Clappy Bird

ES 4 Final Project

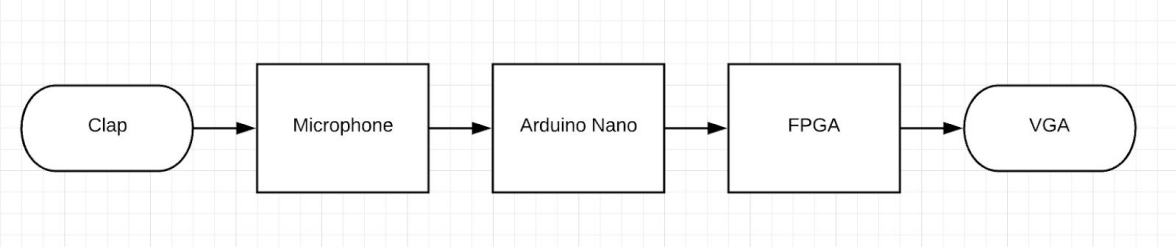
Kaavya Chaparala
Gabi Diaz
Jake Kroner
Faizan Muhammed

Contents

1 Overview

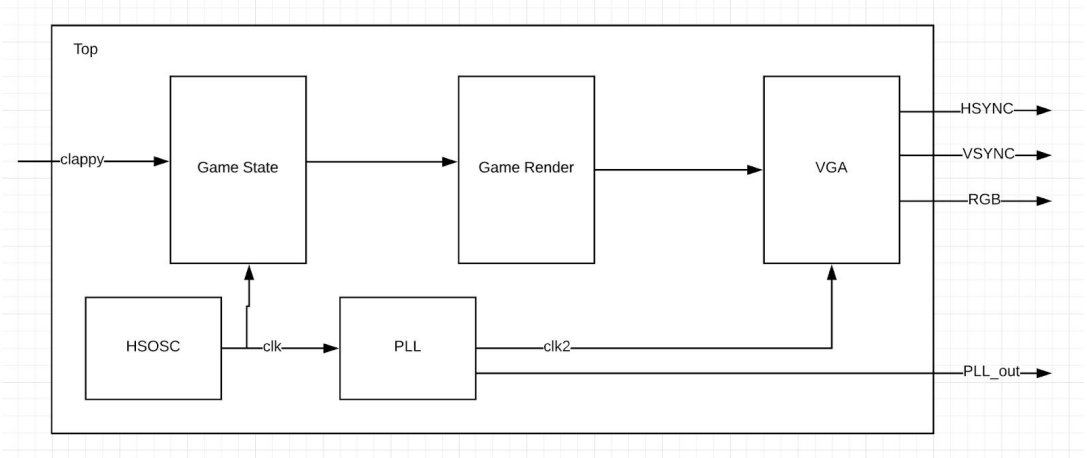
Our project idea was to implement a modified version of the game “Flappy Bird” by combining hardware components and a VGA previously used in labs. The Flappy Bird game features a bird flying through pipe obstacles. The object of the game is to fly the bird through the pipes, avoiding any collision that would cause the game to end. The aspect that we modified in our game is how the bird “flies.” In the original game, the user must tap the screen to make the bird fly up, otherwise the bird sinks to the bottom of the screen. Instead of tapping the screen, our game implements audio processing in that a “clap” or loud sound is used to make the bird fly up...hence the name “Clappy Bird.” The diagram shown in Figure 1 gives a basic representation of the components involved in the project, and how they connect to each other.

Figure 1: Basic Block Diagram of Components



The process begins with the input of a “clap” sound into the microphone. The microphone converts the sound input into an analog output which is then fed into an Arduino Nano. The Arduino processes the sound to output a digital signals of ones and zeros to symbolize clap or no clap. This digital signal is then fed into the FPGA as an input to the top level VHDL module. The block diagram for the top level module and all its submodules can be seen in Figure 2. The VHDL code is then output via VGA to project the game display on the monitor.

