed a block detection function in Color_mapper. For direction up, left and right, the logic is similar to in_air. For direction down, beyond detection. We also need to set zuofu on the block rather than in the block.

In system Verilog, we calculate the position and motion by bit logic which is similar to integer. Since there is gravity in falling, the Y_motion can be rather too large. That is, for example if zuofu is reaching the block

with distance 4 in motion of 10, zuofu would be inside block 6 unit deep and then stop in next frame. To avoid the bug, we calculate the position of zuofu in next frame. Compare it with each blocks in block_array. If we find zuofu will be in one of the blocks next frame, we would directly set zuofu's coming position on the upper surface of that block.

3.8 Shooting

To implement shooting a bullet, we draw a white point on the screen as a bullet. There is a state machine to control each bullet.

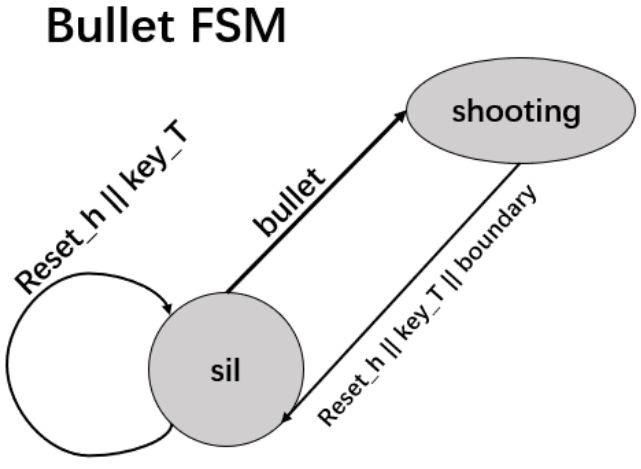

Figure 5: Bullelt FSM

In the state of sil, the bullet would follow the zuofu and hide behind him. It would turn to shooting state due to the press of shooting button. In the state of shooting, the bullet would be assigned a constant X_motion due to the current direction of zuofu. After the bullet reach the boundary of the screen, it returns to sil state and can be shoot again. Zuofu's bullets can due damage to apple and the boss. In order to determining the hitting, we compare the is_bullet signal with the position and size of the object. If the current pixel for the object is not transparent, we will judge it as hitting. To achieve shooting multiple bullets consecutively. We designed a general bullet module. It will use case statement and a counter to divide shooting signals and combine bullet signals

3.9 Status Bar (Printing text)

Our game has a status bar which tracks the number of deaths and the time that has passed.

To track the number of deaths, we just need to have a synchronized variable triggered when the character is alive before a rising edge and dead after the rising edge. To track the time, we need a real time clock which has a frequency of 1 Hz. The real time clock triggers the addition of another variable, second_counter, from which we calculate the minute and second digits. The drawing technique of the status bar does not change much from before. We just need to use the digits as an offset to calculate the address in the sprite.

Figure 6: Status bar in the game

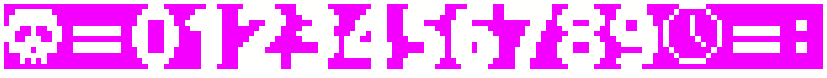

Figure 7: Sprite of status bar

3.10 Game State

There are several special states we need to handle with in the game. They are death, map changing and winning.

In the game, apples, spikes and boss itself can kill zuofu by touch, so we set a killer type for those three objects. The logic is like judgement of bullet hitting. Check if block of zuofu is overlapping with any kind of killer in opacity pixel. If so, death will trigger the gameover logo show in the screen and lock the movement of zuofu unless player press “R” to restart in current map.

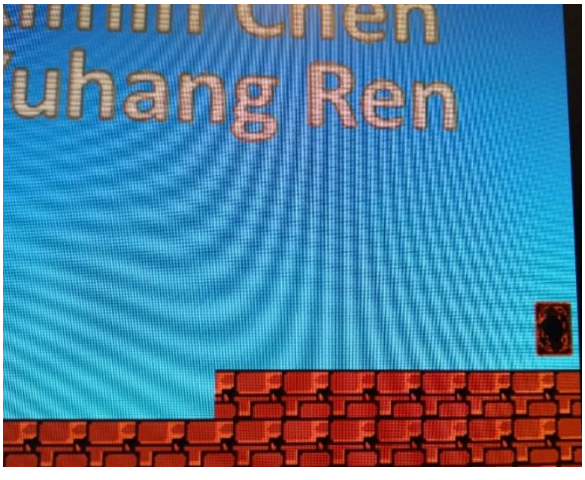

Figure 8: Door as map end, where the player goes to the next level.

