ECE 385

Spring 2017

Final Project

Flappy Bird

Joe Vande Vusse, Lisa Gentil

vandevu2, gentil2

Section ABI Tuesdays 8-10:50 AM

Xinying Wang

Table of Contents

I. Introduction

For our final project, we chose to make our own version of the video game Flappy Bird using a VGA monitor to display the graphics and a keyboard as input device. The rules will be explained in more details later, in section II.A. but the main idea is that, using the ball game we made for experiment 8 (SOC with USB and VGA Interface in SystemVerilog,) we are forcing a bird to bounce up and down using the keyboard in order to avoid some obstacles (represented as pipes) that move from right to left (as if the bird was actually moving from left to right). We used sprites to display the pipes as well as the bird and the background after having drawn them on Microsoft Paint. A Python script was utilized to convert the MS Paint PNG's to text files which were integrated with Quartus via ROM's/sprites. The programming languages used for our were project were SystemVerilog (majority of the modules), C, and Python.

II. Experiment

A. Flappy Bird: Game Protocol

- <u>Rules, Goal and Commands</u>

The rules of our version of Flappy Bird are that the bird must fly as long as possible without running into the obstacles in motion (the moving pipes). To avoid hitting them the user may hit the spacebar to fly higher. When the user does not hit the spacebar to make the bird

fly, it falls down because of gravity. Since the pipes ar on the top and the bottom of the screen, there is a gap where the bird should fly through. If it hits the side, the top or the bottom of an obstacle the bird dies and the game is over (at state “dead”.)

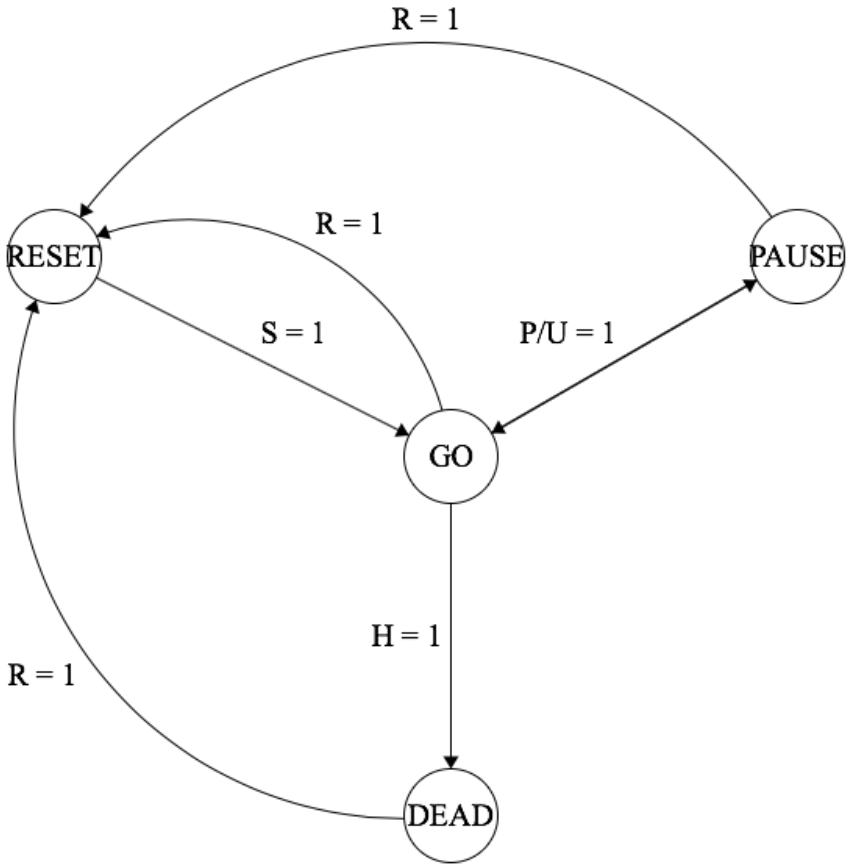
If the user lost, they may hit the reset button from the FPGA DE2 board and start again by hitting the spacebar after the screen has reset to the start screen (bird on the center, with no pipes around.) One last feature that we added is a pause/unpause button on the FPGA board for the user to be able to pause the current game and resume when ready.

B. FSM, Block Diagram & Waveform Simulation

- Final State Machine

There are three states in the game:

- RESET: the player has not started a game, or just hit the reset button on the FPGA after losing a previous game.
- GO: the player has hit the spacebar to start the game and the game is on.
- PAUSE: pause/unpause the game using an FPGA button.
- DEAD: the bird died (hit a pipe), the game is over and the player lost.