Figure 2: VHDL Block Diagram



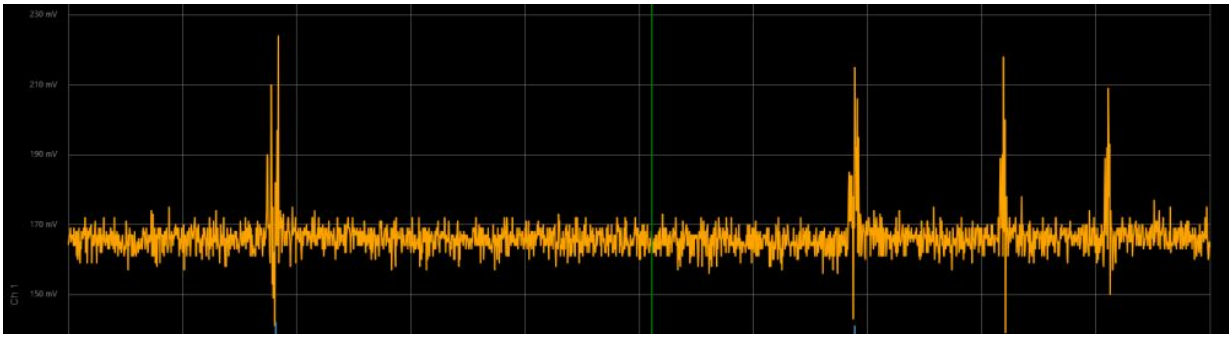
2 Technical Description

All the code and documentation for this project can be found on Github:
https://github.com/faizan-m/ClappyBird

2.1 Microphone

The first component of Clappy Bird is the microphone. The microphone takes in an audio signal and produces an analog waveform. This analog output can be seen in Figure 3. The waveform features mostly steady noise with occasional peaks. These peaks represent claps or loud noises, which are used to toggle the input to the game. The noise is filtered out by the Arduino Nano.

Figure 3: Analog Sound Signal

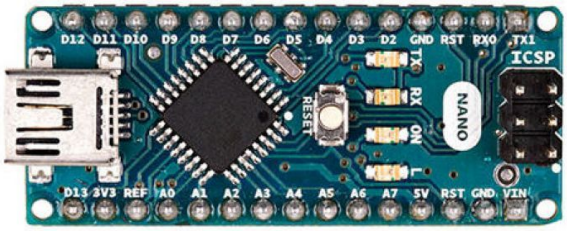


2.2 Arduino Nano

The Arduino Nano is the next component in the overall block diagram. This device inputs the analog signal and converts it into a digital output. The digital output consists of ones and zeros. A clap is represented by a one while silence, or no clap input, is represented by a zero. This conversion made it much easier to process a clap, because it allowed them to be represented as a 1-bit binary signal. Additionally, this component filters out amplitudes below a certain cutoff, so that normal noise does not raise the binary signal.

Figure 4 shows an image of the Arduino Nano. The conversion required using four pins: one pin for power, one pin for ground, one pin to take in the analog signal from the microphone, and one pin to output the binary signal.

Figure 4: Arduino Nano Schematic



Refer to C1 in Appendix for the complete code implemented in Arduino Nano. The code looks for peaks in the analog signal and then produces a “HIGH” output for 0.1 seconds corresponding to every peak. This allows for enough time for FPGA to process the peak as well as makes sure that a single clap corresponds to just one peak.

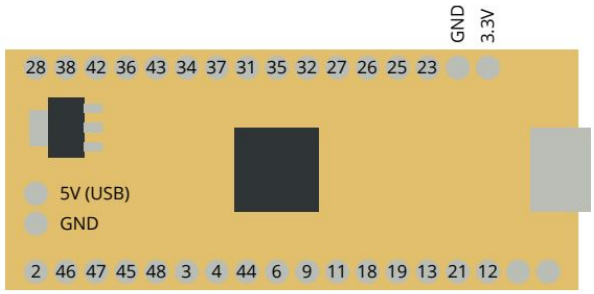
In order to test the signal, we connected the output of the Arduino Nano to an LED. After programming the device, we clapped into the microphone and saw the LED light up. We knew it worked properly because the LED did not light up when we were talking at a normal volume - only when a loud noise was emitted.