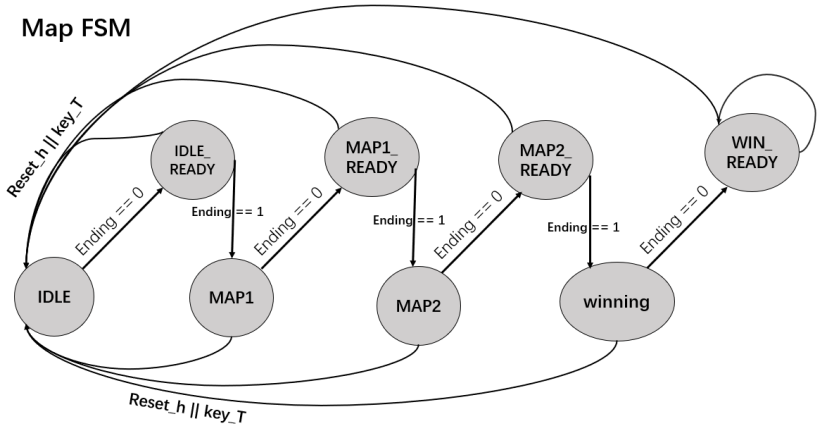

Figure 9: FSM of Map logic, which determines which map the user is in.

There are gates in each map and when zuofu reaches the gate, it would generate a Ending signal which trigger the map state machine.

In different map state, the value of state index would choose corresponding arrays of block, spike etc. and update the video memory in next frame. The winning condition is actually the last map without a gate out. The player can only move and press “T” to reboot the whole game.

3.11 Boss AI

The boss in the last map would move up and down in the right most of the screen and there is also a HP bar on the top of the screen. The boss will automatically shoot apples directly to zuofu. To calculate the motions of shooting apple, we use X_motion_step as a standard then get the ratio and Y_motion. Like we mention before. Logic calculation is different from floating calculation. To ensure the result is valid, use case statement to do divide and multiplication in positive number.

```
if (Ball_Y_Pos>=boss_bullet_Y_Pos)
    boss_bullet_Y_Motion_in = (Ball_Y_Pos - boss_bullet_Y_Pos)/((boss_bullet_X_Pos -
        Ball_X_Pos)/boss_bullet_X_Step);
else
    boss_bullet_Y_Motion_in = ~( (boss_bullet_Y_Pos - Ball_Y_Pos)/((boss_bullet_X_Pos -
Ball_X_Pos)/boss_bullet_X_Step)) + 1'b1;
```

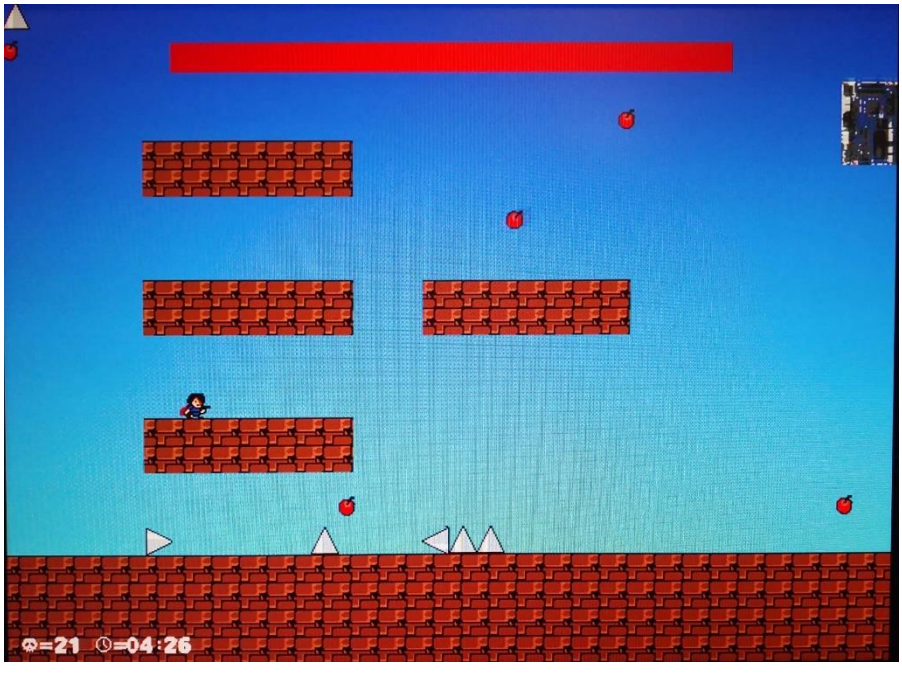

Figure 10: Picture of the boss AI shooting towards the character

4 Design Statistics

LUT	12637
DSP	-
Memory (BRAM)	721107 bits
Flip-Flop	2254
Frequency	120.71 MHz
Static Power	105.68 mW
Dynamic Power	0.86 mW
Total Power	180.77 mW

5 Design Module Information

Module: lab8.sv
Inputs: [3:0] KEY, CLOCK_50, OTG_INT
Outputs: [7:0] VGA_R, VGA_G, 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, TG_RST_N, OTG_INT, DRAM_CAS_N, DRAM_CKE, DRAM_CS_N, DRAM_RAS_N, DRAM_WE_N, DRAM_CLK, [7:0] LEDG, [12:0] DRAM_ADDR, [1:0] DRAM_BA, [3:0] DRAM_DQM
Inout: [15:0] OTG_DATA, [31:0] DRAM_DQ
Description: This is a ball game implemented by NIOS II SoC.
Purpose: This is the top-level module of lab7. It also deals with the connection between FPGA and the board or peripheral.

