ECE 385

Spring 2019

Final Project

Final Project Report:
Flappy Bird

Rob Audino and Soham Karanjikar
Section ABJ, Friday 2:00-4:50
TA: David Zhang

Introduction:

For our final project, we decided to implement the game of Flappy Bird on the FPGA using System Verilog. The game is displayed on a VGA monitor, and is controlled by a PS/2 keyboard. It also features a sound effect that plays when the bird dies, implemented through the audio codec also found on the DE2.

In order to start the project, we began with our completed Lab 8 Project (the bouncing ball), and changed it little by little until it resembled the Flappy Bird game. The first thing that we did was replace the ball with the sprite of the bird. We found the sprite on the internet, and converted the png file into a txt file using Rishi's Python program so that it would be able to be loaded into the game. This sprite was stored in on chip memory. When this was successful, we attempted to insert a full screen background into the game, but the sprite was so large that we were unable to store it in the on-chip memory. As a result, we opted instead for a simple light blue background, implemented in a similar way to the background in lab 8. Our implementation of the pipes involved a predetermined sequence of pipes with the gaps at different heights that would generate on the right end of the screen and move towards the left end of the screen. Since lab 8 gave us a basic understanding of boundary mechanics, we were able to adapt this to the rules and physics of the flappy bird game.

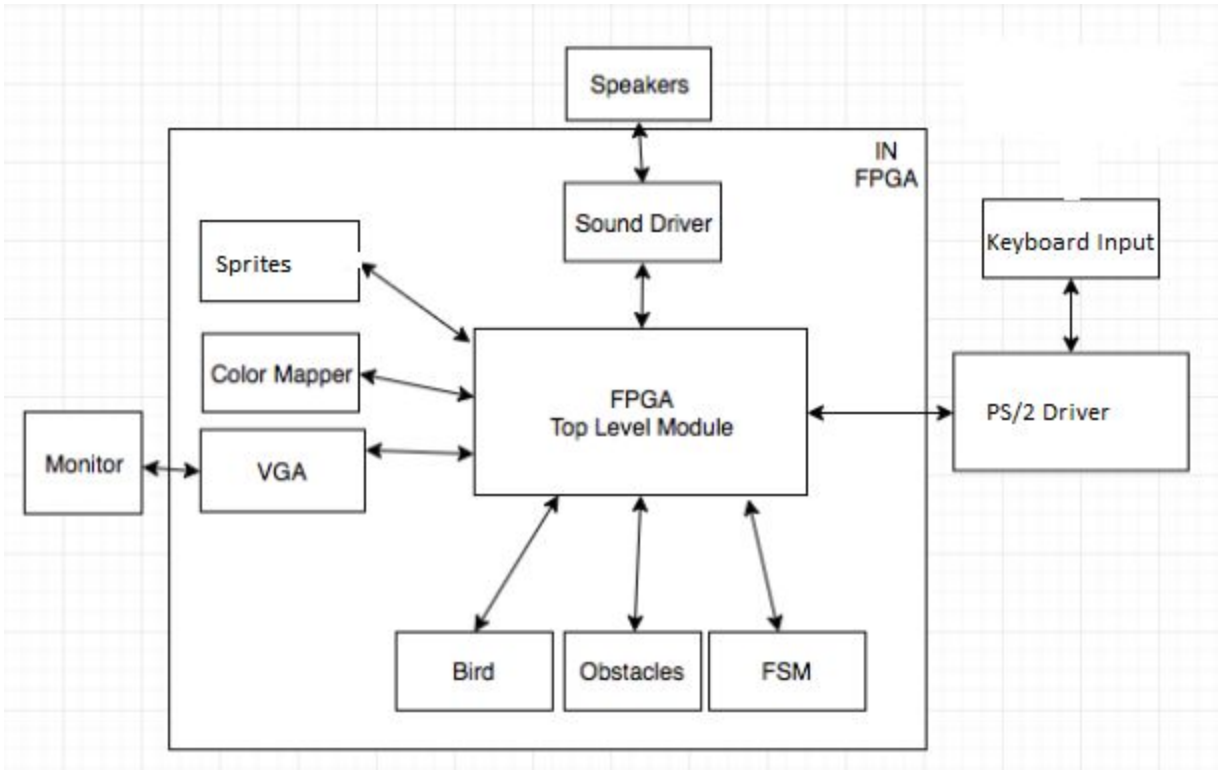
To add difficulty, we decided to use a PS/2 keyboard input. After downloading the driver provided on the Final Project ECE 385 Wiki page, we were easily able to adapt our code to accommodate for the new input. We also wanted to add difficulty through using a sound output to indicate when the bird died, and did this through the Verilog code given on the CD associated with the DE2 board.