The Arduino Nano was not part of the initial design for our project. Originally, we tried to use an ADC converter to transform the signal, but we ran into a major obstacle. The ADC converter used an SPI interface with which none of us had any experience. We spent a large amount of time attempting to debug the code and set it up properly, but we were unsuccessful. Eventually, we realized that the Arduino Nano serves a very similar function, with an interface that was much easier to implement. We were successful in programming the Arduino Nano to convert analog signals into digital signals.

2.3 FPGA

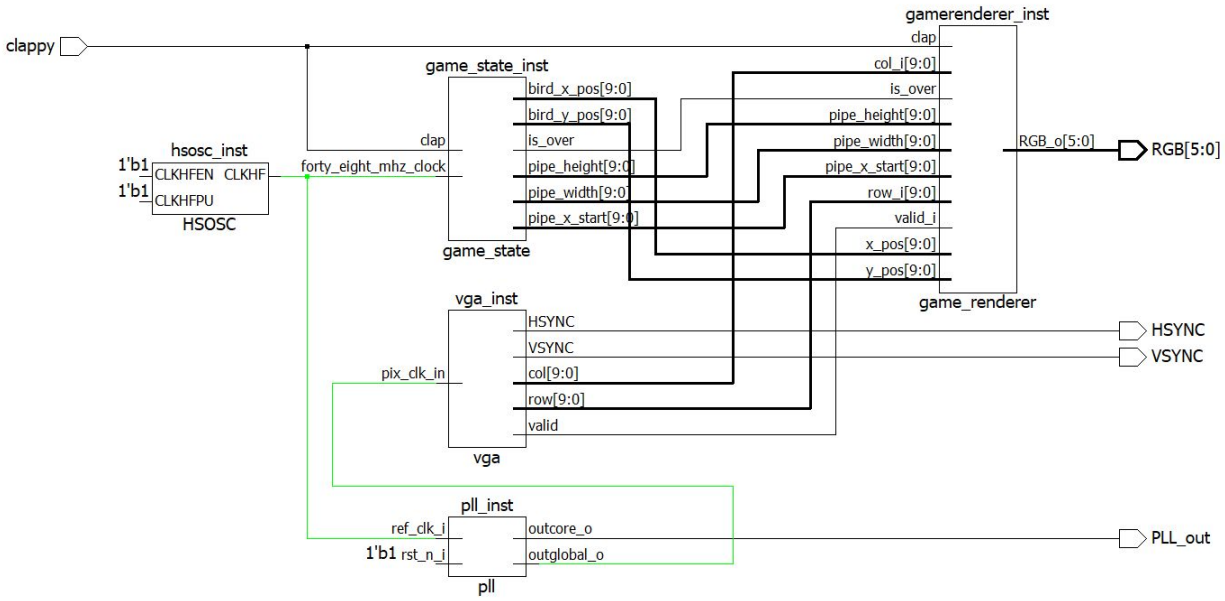
The FPGA is the essential component that brings everything together. The FPGA takes in only one input, which is the binary signal that represents the clap. The entire game is implemented using the FPGA and this binary signal. The FPGA has nine output signals, eight corresponding to pins on the VGA, and one PLL output. These pins on the VGA are Red0, Red1, Blue0, Blue1, Green0, Green1, HSYNC, and VSYNC. The layout of the FPGA can be seen in Figure 5.

Figure 5: FPGA Layout



The game itself is coded onto the FPGA so that it can generate the appropriate output signals based on the game state and the clap input. The block diagram for the game can be seen in Figure 6.

Figure 6: VHDL Block Diagram



As Figure 6 shows, there are three modules within the top entity that work together to implement the game: pll_inst, vga_inst, game_state and game_renderer. When working together, the overall goal is to take in the clap input at every clock cycle and produce the correct output for the VGA. These modules are explained in further detail below.

2.3.1 “Top” Module

The “top” module is the driver module for the game. This file lists all of the other components, creates signals for each input/output, and port maps each signal. These components include the HSOSC, PLL, vga, gamestate, and gamerenderer modules. The HSOSC module sets up the clock for the game. This has an output of CLKHF, which is used as input to the PLL module.

The PLL component then changes the frequency of this clock signal to 25.125MHz. This new frequency is used by the other components for each clock cycle. The VGA, GameState, and GameRenderer modules start, update, and display the game.

