DuckFeed: An Embedded Take on Duck Hunt
Columbia University, Spring 2011
CSEE 4840: Embedded System Design

Daniel Teger Scott Rogowski Julie Dinerman Kevin Ramkishun

May 13, 2011

Contents

Chapter 1

Overview

1.1 Introduction

DuckFeed is a slightly modified implementation of the popular classic game Duck Hunt, which was released by Nintendo^{TM} in the 1980s. The game is played on the Nintendo^{TM} Entertainment System using the patented Nintendo^{TM} Zapper to shoot images of flying ducks on a Cathode-Ray Tube Television. There are limits to the number of bullets that a player gets, as well as a time limit as to how long a duck will stay on the screen before it flies away and the opportunity to shoot it is lost.

When the ducks are killed, our friendly dog returns the corpses to us. In the original game, only two ducks appear on the screen together at any given time.

1.2 Our Incarnation

Our goal is to re-create the game (with some minor changes in game play, described below) entirely on an FPGA. This entails writing custom hardware for the FPGA that will drive a VGA screen, as well as hardware to interact with the Nintendo^{TM} Zapper. The FPGA will also contain a Nios II microcontroller, for which C code will be written to control the game logic.

Being that the assasination of ducks is violent; our version of the classic video game is based on feeding the ducks that appear on the screen. Our "food gun" is armed not with bullets, but with food. When a duck is "hit", it simply flies away, no longer hungry.

In addition, DuckFeed does not limit the player to feeding only two ducks at a time. Up to 8 hungry ducks may appear on the screen at any given time. The structure of the levels in DuckFeed are based on the number of ducks to feed, the amount of food available, and how fast the ducks fly. As the levels progress, the ducks will fly faster (making them harder to hit) and there will be more of them. In addition, the food supply and the amount of time before the ducks fly away get smaller as the game progresses.

Chapter 2

Design

2.1 Introduction

Duck Feed is a take on the classic game by Nintendo^{TM}, Duck Hunt. The game is played by a user yielding a "Zapper" gun originally created and popularized by Nintendo^{TM} in the 1980's. The user sees a varying number of Ducks appear to fly across the screen; points are accrued by successfully shooting (feeding) a Duck.

Duck Feed differs from Duck Hunt in a number of important aspects:

- Instead of being implemented for the Nintendo^{TM} Entertainment System, the game is implemented entirely on an Altera FPGA Board.
- The object of the game is to feed the ducks by shooting them with food, not to harm them!
- The game will be played on a CRT Computer Monitor, which has a different Refresh Rate than the original CRT Televisions of the 1980s (and thus requires modifications to hardware).

2.1.1 Milestones

1. Interface the Zapper hardware with the display to indicate when the trigger is pulled
2. Display simple Sprite-based animated graphics.
3. Have a bug-ridden game functioning

2.2 Overview

Duck Feed is implemented entirely on an Altera FPGA utilizing the Nios II microprocessor and custom hardware peripherals. The hardware is developed in VHDL and the software for the microcontroller is written in the C programming language. Custom registers provide the interface between the software and the hardware.

The hardware design consists of the following two Avalon Bus peripherals:

- The VGA Controller, which is tasked with drawing the stationary background for the game and rendering the Duck sprites on the screen.
- The Zapper Controller, which is responsible for interfacing with the Nintendo^{TM} Zapper: reading when the trigger is pressed as well as determining hits and misses.

The software design consists of the following modules:

- The Game Loop, which calculates the positions of the sprites on the screen and the score.
- The Zapper Interface, which communicates with the Zapper Controller hardware and lets the Game Loop know that a hit was made.
- The VGA Interface, which communicates with the VGA Controller hardware and which can set the visibility, position, and which rendering of a sprite to display on the screen.

The system is summarized by the Block Diagram in Figure 2.1.

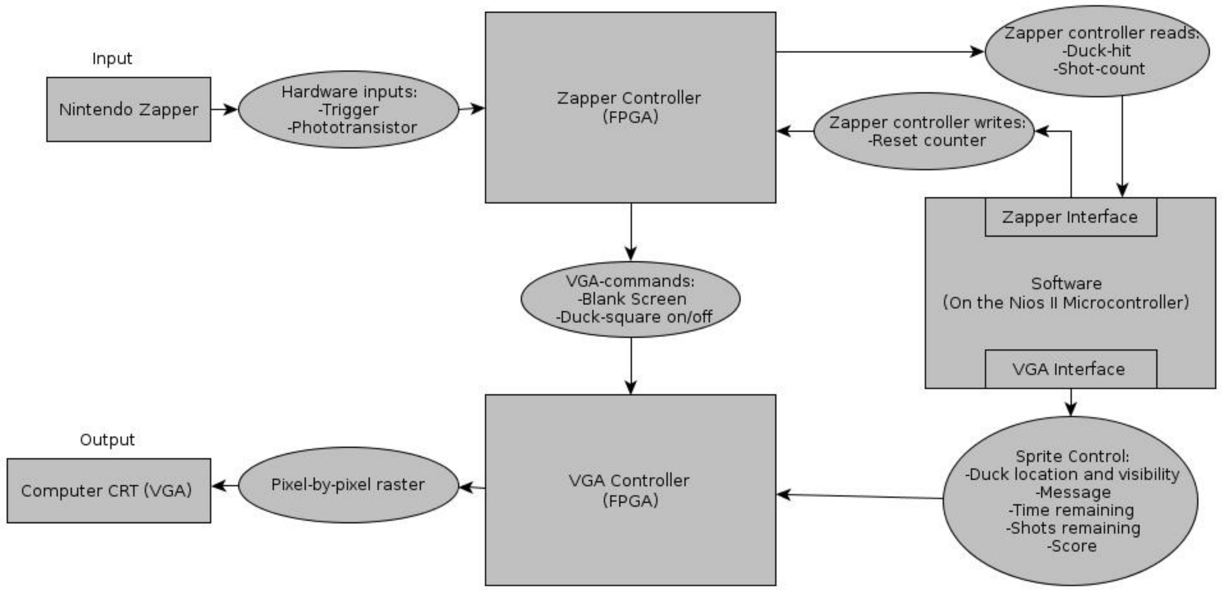

Figure 2.1: The Block Diagram of the Duck Feed system

2.3 Hardware Design

The hardware design includes custom Avalon Bus peripherals developed in VHDL and the Altera Quartus Tools, as well as modifications to the Nintendo^{TM} Zapper Light Gun. This section details all of the design decisions that are likely to be implemented in the final version of the Duck Feed game.

Note that some of the information in this section is not yet confirmed. This document shall be revised once we have the opportunity to take apart and analyze the Nintendo^{TM} Zapper. The information provided here is based on resources such as the original patent for the Zapper, as well as third-party articles and blogs dealing with customizing the Nintendo.

2.3.1 The Nintendo^{TM} Zapper Gun

Because the Zapper was conceived and developed in the 1980's, its operation utilizes very simple concepts. A "hit" or "miss" is determined by the following procedure, which occurs when the trigger is pulled:

1. The screen is made completely black for one frame.
2. The screen is kept completely black, except for the ducks, which are surrounded by a white box to distinguish them from the rest of the screen.
3. If the Zapper detected a completely dark frame, followed by a bright frame, it means that the gun was aimed at a duck, and therefore registers a 'hit'.
4. If the Zapper detected two completely dark frames, it means that the gun was not aimed at a duck, and registers a "miss".

If the Zapper did not detect one of the two above scenarios, then the Zapper was either not aimed at the screen or the user was attempting to cheat.