H = hit R = reset S = start P/U = Pause/Unpause

Figure 1: Final State Machine of Flappy Bird

- Generated Block Diagram

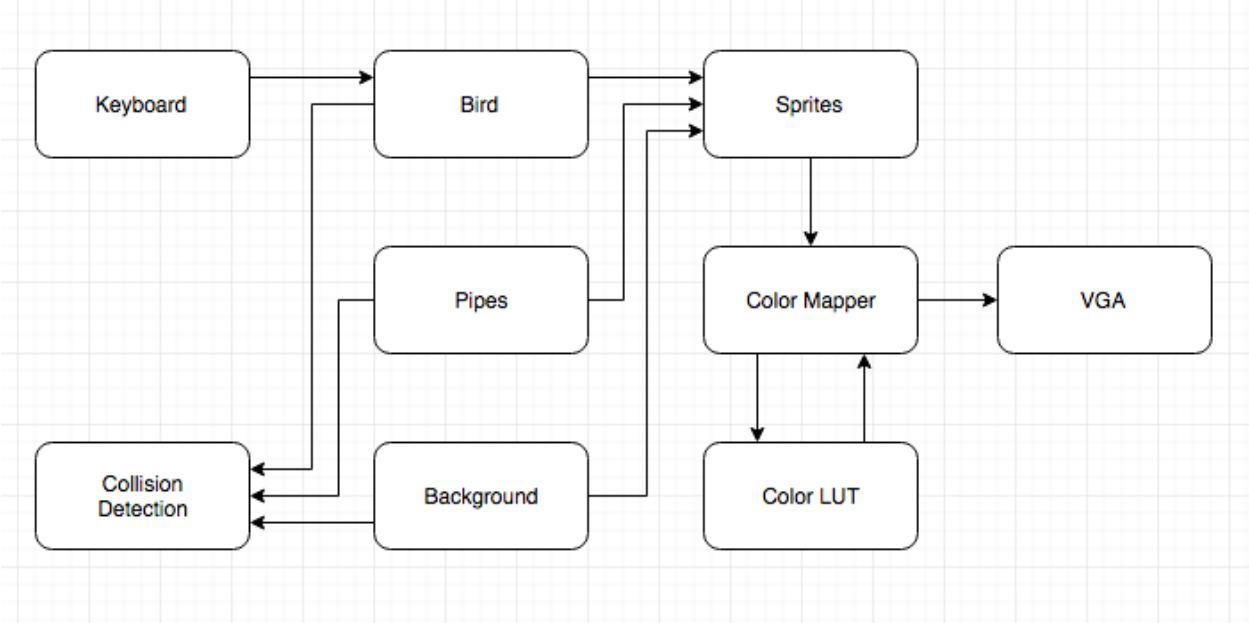


Figure 2: Readable Block Diagram of Flappy Bird

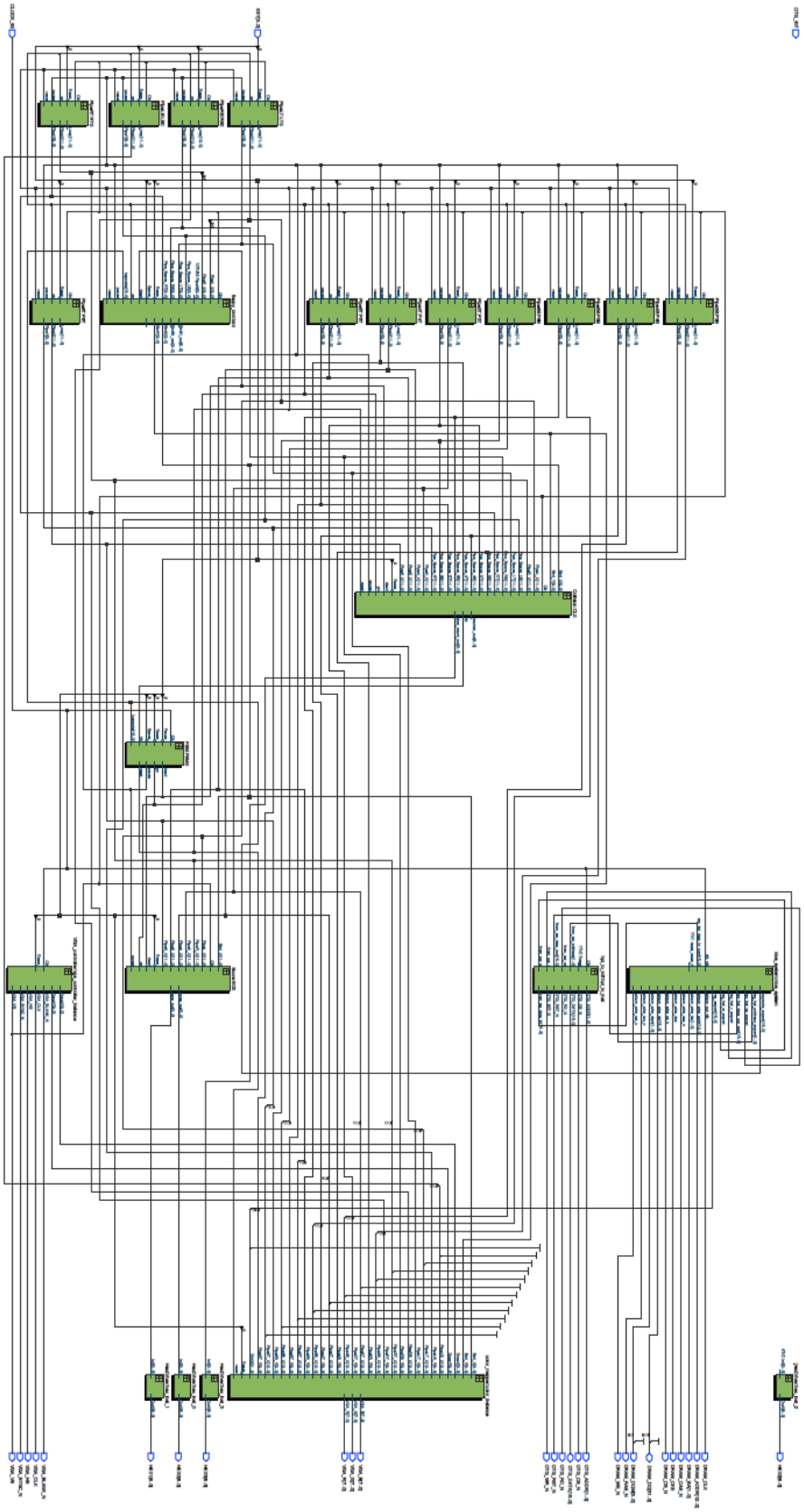