Since the “top” VHDL file is the head module, it acts like a black box and contains the outer-level inputs/outputs, as seen in Figure 7. It takes in a clap as an std_logic signal as input. At every clock cycle, it produces the six RGB values, HSYNC, VSYNC, and PLL_out as output. These outputs are connected to the VGA to display the game on the computer.

Figure 7: Top Entity

```
entity top is
  port(
        clappy : in std_logic;
        RGB : out unsigned(5 downto 0);
        HSYNC : out std_logic;
        VSYNC : out std_logic;
        PLL_out : out std_logic
  );
end top;
```

2.3.2 “VGA” Module

The VGA module is in charge of running through the pixels on the screen and updating the row number, column number, HSYNC, VSYNC, and valid values. It creates signals to specify the values of each section on the screen (horizontal front porch, vertical visible area, etc). Using these values, as well as the clock input, it generates the output values at each rising edge. The logic for the HSYNC, VSYNC, and valid bits is based upon the current row and column number, as seen in Figure 8.

Figure 8: Logic for HSYNC, VSYNC, and valid

```
HSYNC <= '0' when col_num < hoizontal_sync_pulse
          else '1';
VSYNC <= '0' when row_num < vertical_sync_pulse
          else '1';

valid <= '0' when
         col_num <(hoizontal_sync_pulse + hoizontal_back_porch) or
         col_num >= (hoizontal_whole_line - hoizontal_front_porch)
or
         row_num < (vertical_sync_pulse + vertical_back_porch) or
         row_num >= (vertical_whole_line - vertical_front_porch)
          else '1';
```

2.3.3 “GameState” Module

GameState is where all the game logic happens. It takes in the clap input and outputs the positions of the bird and the obstacle as well as if the game is over or not. Therefore, GameState is responsible for maintaining and updating all these values by incrementing and decrementing them for every frame as needed and doing collision detection to see if the bird has hit anything.

GameState has its own game_clock which runs at 45 Hz. An update to the GameState outputs is made at every cycle of this game_clock. The obstacle is updated as shown in Figure 9.

Figure 9: Updating the Obstacle

```
process(game_clock) is
begin
     if rising_edge(game_clock) then
        pipe_x_start <= pipe_x_start - pipe_speed when is_over =
'0'
                        else to_unsigned(1000,10) when is_over =
'1';
     end if;
end process;
```

This moves the obstacle to the bird and loops it right back round when the game is running but sets the obstacle back to the starting point when game is over. Here pipe_speed and 1000 are constants that define the speed at which the obstacle approaches the bird and the spawning point of the pipe. Figure 10 shows how the bird and game_over is updated.

Figure 10: Updating the bird position and game_over

```
process(game_clock) is
begin
     if rising_edge(game_clock) then
        bird_y_pos <= bird_y_pos-10 when clap = '1' and reset = '0'
                        and is_over = '0' -- go up
                        else bird_y_pos + 5 when reset = '0' and
                        is_over = '0' -- fall down
                        else bird_y_pos + 10 when  is_over = '1' and
                        bird_y_pos + 85 < 512 -- die
                         else to_unsigned(100,10) when reset = '1';

        is_over <= '0' when reset = '1'
                    else '1' when screen_colliding or
beak_colliding
                     or head_colliding or body_colliding or
```

```
                    tail_colliding;
     end if;
end process;
```

As we can see, the bird goes up when a clap is detected. The bird falls down when it is not. It falls down much more quickly until it hits the floor when it dies and it is spawned back at starting position when the game is reset.

The game is over when any collision is detected but it is restarted every time reset is ‘1’. The reset is updated as shown in Figure 11.

Figure 11: Restarting the Game

```
process(game_clock) is
begin
     if rising_edge(game_clock) then
          wait_counter <= "0000000000" when is_over = '0' else
                          wait_counter + 1 when is_over = '1';
     end if;
end process;

reset <= '1' when is_over = '1' and clap = '1' and wait_counter >
50
          else '0';
```

The wait_counter counts frames after the game is over. Reset waits atleast 50 frames before a clap can restart the game again. This gives about 1 second to the user so that they can see the game is over and the bird can fall down to the ground. Finally, the collisions are handled as follows in Figure 12.

