ECE 385

Fall 2022

Experiment 6

Final Project: Rally X Video Game

Aleksai Herrera

Julio Cornejo

Zijing Zhao 11:00 a.m.

Introduction

The goal of our final project was to create a playable video game based on the arcade game Rally X, a game involving a player moving around the screen to collect flags while being chased by the enemies they would have to avoid. Along with other features such as a mini-map, score display, and lives display, the main difficulty would be coming from the scrolling of the map and writing the enemy AI to handle different types of collisions. Our implementation uses the USB keyboard drivers from lab 6 to and keys A, S, D, and W to control the player.

Overview

The project was started off using the lab 6.2 ball files as the basis, the module description will describe each specific module. Fixed sprite drawing was done using the color mapping module and predesigned sprites in ROMs, like font ROMs in lab 7. The game involves collecting flags and avoiding enemies while traversing the maze. There are two levels/maps, and the player proceeds onto another after collecting 4 flags. After collecting 4 flags in the second map, a win screen displays. Collisions with enemies result in losing a life. When you lose a life, the player's coordinates are reset. Once all 3 lives are lost a game over screen is displayed. The logic to do this is done between several modules sending signals, but mainly something like a state machine, a sequential always_ff block, in the ball.sv module.

Module Descriptions

Module: VGA_Controller.sv

Inputs: Clk, Reset

Outputs: hs, vs, pixel_clk, blank, sync, [9:0] DrawX, [9:0] DrawY

Description: This module holds the coordinates where each pixel is being drawn, and it outputs the pixel_clk, hs, vs, and blank which will be used by other modules in drawing each color.

Purpose: This module keeps track of DrawX and DrawY which keep track of the electron beam that draws each pixel on the screen. It draws each pixel in raster order and outputs several signals which other modules will use to determine what color and what coordinates will be drawn.

Module: lab62.sv

Inputs: MAX10_CLK1_50, [1:0] KEY, [9:0] SW,

Outputs: [9:0] LEDR, [7:0] HEX0, [7:0] HEX1, [7:0] HEX2, [7:0] HEX3, [7:0] HEX4, [7:0] HEX5, [12:0] DRAM_ADDR, [1:0] DRAM_BA, [15:0] DRAM_DQ, DRAM_LDQM, DRAM_UDQM, DRAM_CS_N, DRAM_WE_N, DRAM_CAS_N, DRAM_RAS_N, VGA_HS, VGA_VS, [3:0] VGA_R, [3:0] VGA_G, [3:0] VGA_B

Inout: [15:0] DRAM_DQ, [15:0] ARDUINO_IO, ARDUINO_RESET_N

Description: This is the top-level module, logic, and connections are made to connect the various modules and it is also where we connect the different blocks in the platform designer.

Purpose: This module serves as the top level where the rest of the sub modules are initialized and connected to the components set up in the platform designer.

Module: HexDriver.sv

Inputs: [3:0] In0

Outputs: [6:0] Out0

Description: This module converts our decimal input to hexadecimal output.

Purpose: This is used to display our answer and inputs to the Hexadecimal display on the FPGA.

Module: Color_Mapper.sv

Inputs: input [10:0] BallX, BallY, DrawX, DrawY, Ball_size,

input [10:0] enemyX,enemyY, enemy1X,enemy1Y,

input [2:0] livesCount, //input thats keeps track of lives

input [3:0] playerVelocity, enemyVelocity, enemy1Velocity,

input Clk,Reset, Winscreen,GameOver,

input [4:0] flagDisplay, //Determines which flags are displayed

input [3:0] flagcount, //Tells score what flagcount is

input [1:0] levelindex, //determines which map is drawn

Outputs: output [4:0] flagBurst, //detects collision with flag

output crash_on,

output[4:0] map_on,mapE_on,map1E_on,

output logic [7:0] Red, Green, Blue

Description: This module is extended in our project, it not only uses logic to determine what to draw on the screen depending on the DrawX and DrawY coordinates, but also contains instantiations for modules dealing with map drawing, scrolling, sprite drawing, and collision