The above algorithm must also be extended for detection of multiple ducks. This is done using a binary search algorithm to minimize the number of screen flashes that occur. The algorithm is shown below:

1. Assuming there are n ducks on the screen, all n ducks are flashed.
2. If a hit was detected, then a binary search is performed as detailed below:
   (a) The ducks are split into two groups of $n/2$ ducks, named Group 1 and Group 2.
   (b) Group 1 is flashed. If there was a detect, then Group 2 is eliminated and we begin at Step 1 with Group 1 and $n/2$ ducks.
   (c) If there was not a detect, then we know the duck that was shot resides in Group 2. We begin again at Step 1 with Group 2 and $n/2$ ducks.

This algorithm above will complete in log2(n) frames where n is the number of ducks on the screen. Therefore, if the maximum 16 ducks are on the screen, the algorithm will take only 4 frames to complete.

The Zapper hardware is fairly simple. The detection circuit consists of a lens that focuses a light-detecting phototransistor and a band-pass filter. The trigger circuit is a switch connected to ground. The schematic of the circuit that was included with the original patent is shown in Figure 2.2.

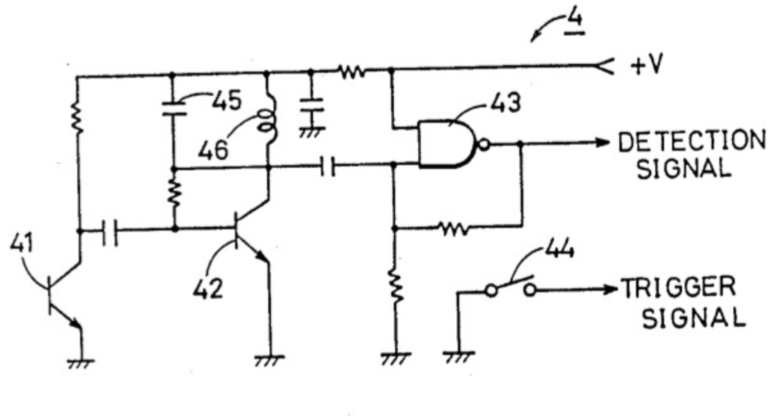

Figure 2.2: The schematic provided by the original Zapper patent

In the schematic, the capacitor and inductor labeled 45 and 46 comprise the band-pass filter. In more modern versions of the gun, this filter is replaced by an integrated circuit that performs the same task. The corner frequency of the filter is instead set by a resistor and capacitor. A schematic of the modern Zapper, which is the one that we used in our project, is presented in Figure 2.3. This schematic shows the integrated circuit that is used for the filter, as well as the open-collector outputs that we pulled up on the development board.

The filter bode plot included with the original patent is presented in Figure 2.4.

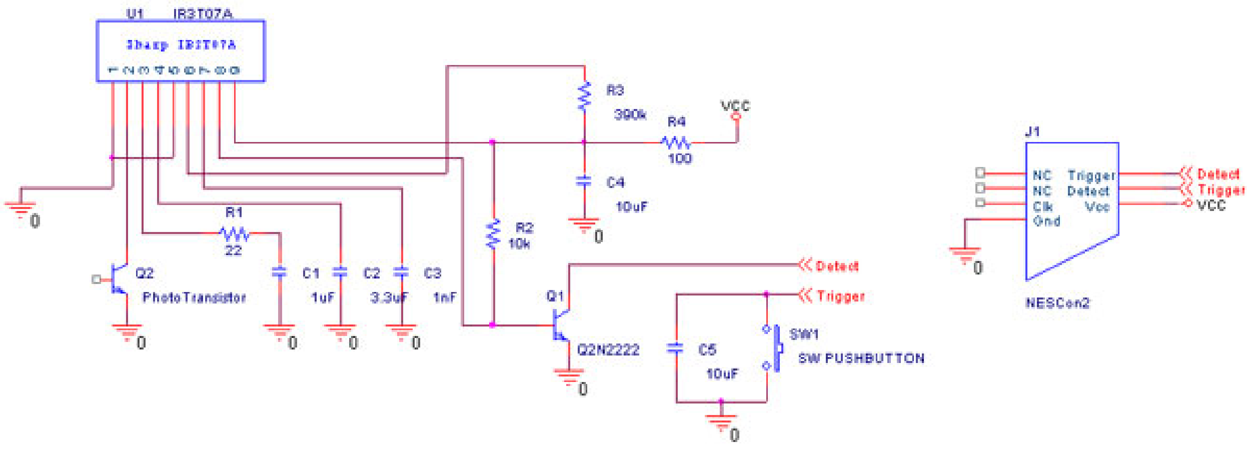

Figure 2.3: The schematic of the Modern Zapper used for this project

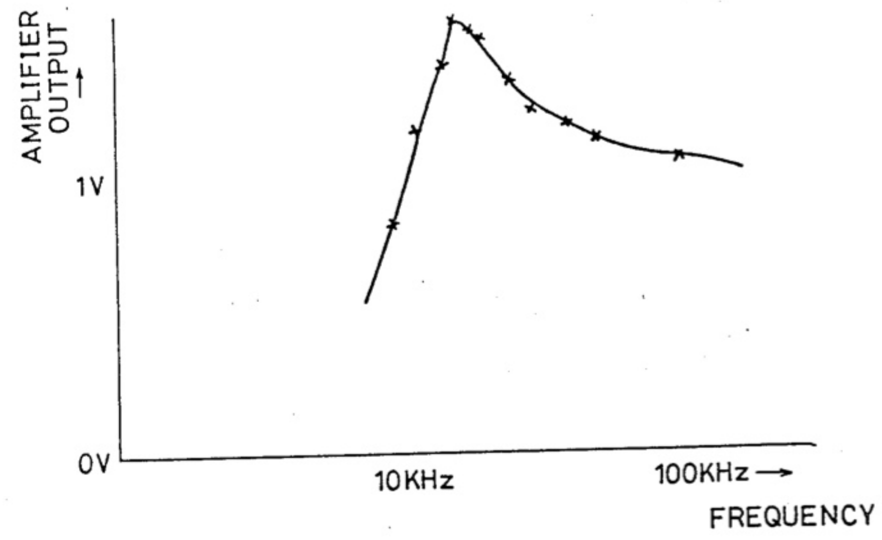

Figure 2.4: The filter Bode Plot provided by the original Zapper patent

The band-pass filter in Figure 2.4 has a corner frequency at around $15kHz$, which *was* the horizontal refresh rate for CRTs in the 1980s. Because of this limitation, it was found through experimentation that the Zapper fails to operate on a computer monitor CRT or LCD screen. Seeing as though the goal is for this game to operate on a computer monitor CRT, this filter must be altered to have a corner frequency near $31kHz$. This will likely be accomplished by altering the resistor value that sets the corner frequency by attaching a resistor in parallel with the one that is already in the gun. Unconfirmed sources state that adding a $390k\Omega$ resistor in parallel with the one that is already in the gun will accomplish this by creating an equivalent resistance of $180k\Omega$.

By altering the band-pass filter in the Zapper gun, it will operate with a CRT Computer Monitor. This allows us to use an existing monitor over VGA. VGA is chosen because we are familiar with the standard and may be able to reuse existing VHDL code.

2.3.2 Interfacing the Zapper

In order to interface the Zapper with our board, a custom hardware board was created to pull the signals from the zapper up to the 3.3V levels expected by our FPGA board. In addition, power was supplied to the Zapper. The schematic for our hardware is presented in Figure 2.5.