Figure 3: Block Diagram generated with Quartus

- Simulation Waveform

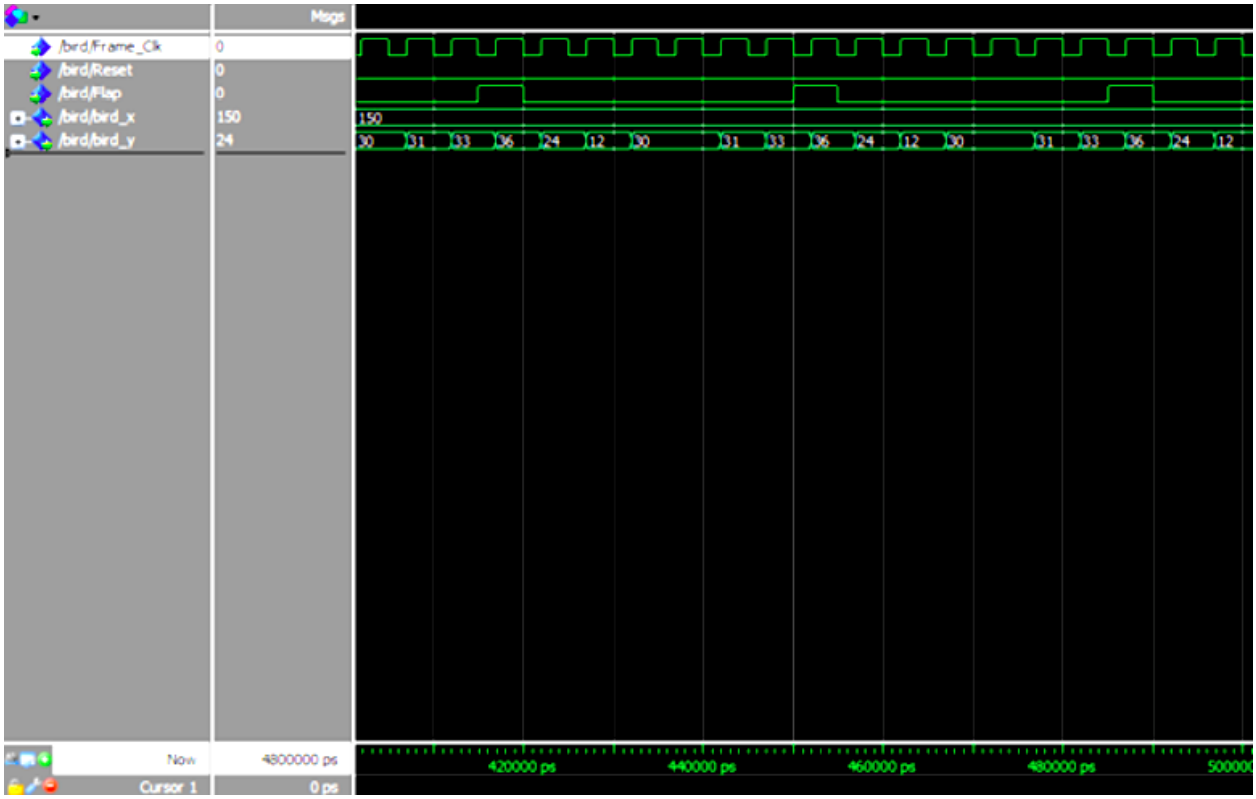


Figure 4: Simulation Waveform of the bird's position

C. USB Protocol

Note that the USB protocol for Flappy Bird is the same as the one provided for experiment 8.