Figure 12: Detecting Collisions

```
screen_colliding <= '1' when bird_y_pos + 85 > 512 or bird_y_pos +
50
                    < 100 else '0';

beak_colliding <= '1' when bird_x_pos + 100 > pipe_x_start  and
bird_y_pos + 35 < pipe_height  and  bird_x_pos + 95 < pipe_x_start
+ pipe_width else '0';

head_colliding <= '1' when bird_x_pos + 95  > pipe_x_start  and
bird_y_pos < pipe_height  and  bird_x_pos + 45 < pipe_x_start +
pipe_width else '0';
```

```
body_colliding <= '1' when bird_x_pos + 70  > pipe_x_start  and
bird_y_pos + 50 < pipe_height  and  bird_x_pos + 15 < pipe_x_start
+ pipe_width else '0';

tail_colliding <= '1' when bird_x_pos + 50  > pipe_x_start  and
bird_y_pos + 55 < pipe_height  and  bird_x_pos  < pipe_x_start +
pipe_width else '0';
```

These are based on the exact dimensions of the bird's body parts which are echoed in the GameRenderer again (See Figure 14). If any of the parts of the bird is colliding with the obstacle, the corresponding variable gets set to '1' and in the next frame cycle the game will be over.

2.3.4 "GameRenderer" Module

GameRenderer does not interfere with any of the game logic. It is purely responsible for converting the GameState's outputs into a visual format that players can use to see what is going on at any point in time. It also does not use any RAM/ROM for this purpose. The inputs given to the GameRenderer by the GameState have a clear mathematical mapping to an image that the GameRenderer generates and passes to the VGA module for display purposes.

As the VGA module polls the GameRenderer pixel by pixel, the entire scene is generated in a single line of VHDL code that decides RGB_O values depending on the pixel that is being requested and what the GameState requires it to be. The code is given as follows:

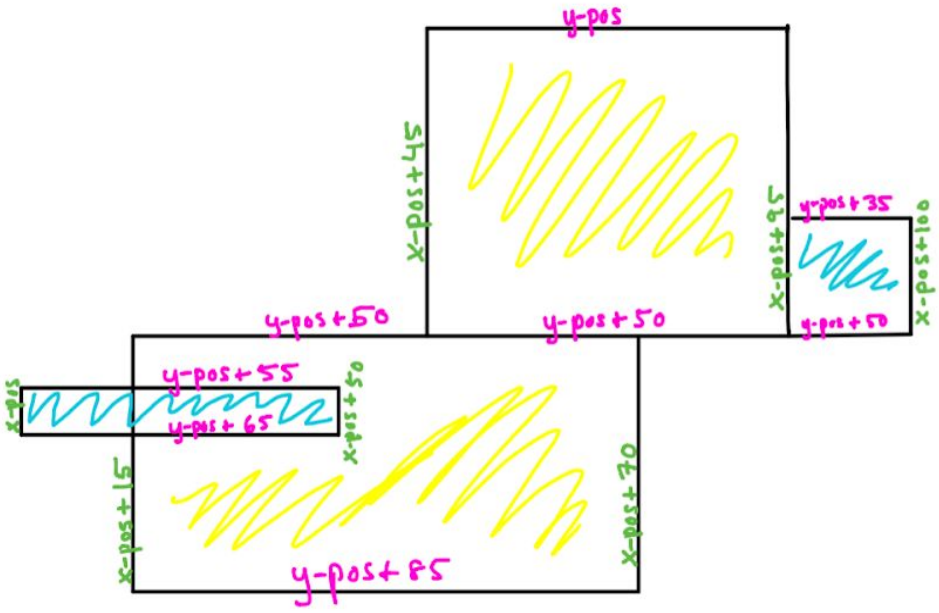
Figure 13: Rendering Graphics

```
RGB_o <=
"101110" when valid_i = '1' and (col_i > pipe_x_start  and col_i <
pipe_x_start + pipe_width) and (row_i > 0 and row_i < pipe_height)
 -- pipe

else
"001111" when valid_i = '1' and (col_i > x_pos + 45 and col_i <
x_pos + 95) and (row_i > y_pos and row_i < (y_pos + 50))
--head

else
"001111" when valid_i = '1' and (col_i > x_pos and col_i < x_pos +
50) and (row_i > y_pos + 55 and row_i < y_pos + 65)
-- wing

else
"011000" when valid_i = '1' and (col_i < x_pos + 70 and col_i >
```

```
x_pos + 15) and (row_i > y_pos + 50 and row_i < y_pos + 85)
 -- body

else
"011000" when valid_i = '1' and (col_i > x_pos + 95 and col_i <
x_pos + 110) and (row_i > y_pos + 35 and row_i < y_pos + 50)
-- beak

else
"101011" when valid_i = '1' and is_over = '1' and col_i < 1024 and
row_i < 512
-- game over background

else
"110101" when valid_i = '1' and col_i < 1024 and row_i < 512
-- normal background

else "000000";
```