The last part of the game that we ended up adding included the score counter. The score counter is shown on the hex displays on the DE2 board.

Game Rules and Physics:

The premise of the game is fairly simple: use the "s" key to make the bird jump, and avoid touching the pipes or the ground for as long as possible. The gap between the pipes appears at various heights, and as a result, the player must move the bird to reconcile where the bird currently is to where the bird needs to be in order to pass through the gap between the pipes. If the bird touches the ground, or the sides of the pipes, or the ends of the pipes, the game is over immediately. The score counter indicates how many pairs of pipes the player has successfully surpassed: for example, if the player has passed 2 pairs of pipes, but dies in between the 3rd pair, the final score will be only 2, as the player did not pass all the way through the 3rd pair of pipes. The difficulty in the game is dealing with gravity and judging how far the bird can fall or how high the bird can fly in order to avoid the pipes or the ground. In addition to the normal game physics, we also added a border at the top of the screen; however, the bird simply bounces off the top, and does not die.

Design Architecture:



We decided to have our design architecture strongly resemble that of lab 8, but with the USB and NIOS blocks removed and replaced with the PS/2 driver (since we are using a PS/2 keyboard instead of a USB keyboard). We decided against using the frame buffer that we suggested in the project proposal, and instead opted to use the fixed function, allowing us to keep our game within the space of a single screen, and creating the need for hardware to facilitate the moving pipes. Thankfully, since we didn't need to use that many sprites, the fixed function method wasn't pushed to its performance limits, and as a result, we were able to save memory by not having to save the entire contents of the screen into memory. We utilized the given PS/2 driver code, and

Written Description of Circuits:

PS/2 Protocols:

We decided to replace the USB keyboard input with a PS/2 keyboard input in order to earn more difficulty points. The PS/2 protocol favors input to computer transmission over computer to input, so this works well in our case since we use it for input to computer transmission. The way the PS/2 keyboard works is that it sends a code comprised of at least one 8 bit word whenever any key is pressed, held down, or released. The PS/2 also doesn't need to respond to continuous polling, and as a result saves power compared to a USB keyboard. Although the PS/2 rollover capacity is much higher than that of the USB keyboard (which can only handle 6 simultaneous key presses), the PS/2 requires a much more complex

implementation on the programmer's part to facilitate this rollover. However, in our case, this added complexity is irrelevant, as flappy bird only requires one key to be pressed at a time. Further, the PS/2 keyboard also has a signal that tells if a key is being pressed down or not. This function helps a lot since you do not want the bird to keep flying up when you press the jump key only once. The PS/2 also requires no external processor like NIOS to run and means that the overall speed is faster since there is no external communication other than direct pins.

Sound Operation:

The way we implemented sound did not need too much modification from the DE2 manual demo. We just wanted a simple noise made whenever we hit a key. The main reason for this was because we just wanted to work with the WM8731 codec and see if we can make good use out of it. Through many hours of work we figured out how it works. It runs on a state machine that waits for a signal that allows you to write data into 2 channels: left and right, each corresponding to a side of headphones. Each of these signals for our case was 16 bits as we wanted a simple sound and the wave we made did not need anymore points. After writing to the registers, the set wave is sent as an output to the speakers which can then be heard in the headphones. Since the Audio state machine and signals run on its own clock, we had to implement another PLL however it was very simple as it is generated through a very user friendly GUI. We would also genuinely like to thank the TA's and peers who helped us get through some major issues, only because of them did we figure out sound in time for our demo.

Color Mapper and VGA:

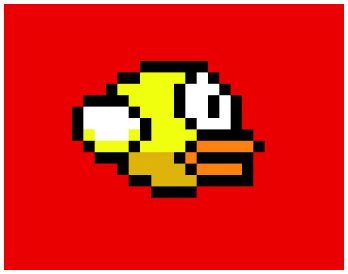
Color Mapper is used very similarly in our final project as in Lab 8, with the additional functionality of reading the color data of the sprites that is stored in RAM. At the beginning of our color mapper file, we define the size of the sprites so that the color mapper "colors within the lines". We defined variables "is_bird" and "is_pipe" to define where the color mapper should draw the appropriate colors. Since the text files that store the sprites have converted the colors to numbers, we read each of these numbers and assign the appropriate RGB value to each of the numbers read from the file. If there is an area on the screen that is not pipes or the bird, we simply assign it to the background colors.