- I/O_read: Given an 8-bit OTG address, the functions reads and outputs the data stored there. The data is a 16-bit unsigned value stored in an HPI register (see block diagram with HPI register's content.)

- I/O_write: The arguments passed in are an 8-bit OTG address that represents an HPI register and a 16-bit unsigned value. The function first stores the data in the otg_hpi_data and writes the address passed in into the otg_hpi_address field.
- USBRead: This function is a sequence of one IO_write and one IO_read. First we pass in the given address to IO_write to get to the register, and then IO_read is called to get the data stored at that address.
- USBWrite: This time, IO_write is called twice in the function. The first call is to write the 16-bit address at which we would like to write the data, and the second call is used to write the 16-bit data in the HPI_DATA field.

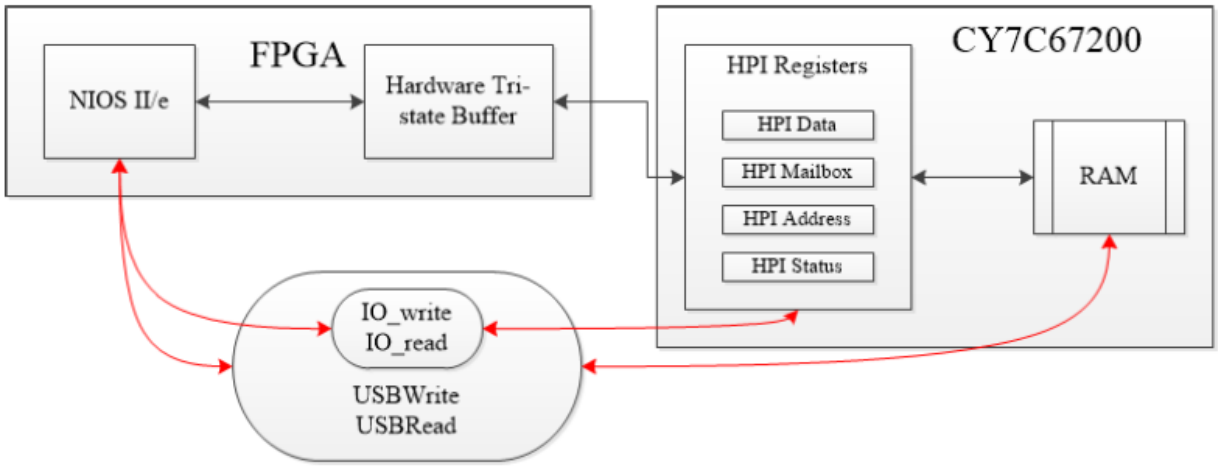


Figure 5: Block Diagram of USB Protocol

D. Hardware/Software Interface

The hardware/software interactions for this project are nearly identical to that of Lab 8. It utilized VGA, USB, and the NIOS II processor. It still contains the 5 HPI control parallel input/output registers responsible for sending /receiving data between the hardware and software. The blocks exist to determine whether the aforementioned registers will read or write and do so to their own SDRAM or the USB component. They facilitate NIOS software and USB controller chip interaction. The VGA display is made possible entirely by the FPGA hardware (and more specifically On-Chip Memory which we utilized exclusively). What it sacrificed in compile time it made up for in understandability and readability of our code to peers or interested parties in the future. These VGA components are instantiated in the top-level entity, and are not connected directly to the NIOS II. Rather, the NIOS II subsystem is also instantiated in the top-level from Qsys. The QSF pin mapper file maps the VGA components to FPGA board pins for video output via the VGA cable.

The project began with slightly more software interaction than Lab 8. Random pipe height generation via the C standard library rand() function was the initial goal. It was exported to the hardware via a 16-bit PIO output register called “RNG” in Qsys. Ultimately, complications arose with keeping the generated random height fixed. The pipe would appear and oscillate while on-screen. This coupled with plenty of other bugs forced a decision to utilize a large fixed number of pipes, mimicking randomness.

E. SystemVerilog Modules

Module: lab8.sv

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

Outputs: [6:0] HEX0, [6:0] HEX1, [6:0] HEX2, [6:0] HEX3, [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, [3:0] DRAM_RAS_N, [3:0] DRAM_CAS_N, [3:0] DRAM_CKE, [3:0] DRAM_WE_N, [3:0] DRAM_CS_N, [3:0] DRAM_CLK

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

Description: Top-level module

Purpose: The purpose of this module is to create the connections between FPGA's components and the other SystemVerilog System-On-Chip modules.