Purpose: This module is really the basis for where drawing sprites, handling collisions, and scrolling the map occurs. It contains several modules inside it that will be further elaborated on how the drawing and map scrolling is done. As for the collision, we take in the player and enemy coordinates, and if any corners on the 16x16 square overlap we detect a crash and send the crash_on signal to ball.sv where the player will display a crash sprite and reset. Detection with the walls of the map was done by checking the screen_rom, which held the data for the current area of the map being drawn, and then checking if it was a 1. If a player or enemy collided with a wall we would have to add and offset so they were no longer colliding with the wall and change the players direction to create the sense of automatic turning on the walls. Overall, the drawing was done using ROM's similar to font_rom from lab 7, but for each 1 or 0 we read we would draw a 16x16 block of pixels on the screen. Once our DrawX reached past 512, we would start to change the logic in the RGB_Display always_comb to draw the section that displays the title, score, mini map, and lives. Because we had to implement scrolling the player would be drawn on the center of the screen and everything around had to have an offset drawn to give the illusion of scrolling, for example the enemies coordinates would be shifted back if the player was driving away from them.

Module: ball.sv

Inputs: input Reset, frame_clk,crash_on,

input [4:0]map_on,

input [7:0] keycode,

input [3:0] flagcount,

Outputs: output [2:0] livesCount, //Counts number of lives, at 0 the game over screen will appear

output Winscreen,flagreset, GameOver,

output [1:0] levelindex,

output [3:0] playerVelocity,

output [10:0] BallX, BallY, BallS

Description: This module contains sequential logic controlling the player's movement, response to collisions, and controls the overall state of the game by using its sequential logic in a way similar to a state machine. The description of this logic will go as follows, if on the press of the Reset button the players coordinates are set to preset initial coordinates and flag drawing and count is set to 0 meaning we display all flags. Additionally, we are also set to level 1 (how this is done will be described in screen_rom module. If the player crashes with an enemy car, livesCount is reduced, and the coordinates are reset. Starting from 3, if livesCount reaches 0 we display the game over screen. If the 4 flags are collected we advance the player to level 2 and reset the flags and players coordinates. The same conditions apply for level 2 but if we collect 4 flags on level 2 the Winscreen signal is sent to Color_Mapper to display the win screen, which is a trophy sprite.

Purpose: The purpose of this module is to control the player and give the player collision properties, making the player movement feel natural and auto turn when colliding with walls, crash when colliding with enemies. Also, it contains the logic to send to tell the Color_Mapper to draw the game over screen or win screen depending on if we collected flags or have crashed.

Module: enemy.sv

Input: parameter StartX=320, StartY=440,Velocity=1

input Reset, frame_clk,

input [7:0] keycode,

input [10:0] PlayerX, PlayerY, //These are the players cordinates, will be used to reference when

input [4:0] mapE_on,

Output: output [3:0] enemyVelocity,

output [10:0] BallX, BallY

Description: This module controls something like the ball.sv, except keycodes are not used to determine the direction but rather calculations using the player's coordinates. First, we take the distance between the enemy and the player in the X coordinate and based on that we try to minimize the distance. Once this reaches 0, the same is done on the Y axis, leading to a collision with the player. However, this was not sufficient to deal with additional collisions with the map walls. The collision with walls was like that of ball.sv, except we did not add an offset to detach them from the wall but did change direction. This meant they would not keep colliding with a wall because our old collision method would remove them from the wall but the logic that determines direction based on player coordinates would redirect them in the same direction causing them to keep crashing into the same wall. The parameters allow multiple instantiations of enemies with different speed and spawn coordinates.

Purpose: This module contains the logic for the AI of the enemies, a crucial part of the game as they are what causes the player to crash. The enemies chase the player to crash the player while traversing through the maze. The parameters allowed the enemies to be customizable regarding speed of movement and spawn coordinates.

Module: Spritefont.sv

Input: input [10:0] addr, addr1, addr2,

Output: output [15:0] data, data1, data2

Description: This module contains 80 16-bit registers with hard coded values of the car sprites, like how lab 7 implements font_rom. Depending on DrawX and DrawY certain bits of registers are read to draw the sprite on the screen.

Purpose: This method was used for sprite drawing in the early to middle stages of our project, the issue is that it is only able to draw single-colored sprites. As we moved on to the multicolored implementation this module was still useful to draw the single colors lives in the bottom right corner displaying how many more attempts the player had.

Module: entiremaprom.sv

Input: input [10:0] addr,

Output: output [79:0] data

Description: This module contains 130 80-bit registers. It is a read only ROM that contains two maps and depending on the levelindex (which is determined by what level we are on in ball.sv), we read 30x40 rectangles from this module and right it to the screenrom.sv. The 1's represents walls and are also used for collisions.