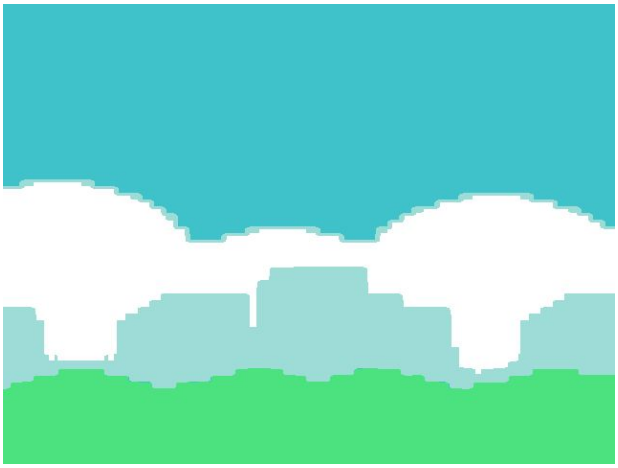
The VGA controller functions exactly the same as the one in Lab 8 in that it traverses the pixels of the screen in row major order and assigns the appropriate color to the corresponding pixel. The colors are dictated by the Color Mapper and the position of the sprites are dictated by the bird and sprite modules, all of which act as inputs to the VGA controller.

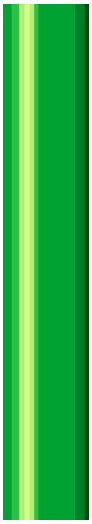
Sprites:

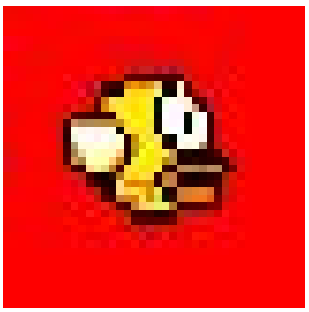
We used the ROM in the on-chip memory to store all of the sprites we utilized in this project. The 40x40 bird, 640x480 background, and 640x80 pipes are all stored in text files, which have been converted from png files via Rishi's python code. We found the pictures in png form of the appropriate size on the internet, and cropped them until they only included the items we needed. We created separate modules for each of these sprites, and calculated the exact number of bits necessary to store each sprite based on the colors used and the size of the sprite.

Sprites before converting to a color palette and after.









Game Features and Implementation:

Background:

The background is a simple skyline with rough outlines of buildings. It takes up the entirety of the screen, other than the space occupied temporarily by the moving bird and pipes.

Bird:

The bird is a sprite overlaid onto a 40 x 40 square. We centered the bird in the middle of the screen, both vertically and horizontally, and we limited the movement of the bird to the Y direction, as the bird can only jump or fall. We set the bird's movement to be default negative Y, to simulate gravity. When the user presses the "s" button on the keyboard once, the bird jumps up 10 pixels, and then gravity takes over to force the bird to fall. Although it looks as though the bird is snaking its way through the pipes, it is really the pipes that are moving towards the bird. The birds motion is contingent on the current state of the game. If the current state is "halted", the game has not yet begun and the bird is still in the center of the screen, with no pipes coming toward the bird. As soon as the "s" key is pressed, the "halted" state If the current state is "alive", the bird can continue to move up and down and navigate through the pipes. If the bird collides with a pipe or with the ground, the user is no longer able to propel the bird upward and the game ends, as the game has entered the "dead" state.

Pipes:

The pipes are sets of sprites (80 pixels wide and 480 pixels tall) that spawn at the right of the screen and move towards the left of the screen in order to provide an obstacle to the movement of the bird. There is always a consistently sized gap between the pipes, both vertically and horizontally; however, the height at which the gap appears changes with each pair of pipes that spawns. Dealing with this height difference provides the difficulty in the game. The movement on the pipes is a constant speed to the left, but only proceeds if the "current state" of the game is alive. If at any point the bird collides with a pipe or with the ground, the movement and generation of pipes immediately stops and the game is end, since the game has entered the "dead" state.