Module: flappy_bird.sv

Inputs: Reset, Clk, reset, go, pause,[15:0] keycode, [9:0] Obstacle_1_X, [9:0] Obstacle_1_Y_Top, [9:0] Obstacle_1_Y_Bottom, [9:0] Obstacle_2_X, [9:0] Obstacle_2_Y_Top, [9:0] Obstacle_2_Y_Bottom, [9:0] Obstacle_width

Outputs: [9:0] BirdX, [9:0] BirdY, [9:0] BirdW_out, [9:0] BirdH_out, hit

Inouts: ∅

Description: Outputs the X and Y coordinates of the bird

Purpose: The main purpose of this module is to keep track of where the bird is on screen. That way we can check using if statements whether or not the bird hit an obstacle (pipe) or not. It also updates the position of the bird as the player hits the spacebar or if they let the bird free fall.

Module: FSM.sv

Inputs: Reset, Clk, Space, Pause, hit, [15:0] keycode

Outputs: reset, go, pause, dead

Inouts: ∅

Description: Final State Machine of the game

Purpose: The purpose of the FSM module is to implement the final state machine of the flappy bird game. This process includes creating the different states of the game (go, pause, reset and dead, as described in section II.B.,) and the key combinations and conditions to go from a state to another (explained in figure 1.)

Module: HexDriver.sv

Inputs: [3:0] In0

Outputs: [6:0] Out0

Inouts: ∅

Description: Assigns hexadecimal outputs to each binary input

Purpose: The purpose of this module is to control what value should be displayed on the Hex display, for that each 4-bit binary number (from 0 to 15) is assigned to the hexadecimal

value it correspond to (from 0 to F.) In our game, the hex display is mostly used for debugging purposes.

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

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

Inouts: [15:0] OTG_DATA

Description: Sets outputs to positive edge of the main clock

Purpose: The purpose of hpi_io_intf is to reset variables to their initial value when the Reset button is pressed on the FPGA board. Else every output signal is set to its corresponding input signal's value.

Module: Color_Mapper.sv

Inputs: Reset, reset, [12:0] PipeLT_X,[12:0] PipeLB_X, [12:0] PipeRT_X, [12:0] PipeRB_X, [12:0] Pipe3T_X, [12:0] Pipe3B_X, [12:0] Pipe4T_X, [12:0] Pipe4B_X, [12:0] Pipe5T_X, [12:0] Pipe5B_X, [12:0] Pipe6T_X, [12:0] Pipe6B_X, [9:0] PipeLT_Y, [9:0] PipeLB_Y, [9:0] PipeRT_Y, [9:0] PipeRB_Y, [9:0] Pipe3T_Y, [9:0] Pipe3B_Y, [9:0] Pipe4T_Y, [9:0] Pipe4B_Y, [9:0] Pipe5T_Y, [9:0] Pipe5B_Y, [9:0] Pipe6T_Y, [9:0] Pipe6B_Y, [9:0] Bird_X, [9:0] Bird_Y, [9:0] DrawX, [9:0] DrawY, RNG

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

Inouts: ∅

Description: Takes care of the monitor display

Purpose: The purpose is to instantiate the sprites using the ROM files, and decides what to draw and where, that is what color should be assigned to each pixel considering the position of the bird, the pipes.

Module: VGA_controller.sv

Inputs: Clk, Reset

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

Inouts: ∅

Description: Generates new clock and outputs X/Y coordinates

Purpose: The purpose of this file is to keep track of the timing using a slower clock, VGA clock and to keep track of where we are are drawing on the screen.

Module: Collision.sv

Inputs: Clk, Reset, reset, go, pause, dead, [11:0] PipeL_X, [11:0] PipeR_X,[11:0] Pipe3_X, [11:0] Pipe4_X, [11:0] Pipe5_X, [11:0] Pipe6_X, [11:0] Pipe_Space_LT, [11:0] Pipe_Space_LB, [11:0] Pipe_Space_RT, [11:0] Pipe_Space_RB, [11:0] Pipe_Space_3T, [11:0] Pipe_Space_3B, [11:0] Pipe_Space_4T, [11:0] Pipe_Space_4B, [11:0] Pipe_Space_5T, [11:0] Pipe_Space_5B, [11:0] Pipe_Space_6T, [11:0] Pipe_Space_6B, [9:0] Bird_X, [9:0] Bird_Y

Outputs: hit, [8:0] counter_out, [8:0] pipe_count_out

Inouts: ∅

Description: Checks if collision occurs.

Purpose: The purpose of the of this module is to calculate if the bird collides with one of the pipes based on the Y_position and the X_position of the bird compared to each pipe.