Purpose: This module is used to hold entire maps, so we can scroll to different parts of the map instead of displaying the entire map. It is separate from screenrom.sv so we do not display the entire map and instead only parts of the map. See ScreenScroller.sv for more in depth on how screen scrolling was done.

Module: screenrom.sv (an updated sequential version of maprom.sv, that module is not used in the final version and will not be fully described as it is almost identical to screenrnom.sv)

Input: input Clk,Reset,

input [10:0] addr, addr1, addr2, addr3,addr4,addr5,addr6, screenAddr,

input [39:0] screenData,

Output: output [39:0] data, data1, data2,data3,data4,data5,data6

Description: The module contains 30 40-bit registers which are written to depending on input data that is determined by ScreenScroller.sv. This is done in a sequential block and continuously to update the map depending on where the player's coordinates are.

Purpose: The purpose of this module is to hold the data for the section of the map that will be displayed on to the screen. Based of this portion of the map, the map's collision and drawing is done. We write to the registers on this module depending on what part of the map we are on (the process will be further explained in ScreenScroller.sv.

Module: ScreenScroller.sv

Input: input [9:0] ScreenX,ScreenY,

input Clk,

input [1:0] levelindex, //This is 0 for level 1 and 1 for level 2

Output: output [10:0] screenAddr,

output [39:0] screenData

Description: This module contains a sequential block of logic that scrolls through, using a variable that is incremented 0 through 30 and reset at 31 back to 0, and using the (ScreenX,ScreenY) player coordinates, reads a section of data from the entiremap_rom. It then reads bits from the data array starting from, ScreenX to 40 bits long using the notation screenData<=entiremap_data[ScreenX+20-:40]. This data is then outputted and written to screen_rom accordingly. When implemented correctly after adding some offsets, we can accurately scroll through the map by reading different parts of entiremap_rom and writing them to screen_rom that is displayed and drawing the player in the center, having the map move around. Also, an input of levelindex is used to add an offset when addressing entiremap_rom to retrieve data from the second map.

Purpose: The purpose of this module to implement scrolling of the map, a very important feature in replicating the original Rally X arcade game. Also, our entiremap_rom was instantiated in this module, which hold the maps of each level.

Module: flagManager.sv

Input: input Clk, Reset, flagreset,

input [4:0]flagBurst, //input from color mapper that uses collision to tell which flags to display

Output: output [4:0]flagDisplay,

output [3:0]flagcount

Description: This module contains a sequential block that depending on which bits of flagBurst (the collision array), or flagDisplay(basically tells us which flags are still being displayed and thus valid to collide with), will increment the flag count and turn off that flagDisplay bit so the flag disappears after being collided with. Its output flagcount is also used in displaying the score and game logic in ball.sv to advance the player to the next level or win screen. Also on flagreset, we set flagcount to 0 and display all flags (flagDisplay to all 1's). This is used on starting the game and resetting the flags when we proceed to level 2.

Purpose: This provides an easy way to determine how many flags the player has, which flags have been collected/collided with, and when to draw which flags. Basically, if we collect a flag, we increment flag count and turn off its display bit and collision so the flag disappears such as any other collectable item would. We also redisplay and set flag count to 0 on reset which is useful for when we proceed onto other levels.

Module: flagrom.sv

Input: input [10:0] addr, addr1, addr2, addr3,

Output: output [15:0] data, data1, data2, data3

Description: Depending on the flag's location on the screen and an offset used for scrolling the screen, we read from this 16 16-bit registers and draw the flag accordingly on screen.

Purpose: This module is similar to Spritefont and is used to draw single colored sprite for flags.

Module: textrom.sv

Input: [14:0] read_address

(**trophy.txt, gameover.txt, numandscoresprite.txt, textfile.txt, racecar.txt**)

Output [3:0] data_out

Description: This is a series of modules that end up synthesizing into ROMs. They have no write ability from input and instead make use of $readmemh("textfile.txt", mem) to write 4 bits of data to n locations based on the area of the png image we used in our python script. Most of the ROMs hold up to five sprites, and we make note of which line the .txt files ends so we can index into our ROM and draw the correct sprite based on game logic.

Purpose: This module was our breakthrough into the sprite drawing methodology. Previously we were using register-based drawing which was great for only singular colors. Now, each pixel holds data that is indexed into a color we can make note of and assign it to color mapper. Specifically, these modules draw our score, logo, trophy, and end screen.