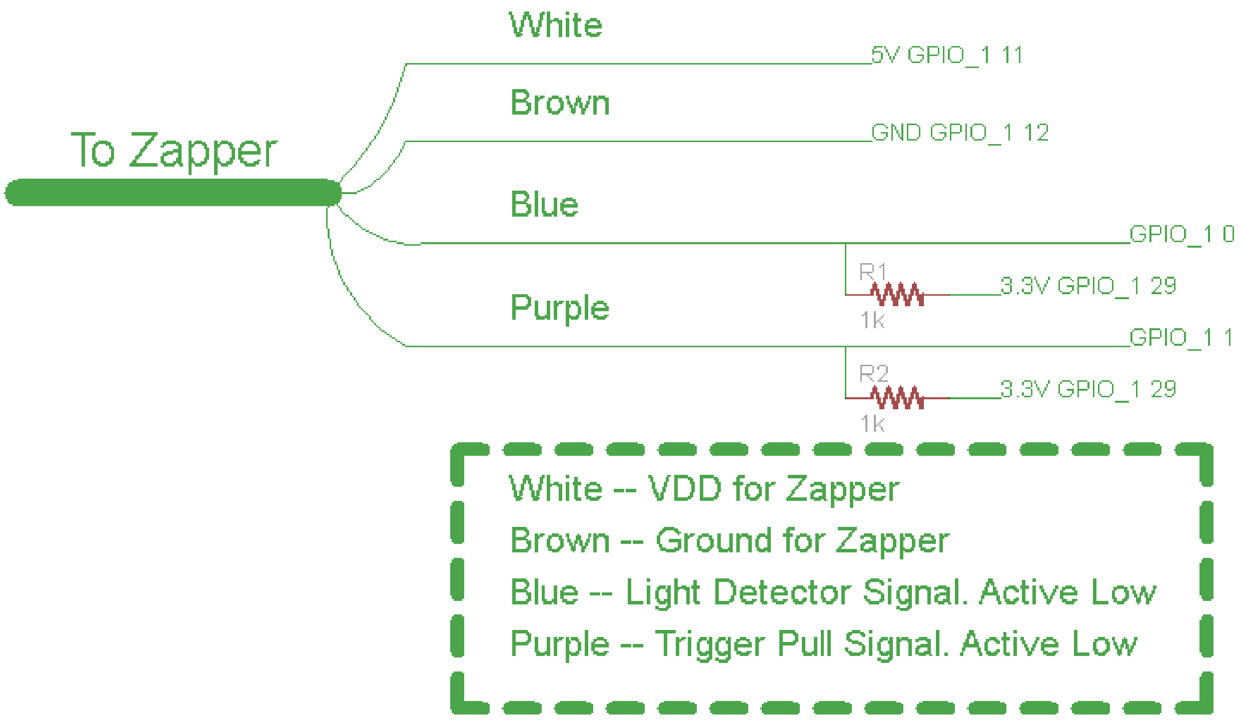

Figure 2.5: The schematic for the interfacing hardware between the Zapper and the FPGA board

2.3.3 The VGA Controller

The VGA Controller will be based off of the de2_vga_raster.vhd file provided with the files in Lab 3. The input ports needed for the vga_raster include reset, a clock which is originally fed in as 50 MHz but will be divided down to 25 MHz for the VGA protocol, vertical and horizontal blank signals that tell the vga whether or not to draw the default background and vertical and horizontal sync signals that tell the vga_raster to draw something besides the default background.

The VGA Controller will render ducks and other images using the notion of Sprites. Since the background is a static image, the VGA Controller will always draw the background in hardware without any communication from the software. The software merely tells the VGA controller where and when to draw Sprites. This will be done using some hardware registers that are written to by software. Among these registers (detailed in the next section) are registers for the X and Y position of any one of the 16 ducks, registers for displaying the correct score, and registers for on-screen ASCII messages.

The controller will also have output ports VGA_R, VGA_G, and VGA_B that tell the controller which color to display in a each location on the VGA. The color output signals will be set according to logic that is performed in different processes in the architecture of the controller VHDL file. The controller will have front_porch and back_porch constants that will keep the controller drawing on the visible screen.

2.3.4 The Zapper Controller

The Zapper controller will be a VHDL construct that interacts with the Zapper's outputs. Its goal is to receive trigger pull signals and target detect signals from the gun and relay these messages to the software stored in the NIOS processor. When the trigger on the Zapper is pulled, the Zapper controller must enact the screen blank and target highlight mechanisms. This is accomplished by communicating directly with the VGA controller. First, the controller determines that the gun was pointed at the screen by blanking it. Then, the controller must establish that a duck was hit by highlighting each visible duck sprite with a white box. If a duck was hit, the controller must establish which duck was hit using the binary search algorithm detailed in section 3.1.

2.4 Hardware/Software Interfaces

The hardware and software communicate through a set of registers that are defined in the VGA controller and Zapper Controller. These registers are detailed in this section.

2.4.1 VGA Controller Register Definitions

Register Name	Format	Description
Duck sprite n x-position	16-bit unsigned int	Contains the X position (in pixels) of duck n.
Duck sprite n y-position	16-bit unsigned int	Contains the Y position (in pixels) of duck n.
Duck sprite visibility	16-bit binary	A binary digit for each duck determining whether or not that duck is visible.
Message	64-byte char string	Message to be displayed across the screen in ASCII. Useful for messages like "Game over!".
Shots remaining	16-bit unsigned int	The number of shots remaining used to determine how many bullet (food?) graphics to display on the bottom-left.
Time remaining	16-bit unsigned int	The amount of time remaining in 'bars' used to determine how many time 'bars' to display on the bottom-center.
Score	16-bit unsigned int	Contains the score which will be displayed in numerals on the bottom-right.

Table 2.1: The Registers for the VGA Controller. All of the following registers are written to by software and read by hardware.

2.4.2 Zapper Controller Register Definitions

Register Name	Format	Description
Zapper detect	16-bit binary	Hardware sets the binary digit corresponding to the index of the duck hit when the hardware logic has determined that the user has shot a duck. When the user has not shot a duck, all binary bits will be zero. Software then reads this register to determine the game logic.
Zapper shots-fired	16-bit unsigned int	Hardware increments this register when the user has pulled the trigger. Software will read this register to determine game logic.
Zapper reset shot-count	16-bit binary	Software sets this register to a specific value when a new level begins. Hardware reads this and will then reset its shot-count.

Table 2.2: The Registers for the Zapper Controller. Note that there will also be internal registers, not read or written to by software used to implement the target hit algorithm

2.5 Software Design

Software is written in the *C* programming language and compiled and debugged using the NIOS II Integrated Development Environment. The required software functions are listed in Table 2.3. Please note that these functions are not an extensive list of all of the functions in the software. We may decide to split functions that grow too large into a number of “helper” functions.