Module: score.sv

Inputs: Clk, Reset, reset, go, pause, dead, [11:0] PipeL_X, [11:0] PipeR_X, [11:0] Pipe3_X, [11:0] Pipe4_X, [11:0] Pipe5_X, Pipe6_X, [11:0] Bird_X

Outputs: [3:0] tens_out, [3:0] ones_out

Inouts: ∅

Description: Get score of current game.

Purpose: The purpose of score.sv is to calculate the score: each pipe the bird went through, at least halfway, increments the score by one. The score is then displayed on the HEX display. For that we use a mod-like calculation for the tens' value and the ones' value.

Module: color_LUT.sv

Inputs: [2:0] inColor

Outputs: [7:0] R_out, [7:0] G_out, [7:0] B_out

Inouts: ∅

Description: Creates look-up table

Purpose: The purpose of the color LUT is to assign the RGB values of specific colors to a specific case (works as a switch case, with unique case) represented by numbers. Those numbers will be used the ROM files to define the color of our objects.

Module: PipeLB.sv

Inputs: Reset, Clk, reset, go, pause,

Outputs: [11:0] PipeX, [9:0] PipeY

Inouts: ∅

Description: Takes care of the first bottom pipe

Purpose: The purpose of the module is to create the first bottom pipe by assigning its height and width, and also to make it move depending on the current state of the game.

Module: PipeLT.sv

Inputs: Reset, Clk, reset, go, pause

Outputs: [11:0] PipeX, [9:0] PipeY

Inouts: ∅

Description: Takes care of the first top pipe

Purpose: The purpose here is to instantiate the first top pipe, it assigns a width, a height and a size for gap that separates the top and bottom pipes. It also updates the pipe's position during the game based on the state.

Module: PipeRB.sb

Inputs: Reset, Clk, reset, go, pause

Outputs: [11:0] PipeX, [9:0] PipeY

Inouts: ∅

Description: Takes care of the second bottom pipe

Purpose: The purpose of PipeRB.sv is to instantiate a second example of a bottom pipe. There again, a height and width are assigned, along with the distance from the first pipe instantiated. X position updated based on state of the game.

Module: PipeRT.sv

Inputs: Reset, Clk, reset, go, pause

Outputs: [11:0] PipeX, [9:0] PipeY

Inouts: ∅

Description: Takes care of the second top pipe

Purpose: The purpose of this module is to instantiate the top part of the right pipe. It sets its height to a fixed value, as well as its width and start position. In the module we also take care of getting the pipe in motion (from right to left) and loop back to the beginning of the screen when it finishes a first loop.

Module: Pipe3B.sv

Inputs: Reset, Clk, reset, go, pause

Outputs: [11:0] PipeX, [9:0] PipeY

Inouts: ∅

Description: Takes care of the third bottom pipe

Purpose: The purpose for this module, like the purpose of PipeLB and PipeRB, is to instantiate a new bottom-pipe object: the height, the width and the start x_position are set. Once again the x_position of the moving pipe is updated.

Module: Pipe3T.sv

Inputs: Reset, Clk, reset, go, pause

Outputs: [11:0] PipeX, [9:0] PipeY

Inouts: ∅

Description: Takes care of the third top pipe

Purpose: The purpose of Pipe3T.sv is to create a new instance of the top pipe. The constants are set for the object and the implementation to make it move along the x axis is also included in this module.

Module: Pipe4B.sv

Inputs: Reset, Clk, reset, go, pause

Outputs: [11:0] PipeX, [9:0] PipeY

Inouts: ∅

Description: Takes care of the fourth bottom pipe

Purpose: The purpose is the same as the one explained in the PipeLB, PipeRB and Pipe3B sections.

Module: Pipe4T.sv

Inputs: Reset, Clk, reset, go, pause

Outputs: [11:0] PipeX, [9:0] PipeY

Inouts: ∅

Description: Takes care of the fourth top pipe

Purpose: The purpose is the same as the one explained in the PipeLT, PipeRT and Pipe3T sections.

Module: Pipe5B.sv

Inputs: Reset, Clk, reset, go, pause

Outputs: [11:0] PipeX, [9:0] PipeY

Inouts: ∅

Description: Takes care of the fifth bottom pipe

Purpose: The purpose is the same as the one explained in the PipeLB, PipeRB and Pipe3B sections.

Module: Pipe5T.sv

Inputs: Reset, Clk, reset, go, pause

Outputs: [11:0] PipeX, [9:0] PipeY

Inouts: ∅

Description: Takes care of the fifth top pipe

Purpose: The purpose is the same as the one explained in the PipeLT, PipeRT and Pipe3T sections.

Module: Pipe6B.sv

Inputs: Reset, Clk, reset, go, pause

Outputs: [11:0] PipeX, [9:0] PipeY

Inouts: ∅

Description: Takes care of the sixth bottom pipe

Purpose: The purpose is the same as the one explained in the PipeLB, PipeRB and Pipe3B sections.