Module: framerom.sv

Input: [10:0] read_address, read_address1, read_address2

Output: [2:0] data_out, data_out1, data_out2

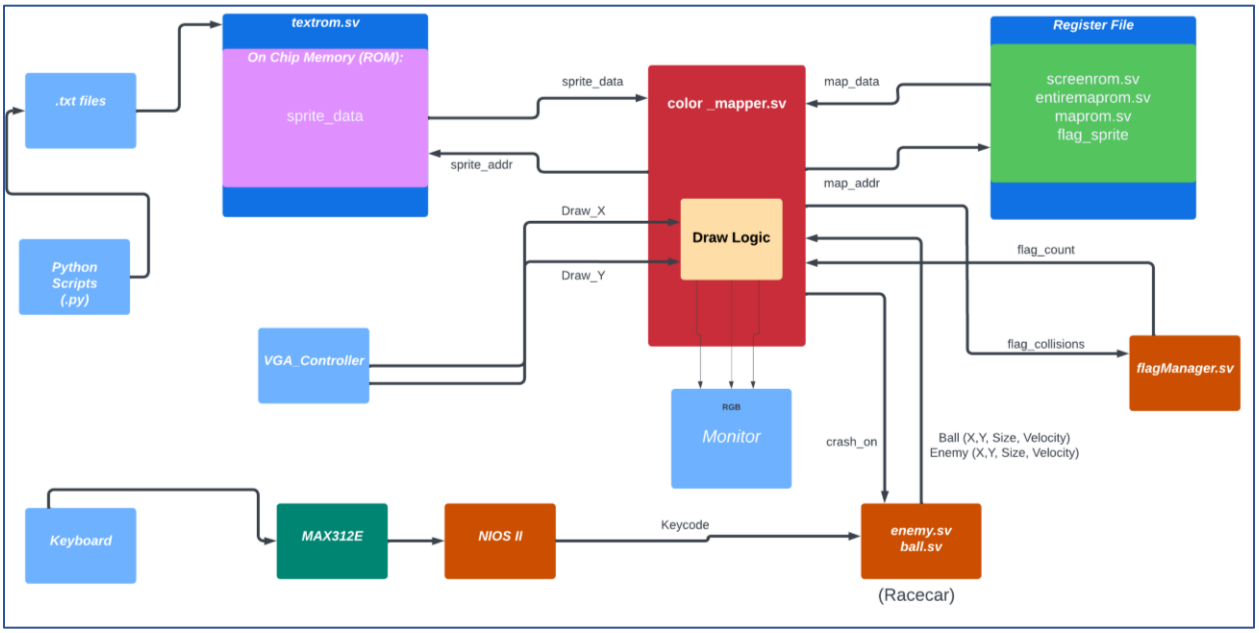
Description: This module is one synthesizable three output ROM that was necessary for drawing the car sprites for the main car (ball.sv) and each instance of the enemy cars. This ROM holds five sprites (16x16), a sprite for each direction the car will be traveling in and a bang sprite for when a collision occurs.

Purpose: This module is the central piece to drawing our game. It allows us to draw our main protagonist and the enemies with all their colors. It also allows us to visually see our collisions when the bang sprite occurs.

Module: trophy.txt, gameover.txt, numandscoresprite.txt, textfile.txt, racecar.txt