Name	Description
`void main()`	Begins the game by printing an introductory message and continuously calls each level function. If the user fails at any point, it prints a game over message. If all levels are completed, it prints a game win message.
`int level(int numDucks, int numShots)`	Begins a level with a given number of ducks and allowed shots. This consists of a game loop which continuously updates the screen and polls the hardware registers to determine if a shot occurred.
`int shotsFired()`	Reads the Zapper Control register to determine how many shots have been fired.
`int duckHit()`	Reads the Zapper Control register to determine if a duck has been hit.
`void resetCount()`	Resets the hardware shot counter.
`void printMessage(char *message)`	Prints a text message across the screen, such as “Game Over” or “Go!”
`void printTime(int timeleft)`	Updates the countdown graphic at the bottom of the screen.
`void printScore(int score)`	Updates the score on the screen
`void printShots(int shotsleft)`	Updates the number of shots left
`void printDuck(int duckIndex, int visible, int x, int y)`	Updates a single duck sprite

Table 2.3: A list of the software functions and their descriptions

Chapter 3

Project Details

This section provides details on the project implementation. It details which team members were responsible for which aspects of the project, and gives the timeline and milestones for the project. In addition, lessons that we learned and advice for future students are also presented.

3.1 Milestones

This section describes the three milestones that we set up at the beginning of the project and details whether or not we actually met these milestones.

3.1.1 Milestone 1 - March 29

For the first milestone, we expected to have the Nintendo^{TM} Zapper completely figured out and interfaced with our hardware. This meant ordering the hardware, taking it apart, and figuring out what wires we would have to manipulate to get it to interact properly with our development board. Our goals were to have the necessary hardware in place that could determine whether the Zapper was being pointed at an area on the screen and whether the trigger on the Zapper was pulled or not.

We were able to meet our first Milestone right on. Our only hiccup was the fact that we had to solder a resistor into the printed circuit board in the Zapper, but after that was done, the hardware that we created (basically just pull-up resistors) worked as expected.

3.1.2 Milestone 2 - April 12

For our second milestone, the goal was to have graphics implemented. This entailed having a background painted using the VGA raster and having at least one sprite on the screen (a duck).

Unfortunately, we underestimated the work that it would take to adapt the simple VGA raster to a fully-fledged sprite machine. We were unable to fully meet our second milestone; things were being displayed on the screen but it was not exactly what we expected.

After we spent some time simulating and understanding the code in the VGA raster properly (see our lessons learned). We implemented a Run-length encoding scheme for the background (to save on memory) and got sprites working shortly thereafter.

Even though we were not able to meet Milestone 2 exactly, we got a jump start on writing the C code for the game. While part of our team worked on the hardware, the other part began doing work that would need to be done eventually anyway. This way we did not lose much in the way of efficiency.

3.1.3 Milestone 3 - April 28

Our third milestone, which we met, was to have a working game (with possible bugs) that would need tweaking. Our game had levels, a shot counter, and score implemented in software. The only thing that was really missing was ducks! We had ducks being represented by red squares on the screen that turned green when shot.

Luckily, the ducks were eventually implemented using sprite logic and animated by using two images (one for duck flap down, one for duck flap up).

3.2 Team Breakdown

All of the team members were actively involved in the high-level design process, as well as the software/hardware interface. Apart from these, the specific responsibilities of each team member are described in this section.

3.2.1 Daniel Teger - Electrical Engineering

For the earlier milestones, Daniel worked very closely with Julie to develop the hardware "black box" that was necessary to interface the Zapper with the FPGA board hardware. Later in the project, Daniel wrote VHDL for the VGA raster and ironed out timing issues with the screen flashing (for detecting ducks).

3.2.2 Julie Dinerman - Electrical Engineering

Julie worked mostly with hardware and VHDL code. She was the major force behind getting the duck sprites to display properly and animate themselves in hardware. She also wrote parts of the VGA raster and Zapper controller.

3.2.3 Kevin Ramkishun - Electrical Engineering

Kevin wrote Java programs for converting JPEG and BMP encoded images to the correct Run-Length format for the background, or pixel-by-pixel format for the sprites. He also worked on various parts of the hardware and helped debugged the microcontroller software.

3.2.4 Scott Rogowski - Computer Sciences

Scott wrote most of the software in C that runs the game logic as well as the VHDL code that interfaces the hardware with the software. In addition, he was a vital part of the design process and developed the hardware algorithm for figuring out which of the 8 ducks was shot (binary search).

3.3 Lessons Learned

3.3.1 VHDL Is Not Programming

The most important thing to understand when venturing into the world of hardware description languages after being taught programming for many years is to really grasp the fact that writing VHDL code is nothing at all like writing code in a programming language (ex. C or Java). VHDL is a *description* language, and must be treated as such. This means that the hardware which is to be described *must* already be well designed. This means that block diagrams, schematics, and timing diagrams should already be completed before any VHDL code is written at all!

Software Practices such as quickly making minor edits, recompiling and rerunning the program will never work with VHDL. Not only does VHDL take very long to compile (over 5 minutes with our final project), there is no debugger! The best way to debug VHDL code is to look at timing diagrams (see the next subsection) and the sythesized logic (RTL form is the easiest to grasp).

3.3.2 Timing and Simulation are Key!

There is no need to guess as to what is being synthesized by the tool: the best way to understand what is going on is to run test benches. This means writing specific test benches to understand the timing of the circuit and match it up against the timing diagrams and designs that have been previously created.

Our team struggled for a long time when getting the background Run-Length decoder to work, all because of simple off-by-one errors. We (falsely) thought that we could step through the logic in the code and figure out where the issue was. We were wrong! It was not until we fully understood the timing in our circuit through simulations that were were able to locate the off-by-one errors and get a stable raster with a nice background on the screen.

3.4 Advice for Future Projects

3.4.1 Stress the Design Stage

We cannot emphasize this enough: the design *must* be completed before VHDL code is written. This is why a project plan and design document are required deliverables! Really focus on the design – draw block diagrams, make timing charts, create schematics. Know exactly what hardware is going to be implemented before any VHDL code is written. This will make the writing of the VHDL code extremely straightforward.

If for some reason something is not working as expected, do not guess and check. Instead, check it against your design! Simulations are the best way to do this, so don't be afraid to write test benches. It may be tedious and boring, but it is the only real way to understand the synthesized logic.

3.4.2 Always Understand Your Code

It may be tempting to copy VHDL code off of the internet or from a professor's notes because it seems like it is what you need, but if you are going to use code from elsewhere, always understand exactly what it does. We were given a simple VGA raster in one of our labs and we utilized it in our project, but we had many issues until we really dug into the code and understood what every statement was doing. What were the details that the professor coded for you so that you could focus on the important concepts? These details matter.

Appendix A

Source Code Listing

This appendix lists all of the source code that was edited or created by hand by our team members. Code that was used from other sources (ex. from the professor's notes) or generated by software (ex. from the Quartus tools and SOPC builder) are not listed here. A complete tarball of the source files that can recreate this project are also submitted with this report.