Score Counter:

Instead of displaying the current score in the middle of the screen itself, we instead opted to show it on the hex displays on the DE2 board that showed the keycodes of the keys being pressed in lab 8. This feature makes it easier on us, since we don't need to worry about implementing sprites of all 10 digits and combining them to make numbers greater than 10. As a result, this improves our performance and reduces the amount of space we need to use in memory for this function, since the sprites would have to be stored in on-chip memory. Although this counter has an upper limit of 99 because of the two digit limit on the hex displays, it is very

unlikely that the user will progress this far, and as a result, we surmised that the hex displays on the FPGA board would be enough to display the score of most users.

Written Description of Modules:

Module: counter
Inputs: [7:0] in, Clk, Reset, [9:0] pipex, killed
Outputs: [7:0] out
Description: This module serves as the counter for the game.
Purpose: This module allows the user to see how far they have gone in the game.

Module: controller
Inputs: Clk, start, Reset, killed
Outputs: [1:0] currentstate, finalkilled
Description: This module is used to moderate the three possible states in the game: alive, dead, and halted.
Purpose: This module stops the game when the bird dies and allows it to continue as long as the bird is alive.

Module: flip_flop
Inputs: Clk, Reset, Load, D
Outputs: Data_Out
Description: This module allows the user to instantiate a D flip flop.
Purpose: This module is the building block of the registers used in this project.

Module: is_killed
Inputs: Clk, Reset, [9:0] birdy, [9:0] pipex, [9:0] pipey
Outputs: killed
Description: This module decides when the bird dies.
Purpose: This module is used to stop the motion and generation of the pipes as well as the motion of the bird when the bird dies.

Module: pipe
Inputs: Clk, Reset, frame_clk, [9:0] DrawX, [9:0] DrawY, [1:0] currentstate, [7:0] score, [1:0] speed
Outputs: is_pipe, [9:0] pypos, [9:0] pxpos
Description: This module generates the pipes in the game and moves them to the left.
Purpose: This module is used to generate the obstacles in the game and is the main source of difficulty in the game.

Module: tristate
Inputs: Clk, tristate_output_enable, [15:0] Data_write, [15:0] Data
Outputs: [15:0] Data, [15:0] Data_read
Description: This module creates a tristate buffer.
Purpose: This allows us to not have a 0 or 1 but rather a NOINPUT. This means that we can have inputs to the bus that don’t matter for the current state.

Module: color_mapper
Inputs: is_bird, Clk, is_pipe, Reset, killed, [9:0] birdy, [9:0] DrawX, [9:0] DrawY, [9:0] pipex, [1:0] currentstate
Outputs: [7:0] VGA_R, [7:0] VGA_G, [7:0] VGA_B
Description: This module decides which color is outputted to each pixel of the VGA monitor.
Purpose: This module creates what we see on the screen and allows us to play the game.

Module: HexDriver
Inputs: [3:0] In0
Outputs: [6:0] Out0
Description: This module takes the inputs to drive the hex LEDs.
Purpose: This module drives the Hex Displays that displays the outputs of the thing we want. Without this, the Hex Displays would display random stuff, or just not work.

Module: lab8
Inputs: CLOCK_50, PS2_CLK, PS2_DAT, AUDIO_CLK, [3:0] KEY, [15:0] OTG_DATA, OTG_INT, [31:0] DRAM_DQ, [1:0] SW, AUD_ADCDAT, AUD_ADCLRCK, AUD_BCLK, AUD_DACLRCK, I2C_SDAT
Outputs: [6:0] HEX0, [6:0] HEX1, [6:0] HEX2, [6:0] HEX3, [6:0] HEX4, [6:0] HEX5, [7:0] VGA_R, [7:0] VGA_G, [7:0] VGA_B, VGA_CLK, VGA_SYNC_N, VGA_BLANK_N, VGA_VS, VGA_HS, [15:0] OTG_DATA, [1:0] OTG_ADDR, OTG_CS_N, OTG_RD_N, OTG_WR_N, OTG_RST_N, [31:0] DRAM_DQ, [1:0] DRAM_BA, [3:0] DRAM_DQM, DRAM_RAS_N, DRAM_CAS_N, DRAM_CKE, DRAM_WE_N, DRAM_CS_N, DRAM_CLK, [7:0] LEDG, [15:0] LEDR, AUD_ADCLRCK, AUD_BCLK, AUD_DACDAT, AUD_DACLRCK, AUD_XCK, I2C_SCLK, I2C_SDAT
Description: This module is the top level file for the whole project.
Purpose: This module integrates all of the other modules to create the project.