Module: Pipe6T.sv

Inputs: Reset, Clk, reset, go, pause

Outputs: [11:0] PipeX, [9:0] PipeY

Inouts: ∅

Description: Takes care of the sixth top pipe

Purpose: The purpose is the same as the one explained in the PipeLT, PipeRT and Pipe3T sections.

Module: bird_ROM.sv

Inputs: [6:0] x, [6:0] y

Outputs: [2:0] out

Inouts: ∅

Description: Outputs “colored” image of the bird

Purpose: The purpose of this module is to hold a 2D array that holds the values (RGB-corresponding value) of the color for each pixel of the bird.

Module: wholePipeBot_ROM.sv

Inputs: [10:0] x, [10:0] y

Outputs: [2:0] out

Inouts: ∅

Description: Outputs “colored” image of the bottom pipe

Purpose: The purpose of this module, like the purpose of the previous module, is to hold a 2D array containing the color code of each pixel of the bottom pipe.

Module: wholePipeTop_ROM.sv

Inputs: [10:0] x, [10:0] y

Outputs: [2:0] out

Inouts: ∅

Description: Outputs “colored” image of the top pipe

Purpose: The purpose of the wholePipeTop_ROM module is hold the value of the color code of every pixels that make up the drawing of the top pipe.

Module: font_rom.sv

Inputs: [10:0] addr

Outputs: [7:0] data

Inouts: ∅

Description: Stores correct ROM depiction to display

Purpose: The purpose of the module is to hold the representation of numbers 0 to 9, capital letters A to Z, and lowercase a to z. Given an address the module stores the "value" stored at that address into the output variable "data" that will be reused when displayed on the monitor.

Module: title_ROM.sv

Inputs: [10:0] x, [10:0] y

Outputs: [3:0] out

Inouts: ∅

Description: Displays title to VGA

Purpose: The purpose of this module is to hold the ROM data for the title screen, that is two-dimension array holding the color (represented as a number) for each pixel of the 101x302 title.

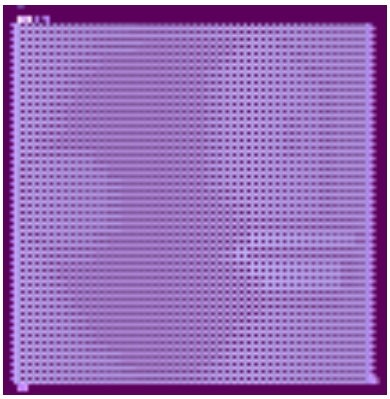


Figure 6: overview of the ROM file for the bird

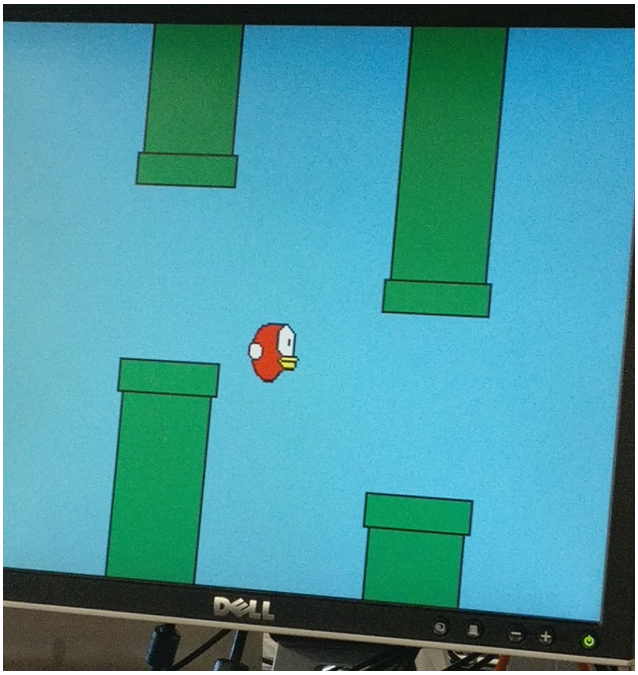


Figure 7: Final display

F. Bugs and Issues Encountered

We encountered numerous bugs while working on our final project, all of which we managed to fix one way or another, usually by just reviewing some part of the code or sometimes by using a completely different method.

Most of the trouble came from the sprites. At first instead of having the bird displaying, we had a yellow or white square, which did not seem to correspond to any part of the bird; to fix this we had to reshape our understanding of how the Color_Mapper works. Indexing into the "array" based on the DrawX and DrawY was not working. After drawing things out, it became apparent that we needed to use an offset (ex. Declare a variable Draw_Pixel_Bird_X, assign Draw_Pixel_Bird_X = DrawX - Bird_X, then utilize that variable as X when instantiating the associated sprite/ROM).