A.1 Zapper Controller

```
-------------------------------------------------------------------------------
--
-- Simple Zapper Controller
--
-------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity de2_zapper_controller is
  port (
    reset : in std_logic;
    clk   : in std_logic;                    -- Should be 25.125 MHz
    read       : in  std_logic;
    write      : in  std_logic;
    chipselect : in  std_logic;
    address    : in  unsigned(4 downto 0);
    readdata   : out unsigned(15 downto 0);
    writedata  : in  unsigned(15 downto 0);

 --Pins
    GPIO_1: inout std_logic_vector(35 downto 0);

 --To the VGA
 flash_zapper : out std_logic;
 triggerpull_zapper : out std_logic
    );
```

```
end de2_zapper_controller;

architecture rtl of de2_zapper_controller is
--Translate GPIO into human readable signals
signal trigger : std_logic:='0';
signal ison : std_logic:='0';

--Three signals to keep track of the trigger state
signal triggerBeingHeld :std_logic :='0';
signal waitForTriggerBounce: integer :=0;
signal triggerAction: std_logic :='0'; --Once we finally decide to issue a trigger interrupt

--amount of cycles to wait before we're sure the switch was released.
--not a concrete number, but it eliminates the bounce problem and is not noticably long
constant SWITCHBOUNCETIME : integer := 200000;

--A running count of the shots
signal shots : unsigned(4 downto 0) := "00000";

signal clk25 : std_logic := '0';

begin

 process (clk) begin
if rising_edge(clk) then
clk25 <= not clk25;
end if;
end process;


HumanReadable : process (clk25) begin
if GPIO_1(0) = '1' then
ison <= '1';
else
ison <= '0';
end if;
if GPIO_1(1) = '0' then    --the trigger is a high signal until the trigger pulls it low
trigger <= '1';
else
trigger <= '0';
end if;
end process;

--the purpose of this process is to prevent the screen from staying blanked while the trigger is held d
--So, while the trigger is held, do nothing, upon release, wait for the switchbouncetime
ZapperInterface : process (clk25) begin


if rising_edge(clk25) then
--Reset shot count if needed
if chipselect = '1' and write = '1' then
if writedata(0) = '1' then
shots <= "00000";
end if;
```

```
--Store when the trigger is being held
elsif trigger='1' then
triggerBeingHeld <= '1';
--Capture the moment the trigger goes from 1 to 0
--start a timer to minimize bounce effects
elsif triggerBeingHeld='1' and trigger = '0' then
waitForTriggerBounce <= SWITCHBOUNCETIME;
triggerBeingHeld <= '0';
--Upon the completion of bounce time, start the trigger action
elsif waitForTriggerBounce = 1 then
triggerAction <= '1';
waitForTriggerBounce <= 0;
shots <= shots + 1;
--Immediately, upon the next clock cycle, the trigger is over
elsif triggerAction = '1' then
triggerAction <= '0';
--Decrement the timer
else
waitForTriggerBounce <= waitForTriggerBounce-1;
end if;
end if;
end process;

flash_zapper <= ison;
triggerpull_zapper <= triggerAction;

GPIO_1(35 downto 2) <= "0000000000000000000000000000000000";

readdata <= "0000000000" & shots;

end rtl;
```