The if-statements are ordered in terms of layers, the statements that come before others get drawn over the ones below. That means the background is always behind pipes and the bird. The bird was designed according to Figure 14.

Figure 14: Bird Design



2.4 VGA Adapter

The outputs corresponding to the VGA from the top module are routed to the pins on the FPGA that are connected with corresponding pins of the VGA Adapter. These pins include the 6 RGB values, as well as HSYNC and VSYNC. This adapter is connected to a monitor that finally displays the rendered image on screen.

2.5 Challenges

The challenges encountered during the construction of this program shaped our approach to the project. The first and most time consuming challenge we faced was converting the analog signal from the microphone into a digital signal that could be analyzed. Our original plan was to use an ADC to feed the converted digital signal into the FPGA. As previously mentioned, the ADC had an SPI interface that none of us had experience with, and as a result we were unable to successfully program the ADC to read in our signal. After abandoning the ADC we briefly attempted to use a high pass filter and a transistor to detect the sound of a clap, but these methods also proved unsuccessful. Finally we got the idea to use an Arduino (explained in detail in section 2.2) to detect a clap from the microphone's output.

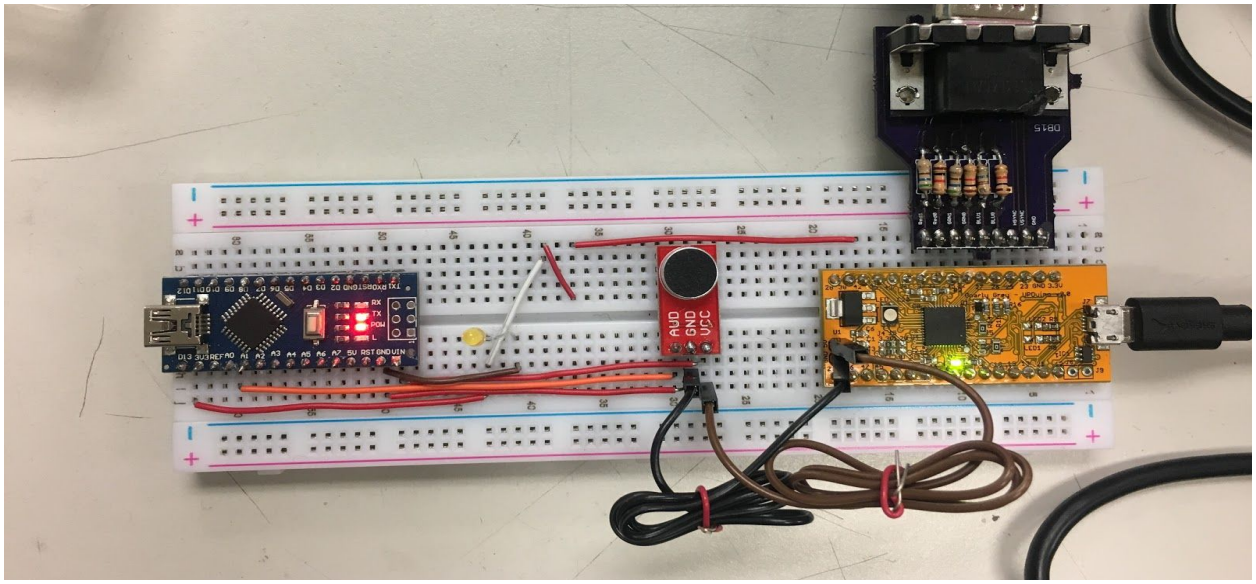
Once the VGA aspect of the project was working and we had designed a bird that would move vertically along the screen, we needed to be able to detect when the bird collided with a pipe in order to end the game. Because the shape of the bird was not a simple square but rather a conglomerate of rectangles that had all been coded separately onto the VGA, there were multiple ways that a pipe could collide with the bird. That is to say, instead of simply checking whether the pipe collided with the bird, we had to check if the pipe collided with the beak, the head, the tail, or the wing.