Module: ball.sv
Inputs: Clk, Reset, frame_clk, rtc_clk, jump, death, in_air, blocked, top, update_flag,
[7:0] keycode, keycode_2,
[9:0] DrawX, DrawY, StayX, StayY,Ball_Y_Pos_Block, MapEnd_X_Pos,MapEnd_Y_Pos,
[9:0] Ball_X_Init, Ball_Y_Init ,
Outputs: is_ball, key_R, bullet,direct,Ending,key_T,actual_shoot,actual_jump,
[9:0] Ball_X_Pos, Ball_Y_Pos,
[9:0] Ball_Y_Motion,Ball_Y_Motion_in,Ball_X_Motion,
[3:0] counter, jump_count,jump_count_time, jump_reset
Description: This is a module for controlling the character in screen.
Purpose: control the character's position and motion due to the physical logic.

Module: hpi_io_intf.sv
Input wires: 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, [15:0] OTG_DATA,
Output wires: [15:0] from_sw_data_in, [1:0] OTG_ADDR, OTG_RD_N, OTG_WR_N, OTG_CS_N, OTG_RST_N
Inout wire: [15:0] OTG_DATA,
Description: This is the file which interfaces between NIOS II and EZ-OTG chip
Purpose: This is the module used as interface between NIOS II and EZ-OTG chip

Module: boss.sv
Inputs: Clk, Reset, frame_clk, rtc_clk, key_R, is_bullet, [3:0] state_index, [9:0] DrawX, [9:0] DrawY, [9:0] Ball_X_Pos, [9:0] Ball_Y_Pos,
Outputs: [9:0] boss_bullet_X_Addr, [9:0] boss_bullet_Y_Addr, [9:0] boss_X_Addr, [9:0] boss_Y_Addr, is_boss,is_boss_bullet,is_bar, [8:0] boss_counter;
Description: This is a module for controlling the boss in the last map.
Purpose: Decide the behaviors and status of the boss.

Module: MAP.sv
Inputs: Reset_h, key_R, Clk, Ending, key_T,
[9:0] DrawX, [9:0] DrawY, [8:0] boss_counter
Outputs: is_MapEnd, update_flag, win,[3:0] state_index,
[9:0] Ball_X_Init, Ball_Y_Init, bullet_X_Init, bullet_Y_Init, MapEnd_X_Pos,
[9:0] MapEnd_Y_Pos, MapEnd_X_Addr, MapEnd_Y_Addr,
[9:0] Block_X_Pos [0:16], Block_Y_Pos [0:16],Block_X_size [0:16], Block_Y_size [0:16]
Description: This is a module that stores the map information.
Purpose: assign constant values in the maps.

Module: rtc.sv
Input wires: Clk,Reset_h,key_R
Output wires: sec_clk
Description: This is the file which generates real-time-like clocks
Purpose: To track the time of game and provide clocks for animation

Module: ram.sv
Input wires: Clk, [8:0] read_address
Output wires: sec_clk
Description: This is the file which stores ram interface for all sprites
Purpose: To use on chip memory, we need ram.sv to utilize txt files as memory

Module: Spike_Array.sv, Block_Array.sv, Apple_Array.sv
Input wires: Reset_h, key_R, VGA_VS, death, [3:0] state_index
[9:0] Ball_X_Pos, Ball_Y_Pos, DrawX, DrawY
Output wires: is_spike, [1:0] Spike_Direct, [9:0] Spike_X_Addr, Spike_Y_Addr
Description: This is the file which stores positions of spike arrays and apple arrays. Each module stores one type of object of one map.
Purpose: To store positions of objects easily

6 Credits

Drawing Tutorial: https://github.com/Atrifex/ECE385-HelperTools

VGA Use: https://wiki.illinois.edu/wiki/display/ece385sp19/Lab+8

7 Conclusions

In this lab, we designed the game, I Wanna Be the Zuofu, and implemented related hardware and software. We tried implementing audio, but it did not work. We also tried using SRAM to store large images, but found that the .ram file generated from ECE385 helper tools seems incorrect. In general, our final project works well, and the game itself is interesting and worth playing.