A.2 VGA Raster

```
-------------------------------------------------------------------------------
--
-- VGA Raster Display using Run-Length Encoding and Sprites stored in a ROM
-- Originally by Professor Stephen Edwards
-- Heavily Edited by the Duck Feed Group
-------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity de2_vga_raster is

port (
reset : in std_logic;
clk   : in std_logic;                    -- Should be 25.125 MHz
read       : in  std_logic;
write      : in  std_logic;
chipselect : in  std_logic;
address    : in  unsigned(4 downto 0);
readdata   : out unsigned(15 downto 0);
writedata  : in  unsigned(15 downto 0);

VGA_CLK,                          -- Clock
VGA_HS,                            -- H_SYNC
VGA_VS,                            -- V_SYNC
VGA_BLANK,                        -- BLANK
VGA_SYNC : out std_logic;     -- SYNC
VGA_R,                             -- Red[9:0]
VGA_G,                             -- Green[9:0]
VGA_B : out std_logic_vector(9 downto 0); -- Blue[9:0]

flash_vga : in std_logic;
triggerpull_vga : in std_logic;

red_pixel_vga: in std_logic_vector(9 downto 0)
);

end de2_vga_raster;

architecture rtl of de2_vga_raster is

-- Video parameters

constant HTOTAL       : integer := 800;
constant HSYNC        : integer := 96;
constant HBACK_PORCH  : integer := 48;
constant HACTIVE      : integer := 640;
constant HFRONT_PORCH : integer := 16;

constant VTOTAL       : integer := 525;
constant VSYNC        : integer := 2;
constant VBACK_PORCH  : integer := 33;
constant VACTIVE      : integer := 480;
constant VFRONT_PORCH : integer := 10;

--*10 just for debugging
constant BLACKSCREENTIME : integer := 332500;       -- was 450000;   --this is the time from end of fie
constant SHOWTARGETTIME : integer := 450000;--450000;
constant RADIUS       : integer := 40;
constant SCOREHEIGHT  : integer := 40;
constant SCOREWIDTH : integer := 40;
--constant FishHEIGHT  : integer := 20;
--constant FishWIDTH : integer := 20;


-- Signals for the video controller
signal Hcount : unsigned(9 downto 0) := "0000000000";  -- Horizontal position (0-800)
signal Vcount : unsigned(9 downto 0) := "0000000000";  -- Vertical position (0-524)
signal EndOfLine, EndOfField : std_logic := '0';

signal vga_hblank, vga_hsync,
vga_vblank, vga_vsync : std_logic := '0';  -- Sync. signals
```

```
--Secondary clocks for VGA timing
signal clk25 : std_logic := '0';

--The duck position storage variables
type position is array(0 to 7) of integer;
signal hcenter : position := (others => 0);
signal vcenter : position := (others => 0);
signal old_hcenter : position := (others => 0);

signal hScore : position := (600,650,700,0,0,0,0,0);
signal vScore : position := (others => 430);

----type fishPosition is array(0 to 15) of integer;
----
----signal hFish : fishposition := (110,140,170,200,230,260,290,320,350,380,410,440,470,500,530,580);
----signal vFish : fishposition := (others => 420);



-- square at current pixel boolean
--type squaretype is array (0 to 7) of std_logic;
type squaretype is array (0 to 8) of std_logic;
--signal square: squaretype := "00000000";
signal square: squaretype := "000000000";
signal squareScore: squaretype := "000000000";
----type squareFishtype is array (0 to 15) of std_logic;
----signal squareFish: squareFishtype := "0000000000000000";
signal fishSquare : std_logic := '0';
signal fishAddress : integer := 0;
signal fishOpaque : std_logic := '0';

signal duckHit: unsigned (7 downto 0) := "00000000";
signal duckVisible: unsigned(7 downto 0) := "11111111";

signal score : integer;

signal gameReset : std_logic := '1';
signal levelReset : std_logic := '1';

--zapper signals
signal hold_2: std_logic := '0';
signal waitFor: integer := 0; --For 525*800
signal waitForBlackScreen: integer :=0; --For 525*800
signal waitForTarget: integer :=0;
signal startTargetFlash : std_logic :='0';
signal blackScreen : std_logic := '0';
signal showTarget : integer := 0;
signal blankOnNextScreen : std_logic :='0';
signal targetFindingComplete : std_logic :='0';


signal anyHit: std_logic := '0';
signal targetFours: std_logic := '0';
signal targetTwos: std_logic := '0';
signal targetOnes: std_logic := '0';
signal targetHit: integer := 0;

signal backgroundPixelsRemaining : integer := 1;
signal rCurrent : std_logic_vector (9 downto 0) := "1111111111";
signal gCurrent : std_logic_vector (9 downto 0) := "0000000000";
signal bCurrent : std_logic_vector (9 downto 0) := "1111111111";
signal opaque : std_logic := '1';


--type pixel_values is array (7 downto 0) of std_logic_vector (9 downto 0);
type pixel_values is array (8 downto 0) of std_logic_vector (9 downto 0);
signal rCurrentSprite : pixel_values := (others => "1111111111");
signal gCurrentSprite : pixel_values := (others => "1111111111");
signal bCurrentSprite : pixel_values := (others => "1111111111");
signal opaqueSprite : squaretype := (others => '1');

signal rCurrentScore : pixel_values := (others => "1111111111");
signal gCurrentScore : pixel_values := (others => "1111111111");
signal bCurrentScore : pixel_values := (others => "1111111111");
signal opaqueScore : squaretype := (others => '1');

----type fish_pixel_values is array (15 downto 0) of std_logic_vector (9 downto 0);
----signal rFish : fish_pixel_values := (others => "1111111111");
----signal gFish : fish_pixel_values := (others => "1111111111");
----signal bFish : fish_pixel_values := (others => "1111111111");
----signal opaqueFish : squareFishtype := (others => '1');

-- background and sprite display
signal sr_instructionAddress  : unsigned(15 downto 0) := "0000000000000000";
signal sr_spriteAddress0 : integer := 0;
signal sr_spriteAddress1 : integer := 0;
signal sr_spriteAddress2 : integer := 0;
signal sr_spriteAddress3 : integer := 0;
signal sr_spriteAddress4 : integer := 0;
signal sr_spriteAddress5 : integer := 0;
signal sr_spriteAddress6 : integer := 0;
signal sr_spriteAddress7 : integer := 0;

signal sr_addressNumber : integer := 0;

signal wing_position :std_logic := '1';
signal frame_count : integer := 0;
--signal sr_spriteAddress : position := (others => 0);
signal read_sr_red : unsigned(15 downto 0) := "0000000000000000";
signal currColor      : unsigned (3 downto 0) := "0000";
signal numElements    : integer := 0;
signal numInstructions : integer := 8;
signal nextInstruction : unsigned(15 downto 0) := "0000000000000000";
signal incrementedAddress : std_logic := '0';


type sprite_array is array (1599 downto 0) of std_logic_vector (9 downto 0);
```

```
signal index : position := (others => 0);
signal row : position := (others => 0);
signal col : position := (others => 0);

signal indexScore : position := (others => 0);
signal rowScore : position := (others => 0);
signal colScore : position := (others => 0);

----signal indexfish : fishposition := (others => 0);
----signal rowFish : fishposition := (others => 0);
----signal colFish : fishposition := (others => 0);

type curr_color_array is array (7 downto 0) of unsigned (3 downto 0);
type score_color_array is array (2 downto 0) of std_logic;
----type fish_color_array is array (15 downto 0) of std_logic

--type curr_color_array is array (8 downto 0) of unsigned (3 downto 0);
signal read_currSpriteColor : curr_color_array := (others => "0000");
signal read_currScoreColor : score_color_array := (others => '0');
----signal read_currFishColor : fish_color_array := (others => '0');
signal read_currFishColor : std_logic;
--signal fishIndex : integer := 0;

signal timer : integer := 0;
signal shotsRemaining : integer := 0;



begin


spriteStore: entity work.sprite_rom port map (
Clk => clk25,
spriteAddress0 => sr_spriteAddress0,
spriteAddress1 => sr_spriteAddress1,
spriteAddress2 => sr_spriteAddress2,