Coding the VGA display to show a certain image was made even harder by the fact that the monitor we used was 1280 x 960 pixels, and the VGA is set up to show a 1024 x 512 display. This discrepancy was especially problematic when we were trying to detect collisions between the bird and the upper and lower edges of the screen. Originally we assumed that the upper left-hand corner of the screen was located at (0,0), but it became clear during testing that the upper edge of the monitor was not actually located at the zeroth row. After spending time debugging we were able to realize that the upper edge of the screen had to be greater than zero, and through trial and error we found that this upper edge was 50 pixels down from row zero.

3 Results

Ultimately, our project successfully implemented our modified version of the Flappy Bird game. As planned, we were able to create a game where a “hand clap” could be used as a signal to make a bird fly. The audio processing is based on amplitude meaning that the game only detects loud sounds as inputs-- low volume conversation around the game will not affect the bird’s motion. Figure 15 shows the breadboard configuration of the game with all of the hardware components involved including the microphone, arduino nano, and FPGA.

Figure 15: Breadboard Configuration



The game begins when the first clap is detected and ends when the bird hits either a pipe, the top of the screen, or the bottom of the screen. When a game-ending collision is detected, the screen turns red to signify that the game is over and the bird falls to the bottom. Figure 16 shows an image of the display while the game is in process and figure 17 depicts the screen image after the game ends and the bird has fallen to the bottom of the display. As far as the game rendering, the VGA display shows a bird flying vertically through obstacles. In terms of game display, the main difference between our game and the original Flappy Bird is that our display only shows pipes coming from the top of the screen, and the length of these pipes is constant. Other than that, similar to how a “tap” will propel the Flappy Bird upward, Clappy Bird flies up when a clap is detected and falls to the bottom of the screen when no inputs are given.

Figure 16: Image While Game is Being Played

Figure 17: Image When Game is Over

The only bug present in our game is that if it is played in a very noisy room with loud voices nearby, the audio processing system has trouble distinguishing between claps loud voices that are in close proximity to the microphone. This causes the bird to fly up even when the user does not intend it too.

The part of our project that fell short of our original goals was our limited pipe/obstacle implementation due to time constraints. If allotted more time, we would add pipes originating from the bottom of the screen and randomize the origin and length of the pipe obstacles.

Additionally, we would fix the previously mentioned audio processing bug so that the arduino included frequency filtering for the audio signals. This filtering would make it so that the game would only process sound inputs that are within the frequency bandwidth of a hand clap, getting rid of any human voices regardless of their volume. To make the game particularly cool, we would also enforce a way to keep track of and display a score and have the game speed up as the user plays in order to increase difficulty.

4 Reflection

This group worked well together from the beginning. During the first week of the project while we were waiting for our parts to come in, we met multiple times to plan the project. This meant that when we were actually building our project we did not have to stop as often to discuss questions with the group, and we were able to make more progress in sub-groups of two because we were sticking to a predefined plan. Being able to make progress in smaller groups allowed us to have a flexible schedule; although some things came up during the week where group members could not make it to longer working sessions, the other group members were able to meet and make progress so that one person's absence did not slow down the whole project. When members returned they were caught up on the progress that was made and they then jumped in to help with the next stage of the project. When prompted with the question of what did not go well with our team, all team members agreed that nothing went badly. The only thing we would do differently if we were to do this process again would be to try to make time to ensure all members of the group thoroughly understood the project and all of its components, not just the aspects of the project that a particular member worked on.

All in all, throughout these two weeks everyone on this team had a positive attitude, a strong work ethic and a commitment to the group that made everyone feel as though each person was contributing equally.