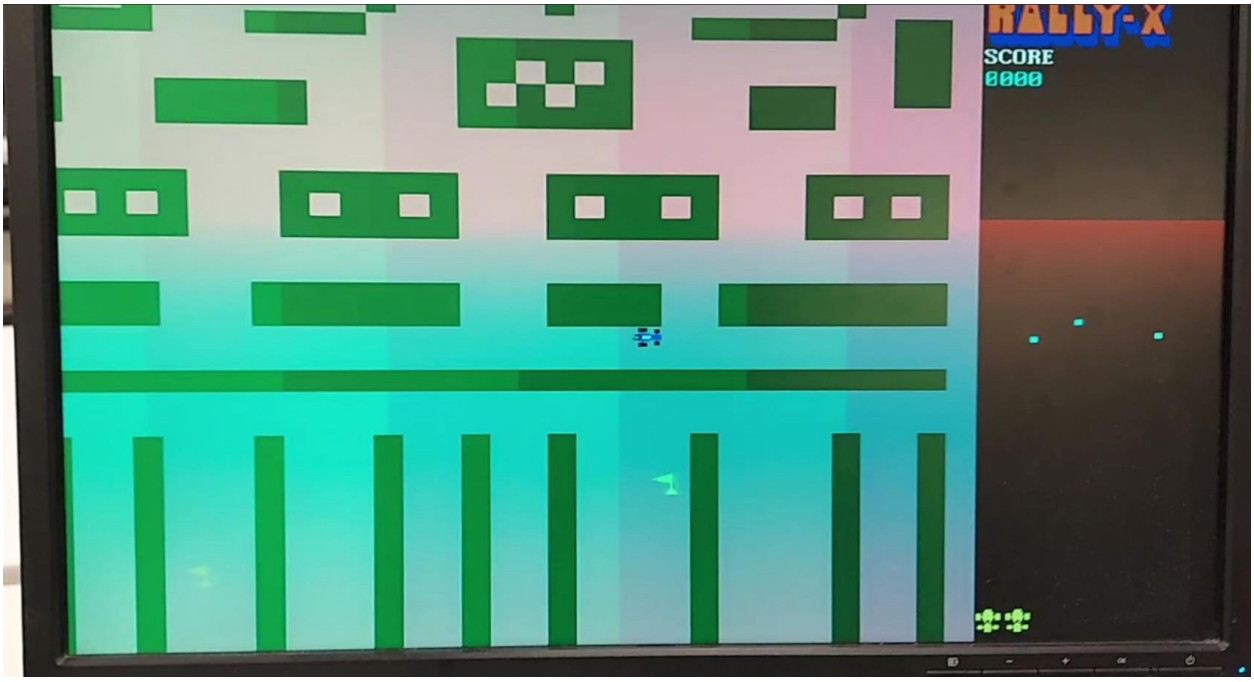
Description: These are not modules but are rather outputs from Helper Tools when we ran the python scripts on our PNG image of choice. These were crucial to drawing multi-colored sprites.

Block Diagram



This is a very simplified version of our working final project. We noticed and made a lot of simplifications from our proposal as we are devoid of a FSM as the backbone of our project. Instead, a lot of our collision and next state logic is handled in each of our System Verilog files corresponding to the cars themselves. We then pass this information to our Color Mapper which draws the results. It works perfectly without the need of extra logic for an FSM.

Photo of Game



Design Resource Statistics

Final Project

LUT	**12,154 (Logic Units)**
DSP	0
Memory (BRAM)	110,888 (Bits)
Flip-Flop	5,184 (Registers)
Frequency	128.01 (MHz)
Static Power	96.18 (mW)
Dynamic Power	0.71 (mW)
Total Power	106.20 (mW)

Our implementation made more use of on-chip memory than previous labs. We created a file “textrom.sv” that consists of four ROM modules. Each module initializes memory holding information for certain sprites (some are indexable). We made use of the python scripts from the helper tools, so

each ROM approximately holds each individual pixel of the sprite. The data itself varies in length depending on how many colors are needed to be represented per sprite. So, no surprise do we have an increased amount of 110,888 bits than our traditional lab sessions. It is worth noting that we were able to accomplish so much graphically (including a win and game over screen) without ever drawing a full screen's worth of graphics. Our compile time, resultantly, was very short at approximately two and half minutes; we were very cautious of our on-chip memory usage.

Current Issues/Bugs:

Although are game may seem fully functional to the player, there are still certain features that are not 100% functional or optimal. For example, on certain collisions with inner corners, the player can force themselves through the wall if they spam the key into the wall. Enemies also sometimes can go through walls if they go through the corner. Additionally, although not necessarily a bug, we would wish to create time in between crashing and resetting so our crash sprite would be more visible. Also, collision is done on the screen instead of the entire map, meaning that once the enemies are offscreen they no longer deal with map collisions as the map is not rendered off screen. This makes it more difficult for the player as the enemies can go straight towards the player without any obstacles. To address this we would need to retrieve collision data from entiremap_rom instead of screen_rom.

Conclusion

Overall, we were very happy with the results of our final project. We were able to implement all the baseline features and many of the additional. For instance, we had not initially planned on being able to successfully implement scrolling but were able to do it after a lot of time working on that feature. Some additional features we had implemented were multiple levels/maps, customize able enemies using parameters, mini-map display, lives display, and a multicolored title sprite. In the future our design could be used to easily add more levels, more enemies, and more flags. The maps are very customizable as viewed in the entiremap_rom, so anyone with basic to low level SystemVerilog experience could go in and change the map accordingly. Perhaps features we would add if were to continue the project would be having the flags randomly placed instead of preset, adding smoother turning animations, and title screen.