spriteAddress3 => sr_spriteAddress3,
spriteAddress4 => sr_spriteAddress4,
spriteAddress5 => sr_spriteAddress5,
spriteAddress6 => sr_spriteAddress6,
spriteAddress7 => sr_spriteAddress7,
up_wing => wing_position,

addressNumber0 => indexScore(0),
addressNumber1 => indexScore(1),
addressNumber2 => indexScore(2),

whichNumber => score,

duck0 => read_CurrSpriteColor(0),
duck1 => read_CurrSpriteColor(1),
duck2 => read_CurrSpriteColor(2),
duck3 => read_CurrSpriteColor(3),
duck4 => read_CurrSpriteColor(4),
duck5 => read_CurrSpriteColor(5),
duck6 => read_CurrSpriteColor(6),
duck7 => read_CurrSpriteColor(7),

number0Pixel => read_CurrScoreColor(0),
number1Pixel => read_CurrScoreColor(1),
number2Pixel => read_CurrScoreColor(2),

fishPixel => fishOpaque,
--fishPixel1 => read_CurrFishColor(1),
--fishPixel2 => read_CurrFishColor(2),
--fishPixel3 => read_CurrFishColor(3),
--fishPixel4 => read_CurrFishColor(4),
--fishPixel5 => read_CurrFishColor(5),
--fishPixel6 => read_CurrFishColor(6),
--fishPixel7 => read_CurrFishColor(7),
--fishPixel8 => read_CurrFishColor(8),
--fishPixel9 => read_CurrFishColor(9),
--fishPixel10 => read_CurrFishColor(10),
--fishPixel11 => read_CurrFishColor(11),
--fishPixel12 => read_CurrFishColor(12),
--fishPixel13 => read_CurrFishColor(13),
--fishPixel14 => read_CurrFishColor(14),
--fishPixel15 => read_CurrFishColor(15),
--
--
fishAddress => fishAddress
--fishAddress1 => Indexfish(1),
--fishAddress2 => Indexfish(2),
--fishAddress3 => Indexfish(3),
--fishAddress4 => Indexfish(4),
--fishAddress5 => Indexfish(5),
--fishAddress6 => Indexfish(6),
--fishAddress7 => Indexfish(7),
--fishAddress8 => Indexfish(8),
--fishAddress9 => Indexfish(9),
--fishAddress10 => Indexfish(10),
--fishAddress11 => Indexfish(11),
--fishAddress12 => Indexfish(12),
--fishAddress13 => Indexfish(13),
--fishAddress14 => Indexfish(14),
--fishAddress15 => Indexfish(15)
);

backgroundStore: entity work.background_rom port map (
Clk => clk25,
instructionAddress =>nextInstruction,
data => read_sr_red,
arrayLength => numElements
);


HalfClock : process (clk) begin
if rising_edge(clk) then
```

```
clk25 <= not clk25;
end if;
end process;


--When the processor writes to this peripheral
SoftwareInterface : process (clk) begin
if rising_edge(clk) then
if chipselect = '1' and write = '1' then
if writedata(15 downto 13) = "000" then --Update horizont
old_hcenter <= hcenter;
hcenter(to_integer(writedata(12 downto 10))) <= to_integer(writedata(9 downto 0));
elsif writedata(15 downto 13) = "001" then --Update verical
vcenter(to_integer(writedata(12 downto 10))) <= to_integer(writedata(9 downto 0));
elsif writedata(15 downto 13) = "010" then --update duck visibility
duckVisible <= writedata(7 downto 0);
elsif writedata(15 downto 13) = "011" then --reset level
if writedata (3 downto 0) = "0101" then
levelReset <= '1';
else
levelReset <= '0';
end if;
elsif writedata(15 downto 13) = "100" then -- update timer
timer <= to_integer(writedata(9 downto 0));
elsif writedata(15 downto 13) = "101" then -- update shot counter
shotsRemaining <= to_integer(writedata(5 downto 0));
elsif writedata(15 downto 13) = "110" then -- reset game
if writedata(3 downto 0) = "0101" then
gameReset <= '1';
else
gameReset <= '0';
end if;
end if;
end if;
end if;
end process;

--set a short timer to blank the screen
process (clk25) begin
if rising_edge(clk25) then

if triggerpull_vga='1' then
blankOnNextScreen <= '1';
waitForBlackScreen <= -1; --so it never hits the case where blackScreenTime =1 or =0 before we reach en
elsif blankOnNextScreen = '1' and EndOfField = '1' then --set up a mechanism so blacking the screen onl
blankOnNextScreen <= '0';
blackScreen <= '1';
waitForBlackScreen <= BLACKSCREENTIME;
elsif waitForBlackScreen = 1 then     --flicker variable "startTargetFlash" to enter the target flashin
startTargetFlash <= '1';
waitForBlackScreen <= 0;
elsif waitForBlackScreen = 0 then
blackScreen <= '0';
startTargetFlash <= '0';
```

```
elsif waitForBlackScreen > 1 then
waitForBlackScreen <= waitForBlackScreen-1;
end if;
end if;
end process;

--next stage, set a timer to show targets on a black screen
process (clk25) begin
if rising_edge(clk25) then
if gameReset = '1' then
score <= 0;
duckHit <= "00000000";
elsif levelReset ='1' then
duckHit <= "00000000";
showTarget <= 0;
anyHit <= '0';
--score <= score + 100;
--Reset everything upon start of target flash
elsif startTargetFlash = '1' then
showTarget <= 4;
waitForTarget <= SHOWTARGETTIME;
anyHit <= '0';
targetFours <= '0';
targetTwos <= '0';
targetOnes <= '0';
targetHit <= 0;
targetFindingComplete<='0';
--If we have waited through the target, but this is not the last target
--then decrement and reset the target time
elsif waitForTarget = 0 and showTarget>0 then
showTarget <= showTarget - 1;
waitForTarget <= SHOWTARGETTIME;
--If we have gone through all of the targets, and atleast one target has been hit, update which duck wa
elsif showTarget=0 and anyHit='1' and targetFindingComplete='0' then
targetHit <= to_integer(targetFours&targetTwos&targetOnes);
duckHit(to_integer(targetFours&targetTwos&targetOnes)) <= '1';
score <= score + 1;
targetFindingComplete<='1';
--Otherwise, decrement the timer and check for target hits
else
waitForTarget <= waitForTarget-1;

if flash_vga='1' then
if showTarget=4 then
anyHit <= '1';
elsif showTarget=3 then
targetFours <= '1';
elsif showTarget=2 then
targetTwos <= '1';
elsif showTarget=1 then
targetOnes <= '1';
end if;
end if;
end if;
end if;
end process;


-- Horizontal and vertical counters
HCounter : process (clk25) begin
if rising_edge(clk25) then
if reset = '1' then
Hcount <= (others => '0');
elsif EndOfLine = '1' then
Hcount <= (others => '0');
else
Hcount <= Hcount + 1;
end if;
end if;
end process HCounter;

EndOfLine <= '1' when Hcount = HTOTAL - 1 else '0';

VCounter: process (clk25) begin
if rising_edge(clk25) then
if reset = '1' then
Vcount <= (others => '0');
elsif EndOfLine = '1' then
if EndOfField = '1' then
Vcount <= (others => '0');
else
Vcount <= Vcount + 1;
end if;
end if;
end if;
end process VCounter;

EndOfField <= '1' when Vcount = VTOTAL - 1 else '0';



-- State machines to generate HSYNC, VSYNC, HBLANK, and VBLANK
HSyncGen : process (clk25) begin
if rising_edge(clk25) then
if reset = '1' or EndOfLine = '1' then
vga_hsync <= '1';
elsif Hcount = HSYNC - 1 then
vga_hsync <= '0';
end if;
end if;
end process HSyncGen;

HBlankGen : process (clk25) begin
if rising_edge(clk25) then
if reset = '1' then
vga_hblank <= '1';
elsif Hcount = HSYNC + HBACK_PORCH then
vga_hblank <= '0';
```

```
elsif Hcount = HSYNC + HBACK_PORCH + HACTIVE then
vga_hblank <= '1';
end if;
end if;
end process HBlankGen;

VSyncGen : process (clk25) begin
if rising_edge(clk25) then
if reset = '1' then
vga_vsync <= '1';
elsif EndOfLine ='1' then
if EndOfField = '1' then
vga_vsync <= '1';
elsif Vcount = VSYNC - 1 then
vga_vsync <= '0';
end if;
end if;
end if;
end process VSyncGen;

VBlankGen : process (clk25) begin
if rising_edge(clk25) then
if reset = '1' then
vga_vblank <= '1';
elsif EndOfLine = '1' then
if Vcount = VSYNC + VBACK_PORCH - 1 then
vga_vblank <= '0';
elsif Vcount = VSYNC + VBACK_PORCH + VACTIVE - 1 then
vga_vblank <= '1';
end if;
end if;
end if;
end process VBlankGen;

--For all of the ducks, are we on a visible or invisible pixel
SquareGen : process (clk) begin
if rising_edge(clk) then
for i in 7 downto 0 loop --squareOff(i) = '0' and
if (duckVisible(i)='1' and vcenter(i)<Vcount and Vcount<vcenter(i)+RADIUS and hcenter(i)<HCount and Hcou
square(i) <= '1';
else
square(i) <= '0';
end if;
end loop;
end if;
end process squareGen;

--For all of the score squares, are we on a visible or invisible pixel
ScoreGen : process (clk) begin