Module: VGA_controller
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 helps to interface the FPGA with the VGA monitor.
Purpose: This module allows us to see the game on the screen.

Module: ram
Inputs: [10:0] read_address, Clk
Outputs: [3:0] data_Out
Description: This module is used to create memory to store the sprite of the bird.
Purpose: This module stores the image of the bird.

Module: background
Inputs: [18:0] read_address, Clk
Outputs: [3:0] data_Out
Description: This module is used to create memory to store the sprite of the background.
Purpose: This module stores the image of the background of the game, providing a setting to the game.

Module: piperam
Inputs: [15:0] read_address, Clk
Outputs: [3:0] data_Out
Description: This module is used to create memory to store the sprites of the pipes.
Purpose: This module stores the image of the pipes, which provide the obstacles to the game.

Module: startram
Inputs: [11:0] read_address, Clk
Outputs: [3:0] data_Out
Description: This module is used to create memory to store the starting screen of the game.
Purpose: This module stores the opening screen of the game, welcoming the user to the game.

Module: bird
Inputs: Clk, Reset, frame_clk, killed, [9:0] DrawX, [9:0] DrawY, [7:0] keycode, [7:0] press, [1:0] currentstate
Outputs: is_bird, [9:0] birdouty, [10:0] birdy10pos
Description: This module creates and monitors the behavior of the bird.
Purpose: This module dictates the movement of the bird in the game, and allows the user to progress through the game.

Module: reg_11
Inputs: Clk, Reset, Shift_In, Load, Shift_En, [10:0] D
Outputs: Shift_Out, [10:0] Data_Out
Description: This module creates an 11 bit register.
Purpose: This module is used to interface with the PS/2 keyboard.

Module: Dreg
Inputs: Clk, Load, Reset, D
Outputs: Q
Description: This module creates a 1 bit register.
Purpose: This module is used to interface with the PS/2 keyboard.

Module: keyboard
Inputs: Clk, psClk, psData, reset
Outputs: press, [7:0] keyCode

Description: This module serves as the interface between the DE2 board and the PS/2 keyboard.
Purpose: This module allows the user to use the PS/2 keyboard to control the bird in the flappy bird game.

Module: adio_codec
Inputs: key1_on, key2_on, key3_on, key4_on, [1:0] iSrc_Select, iCLK_18_4, iRST_N, [15:0] sound1, [15:0] sound2, [15:0] sound3, [15:0] sound4, instru
Outputs: oAUD_DATA, oAUD_LRCK, oAUD_BCK
Description: This module integrates the program with the speaker by means of the audio codec on the DE2 board.
Purpose: This module is used to produce the sound for the project.

Module: AUDIO_DAC
Inputs: [7:0] iFLASH_DATA, [15:0] iSDRAM_DATA, [15:0] iSRAM_DATA,. [1:0] iSrc_Select, iCLK_18_4, iRST_N
Outputs: [19:0] oFLASH_ADDR, [22:0] oSDRAM_ADDR, [18:0] oSRAM_ADDR
Description: This module performs the Digital to Analog conversion of the audio
Purpose: This module is necessary to get the audio that was generated by the board out of the speakers.

Module: I2C_AV_Config
Inputs: iCLK, iRST_N, I2C_SDAT
Outputs: o_I2C_END, I2C_SCLK, I2C_SDAT,
Description: This module configures the I2C protocol.
Purpose: This module allows interaction with the DAC necessary for audio production.

Module: I2C_Controller
Inputs: CLOCK, [23:0] I2C_DATA, GO, RESET, W_R, I2C_SDAT
Outputs: I2C_SDAT, I2C_SCLK, END, ACK, [5:0] SD_COUNTER, SDO
Description: This module serves as a master-slave controller.
Purpose: This module allows interaction with the DAC necessary for audio production.

Module: staff
Inputs: [7:0] scan_code1, [7:0] scan_code2, [7:0] scan_code3, [7:0] scan_code4
Outputs: [15:0] sound1, [15:0] sound2, [15:0] sound3, [15:0] sound4, sound_off1, sound_off2, sound_off3, sound_off4
Description: This module configures the audio channels.
Purpose: This module controls the layers of audio produced by the board.