The same problem happened with the pipes. In fact, the top pipes would move along the x axis perfectly fine, but the actual drawing would not display and thus we would only see a yellow rectangle with height changing randomly at every clock cycle (i.e. 20 ns) and sometimes part of the rectangle would be green (technically the right color but not the right design) as if the bottom pipes, that were not being displayed at the time, would be at the top of the screen, overlapping the actual top pipes. The first issue we addressed was the random height changing quickly. The reason why this happened was because we used the random number generator to get the pipes' heights changing when coming back on the screen, that way we could only instantiate two pairs of pipes and get different random heights as if we instantiated an endless number of sets. However, instead of changing while the pipe was not on screen (if (pipeX > 0

&& pipeX < 639) height is fixed to last randomly generated number, else height is set to any randomly generated number.), the height would change while being displayed causing the the pipe to flicker. After trying multiple times to change the code in order to fix the bug, we decided to instead set the pipes height to fixed values and instantiate more than two to keep the game from being too easy. Next, we took care of the rectangles that appeared instead of the actual pipe designs. We got the right design to appear by utilizing the same offsets that were making only a yellow square appear for the bird (explained above), but they were piling up on the bottom instead of having the top pipe and the bottom pipe where they belonged. To fix the problem we had to rethink the way we were looping our pipes.

Looping the pipes seemed easy enough because the pipes each had the same space between each other. We just had to make sure the x variable had enough bits to hold all of the pipes' starting locations off screen. However, for some reason, passing in values and offsets to each of the modules was problematic. The variables were consistently appearing as garbage values when displayed on the hex displays. After a lot of trial and error, the solution simply came from using a lot of math and constants. We realized that the less variables we were passing into the modules, the less could go wrong. So instead of utilizing gap sizes and offsets with the pipe net variables (which was much cleaner), we had to manually keep track of where the other pipes would be when a given pipe should loop back and reappear. Lo and behold, it worked.

This lesson of utilizing constants and arithmetic as opposed to passing in values everywhere was a valuable one. When we had major problems at the end with collision detection, it made solving that go from extremely painful to just moderately painful. The fact

that constants (when given enough bits) really could not be garbage values was an instrumental, albeit messy, realization that was imperative for the success of our pipe looping and our bird and pipe/frame collision detection.

Figure 8: Issue 1: White Square instead of Bird

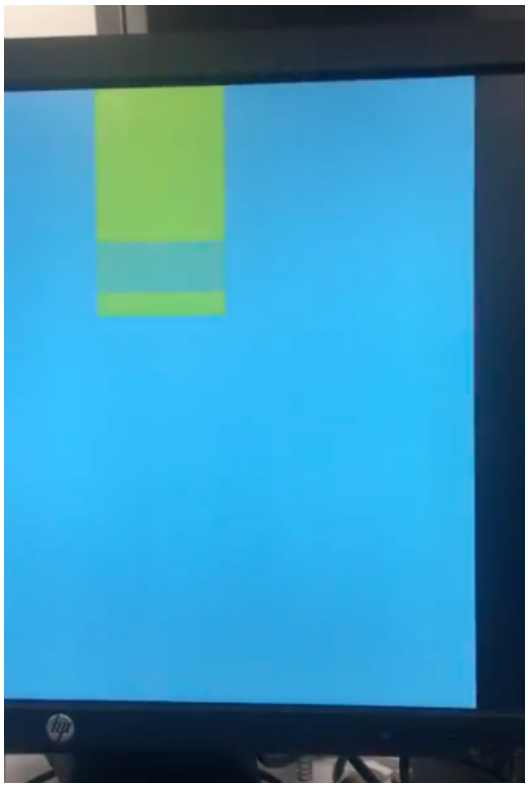


Figure 9: Issue 2: Flickering yellow pipe

III. Conclusion

Although all of our errors came from different files and different aspects of our implementation of Flappy Bird, they all appeared on the VGA display. Sometimes, it would actually be helpful in order to fix it. However, some other times it would make it more confusing, and thus harder to fix. A good thing about re-creating Flappy Bird in this specific context and course was that we got the chance to strengthen our knowledge and understanding of the way hardware and software interface through the FPGA board, and the VGA and USB components.

The project was an ultimately rewarding experience, but it did not come without hours and days of frustration. The lack of printing to debug problems like a software language

obviously makes life difficult. Each large problem we encountered took approximately a 24-hour period to fix (which adds up). Printing to the hex displays was crucial, although it was limited by space availability and compile time. Compile time was what made the project so difficult to debug at the end. Even when using only hardware for a majority of the debugging stage to eliminate variability and speed up revisions, the end of the project naturally plagues groups with long compile times. Waiting ten-fifteen minutes between often minute changes can really make the project a time sink.

Although software compatibility, general hardware/display errors, and long compile durations made the project a drag at times, when things came together and worked, it felt very rewarding. Beyond that, we realized how things that were taking us a whole day in the beginning of the project were solved in a half hour by the end. Ultimately, we learned a lot about hardware design and overall time management from this project.