Module: VGA_Audio_PLL
Inputs: areset, inclk0
Outputs: c0, c1, c2

Description: This module creates a phase locked loop that is used for audio synthesis.
Purpose: This module is necessary to generate stable frequencies in audio production.

Module: wave_gen_brass
Inputs: [5:0] ramp
Outputs: [15:0] music_o
Description: This module is used to simulate the sound of brass instruments through sine waves.
Purpose: This module is used to produce the frequencies for one of an array of sounds.

Module: ramp_wave_gen
Inputs: [5:0] ramp
Outputs: [15:0] music_o
Description: This module is used to produce a sawtooth wave.
Purpose: This module is used to produce the frequencies for one of an array of sounds.

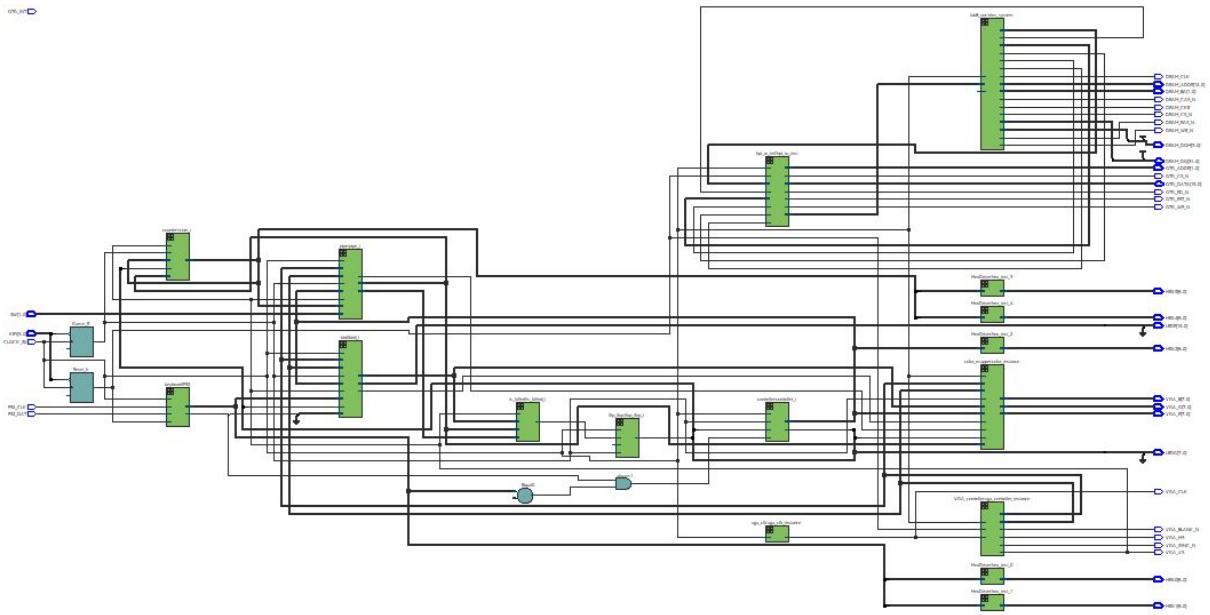
Module: wave_gen_sin
Inputs: [5:0] ramp
Outputs: [15:0] music_o
Description: This module is used to produce a sine wave.
Purpose: This module is used to produce the frequencies for one of an array of sounds.

Module: wave_gen_square
Inputs: [5:0] ramp
Outputs: [15:0] music_o
Description: This module is used to produce a square wave.
Purpose: This module is used to produce the frequencies for one of an array of sounds.

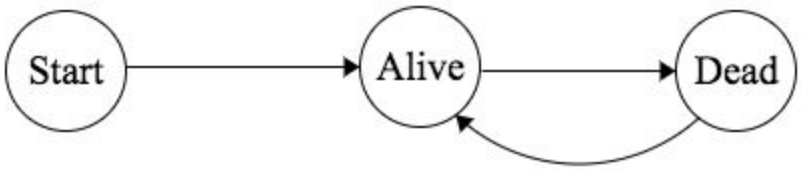
Module: wave_gen_string
Inputs: [5:0] ramp
Outputs: [15:0] music_o
Description: This module is used to simulate the sound of a string section through the combination of sine waves.
Purpose: This module is used to produce the frequencies for one of an array of sounds.

Module: wave_gen_x2
Inputs: [5:0] ramp
Outputs: [15:0] music_o
Description: This module is used to create the sound of a synthesizer.
Purpose: This module is used to produce the frequencies for one of an array of sounds.

Block Diagram:



State Machine for Controller:



The state machine is very simple and is a mealy machine. The transition from “start” to “alive” only occurs once when you start the game. The key press is “S” for a start signal. The transition from “alive” to “dead” occurs each time the bird dies due to any condition and at this point the only possible states reachable are alive and dead. Only way to reach back to start is to reset. To go from “dead” to “alive” you press the same start key, “S,” again. The control signals from the state machine are sent to many different modules, the main being color mapper, bird, and pipe. The states are represented by binary bits 000 is start, 001 is alive and 010 is dead. This is visible on the HEX Displays on Board because we wanted to make sure everything was transitioning correctly and did not want to write a whole test bench for a simple unit test.

Simulations:

Since our design was based off of lab 8, simulations were not necessary to verify progress, as we were able to see on the screen whether or not what we had attempted worked or not. The sprites as well as the game physics were all verifiable through simply watching the screen to see what was going on, and we were able to adjust our code accordingly. A lot of the testing was also done using HEX LEDS and LEDRs and LEDGs on board. The reason behind not writing a test bench was simple: none of our unit tests were failing to the point where we had no idea what was going on and usually it was a simple fix that was manageable either by looking at the VGA display or the LEDs.

Design Statistics:

LUT	3679
DSP	0
Memory (BRAM)	171,392 bits
Flip-Flop	2178
Frequency	130.41 MHz (altera_reserved_tck)
Static Power	108.94 mW
Dynamic Power	89.52 mW
Total Power	293.15 mW

Difficulty Evaluations:

Here are the features included in our implementation of Flappy Bird:

- Jumping action of bird
- Bird death mechanics
- Pipe generation and scrolling
- Sound effects
- Counting operation
- Gravity
- Sprites
- PS/2 keyboard input

When we initially proposed the game, we assigned the game itself a 5 or 6 in base difficulty with another point for sound, with a total value of 7 difficulty points. However, after discussing with our TA (David), we were shocked to discover that Flappy Bird only warranted 3.5 difficulty points on the game alone. Since we were not going to settle for 4.5 difficulty points, we decided to add a PS/2 keyboard input instead of using the code given in lab 8 for the USB keyboard input to control the bird, and this gave us an additional 2.5 difficulty points. In summation, here were the difficulty points that were given to us:

Feature	**Difficulty Points**
Base Flappy Bird Game	3.5
Sound Output	1
PS/2 Keyboard Input	2.5
Total Difficulty Points	**7**

Post-Project Thoughts:

All in all, I think we accomplished the basics of the flappy bird game pretty well. There are some trivial things that we left out of our final project that are included in the original flappy bird game, but none that drastically alter how the game is played. The scrolling ground, flapping bird, and detailed background are all examples of purely cosmetic aspects of the game that are nonessential. If we had more time, we might have been able to implement these features, but we would also have probably needed to use more memory in the board, and different memory at that, making our project more complicated. We could have also implemented a voice input to control the jumping of the bird, but again this would have required more memory and also would have required one or two weeks working with DSP, in addition to obtaining the microphone necessary to create this.

Conclusion:

In summation, our final project proved to be a welcome challenge. Although we based our project on our lab 8, we had to expand upon this foundation greatly, adding functionality for both input and audio output as well as introduce both the sprites and the game physics into the project. Lab 8 really just provided a base of knowledge of how to interface with the VGA display as well as how to create the objects displayed on the FPGA. Thankfully, since we used a PS/2 instead of a USB keyboard input, we no longer needed to interface with NIOS, which saved us quite a bit of time and trouble considering how prone NIOS was to crashing. We had to study the game physics and appearance ourselves, and try to translate and implement these aspects of the game into systemverilog the best we could. The hardest part ended up being the addition of the pipes, since they not only had to spawn into the screen, they also had to move to the left and kill the bird if they touched the bird. In conclusion, this project tied up almost everything we learned and worked on during the semester and challenged us to integrate software and hardware in ways we previously had not.

Citations:

Since we used the PS/2 driver by Sai Ma and Maria Liu (given on the course website) for the PS/2 keyboard input, we need to give credit here. Additionally, we used the Verilog Audio Driver given in the CD associated with the DE2 board. We also used Rishi's Python code to convert our sprites from PNG to TXT format. All links are listed on ECE 385 website on the final project page.