if rising_edge(clk) then
for i in 3 downto 0 loop
if (vScore(i)<Vcount and Vcount<vScore(i)+SCOREHEIGHT and hScore(i)<HCount and Hcount<hScore(i)+SCOREWI
squareScore(i) <= '1';
else
```

```
squareScore(i) <= '0';
end if;
end loop;
end if;
end process ScoreGen;

fishGen: process (clk) begin
if rising_edge(clk) then
if(420<vcount and vcount<440) then
if (200<hcount and hcount<220 and shotsRemaining>0) then
fishSquare <= '1';
fishAddress <= (to_integer(vcount) - 420)*20+to_integer(hcount)-200;
elsif (230<hcount and hcount<250 and shotsRemaining>1) then
fishSquare <= '1';
fishAddress <= (to_integer(vcount) - 420)*20+to_integer(hcount)-230;
elsif (260<hcount and hcount<280 and shotsRemaining>2) then
fishSquare <= '1';
fishAddress <= (to_integer(vcount) - 420)*20+to_integer(hcount)-260;
elsif (290<hcount and hcount<310 and shotsRemaining>3) then
fishSquare <= '1';
fishAddress <= (to_integer(vcount) - 420)*20+to_integer(hcount)-290;
elsif (320<hcount and hcount<340 and shotsRemaining>4) then
fishSquare <= '1';
fishAddress <= (to_integer(vcount) - 420)*20+to_integer(hcount)-320;
elsif (350<hcount and hcount<370 and shotsRemaining>5) then
fishSquare <= '1';
fishAddress <= (to_integer(vcount) - 420)*20+to_integer(hcount)-350;
elsif (380<hcount and hcount<400 and shotsRemaining>6) then
fishSquare <= '1';
fishAddress <= (to_integer(vcount) - 420)*20+to_integer(hcount)-380;
elsif (410<hcount and hcount<430 and shotsRemaining>7) then
fishSquare <= '1';
fishAddress <= (to_integer(vcount) - 420)*20+to_integer(hcount)-410;
else
fishSquare <= '0';
fishAddress <= 0;
end if;
elsif (460<vcount and vcount<480) then
if (200<hcount and hcount<220 and shotsRemaining>8) then
fishSquare <= '1';
fishAddress <= (to_integer(vcount) - 460)*20+to_integer(hcount)-200;
elsif (230<hcount and hcount<250 and shotsRemaining>9) then
fishSquare <= '1';
fishAddress <= (to_integer(vcount) - 460)*20+to_integer(hcount)-230;
elsif (260<hcount and hcount<280 and shotsRemaining>10) then
fishSquare <= '1';
fishAddress <= (to_integer(vcount) - 460)*20+to_integer(hcount)-260;
elsif (290<hcount and hcount<310 and shotsRemaining>11) then
fishSquare <= '1';
fishAddress <= (to_integer(vcount) - 460)*20+to_integer(hcount)-290;
elsif (320<hcount and hcount<340 and shotsRemaining>12) then
fishSquare <= '1';
fishAddress <= (to_integer(vcount) - 460)*20+to_integer(hcount)-320;
elsif (350<hcount and hcount<370 and shotsRemaining>13) then
```

```
fishSquare <= '1';
fishAddress <= (to_integer(vcount) - 460)*20+to_integer(hcount)-350;
elsif (380<hcount and hcount<400 and shotsRemaining>14) then
fishSquare <= '1';
fishAddress <= (to_integer(vcount) - 460)*20+to_integer(hcount)-380;
elsif (410<hcount and hcount<430 and shotsRemaining>15) then
fishSquare <= '1';
fishAddress <= (to_integer(vcount) - 460)*20+to_integer(hcount)-410;
else
fishSquare <= '0';
fishAddress <= 0;
end if;
else
fishSquare <= '0';
fishAddress <= 0;
end if;
end if;
end process fishGen;

--
WingFlap: process (EndofField) begin
if EndofField = '1' then
if frame_count < 9 then
frame_count <= frame_count + 1;
else
wing_position <= not wing_position;
frame_count <= 0;
end if;
end if;
--wing_position <= not wing_position when frame_count = 5 else wing_position;
end process wingFlap;


-- These should be the correct values for colors 0 - 9
-- The colors 10 - 15 will be added when we figure out the colors
-- needed for the ducks.
colorList: process (currColor, clk) begin
case currColor is
when "0000" => rCurrent <= x"41" & "11";
gCurrent <= x"40" & "11";
bCurrent <= x"2b" & "11";
opaque <= '0';
when "0001" => rCurrent <= x"30" & "11";
gCurrent <= x"73" & "11";
bCurrent <= x"11" & "11";
opaque <= '1';
when "0010" =>  rCurrent <= x"2b" & "11";
gCurrent <= x"cb" & "11";
bCurrent <= x"ff" & "11";
opaque <= '1';
when "0011" =>  rCurrent <= x"8b" & "11";
gCurrent <= x"d2" & "11";
bCurrent <= x"65" & "11";
opaque <= '1';
```

```
when "0100" =>  rCurrent <= x"51" & "11";
gCurrent <= x"ba" & "11";
bCurrent <= x"c6" & "11";
opaque <= '1';
when "0101" =>  rCurrent <= x"84" & "11";
gCurrent <= x"1a" & "11";
bCurrent <= x"05" & "11";
opaque <= '1';
when "0110" =>  rCurrent <= x"b4" & "11";
gCurrent <= x"f7" & "11";
bCurrent <= x"15" & "11";
opaque <= '1';
when "0111" =>  rCurrent <= x"5b" & "11";
gCurrent <= x"c0" & "11";
bCurrent <= x"9f" & "11";
opaque <= '1';
when "1000" =>  rCurrent <= x"ad" & "11";
gCurrent <= x"7d" & "11";
bCurrent <= x"00" & "11";
opaque <= '1';
when "1001" =>  rCurrent <= x"8c" & "11";
gCurrent <= x"d4" & "11";
bCurrent <= x"2d" & "11";
opaque <= '1';
when "1010" =>  rCurrent <= "0010111100";
gCurrent <= "0011010100";
bCurrent <= "0000101101";
opaque <= '1';
when "1011" =>  rCurrent <= "0011111111";
gCurrent <= "0010001111";
bCurrent <= "0000011111";
opaque <= '1';
when "1100" =>  rCurrent <= "0010001111";
gCurrent <= "0011111000";
bCurrent <= "0011111111";
opaque <= '1';
when "1101" =>  rCurrent <= "0011111111";
gCurrent <= "0000000111";
bCurrent <= "0011111000";
opaque <= '1';
when "1110" =>  rCurrent <= "0010011101";
gCurrent <= "0011100001";
bCurrent <= "0011111111";
opaque <= '1';
when "1111" =>  rCurrent <= "0011111111";
gCurrent <= "0011111100";
bCurrent <= "0011100011";
opaque <= '1';

when others =>rCurrent <= "1111111111";
gCurrent <= "1111111111";
bCurrent <= "1111111111";
opaque <= '1';
end case;
end process colorList;


spriteColorList: process (read_currSpriteColor, clk) begin
for i in 7 downto 0 loop
--for i in 8 downto 0 loop
case read_currSpriteColor(i) is
when "0000" => rCurrentSprite(i) <= x"00" & "11"; --1111
gCurrentSprite(i) <= x"09" & "11"; --black <-- orange
bCurrentSprite(i) <= x"02" & "11";
opaqueSprite(i) <= '1';
when "0001" => rCurrentSprite(i) <= x"ff" & "11";  --1110
gCurrentSprite(i) <= x"b4" & "11"; --orange <-- black
bCurrentSprite(i) <= x"60" & "11";
opaqueSprite(i) <= '1';
when "0010" =>  rCurrentSprite(i) <= x"fa" & "11"; --1101
gCurrentSprite(i) <= x"fe" & "11"; --white <-- green
bCurrentSprite(i) <= x"ec" & "11";
opaqueSprite(i) <= '1';
when "0011" =>  rCurrentSprite(i) <= x"5c" & "11";  --1101    --blue
gCurrentSprite(i) <= x"a4" & "11";
bCurrentSprite(i) <= x"f4" & "11";
opaqueSprite(i) <= '0';   --Blue represents background.
when "0100" =>  rCurrentSprite(i) <= x"11" & "11"; --1011
gCurrentSprite(i) <= x"56" & "11"; --green <-- white
bCurrentSprite(i) <= x"04" & "11";
opaqueSprite(i) <= '1';
when others =>rCurrentSprite(i) <= "1111111111";
gCurrentSprite(i) <= "1111111111";
bCurrentSprite(i) <= "1111111111";
opaqueSprite(i) <= '1';
end case;
end loop;
end process spriteColorList;


scoreColorList: process (read_currSpriteColor, clk) begin
for i in 2 downto 0 loop
case read_currScoreColor(i) is
when '0' =>
opaqueScore(i) <= '0';
when others =>
rCurrentScore(i) <= "1111111111";
gCurrentScore(i) <= "1111111111";
bCurrentScore(i) <= "1111111111";
opaqueScore(i) <= '1';
end case;
end loop;
end process scoreColorList;

----fishColorList: process (read_currFishColor, clk) begin
----for i in 15 downto 0 loop
------for i in 8 downto 0 loop
----case read_currFishColor(i) is
```

```
----when '0' => rFish(i) <= x"00" & "11"; --1111
----gFish(i) <= x"09" & "11"; --black <-- orange
----bFish(i) <= x"02" & "11";
----opaqueFish(i) <= '0';
----when others =>rFish(i) <= "1111111111";
----gFish(i) <= "1111111111";
----bFish(i) <= "1111111111";
----opaqueFish(i) <= '1';
----end case;
----end loop;
----end process fishColorList;


nextInstruction <= sr_instructionAddress+1;


CounterThing : process(clk25, hcount, vcount)
begin
if rising_edge(clk25) then

for i in 3 downto 0 loop
 col(i) <= to_integer(Vcount) - vcenter(i) ;
if wing_position ='1' then
row(i) <= to_integer(Hcount) - hcenter(i) + 3;
index(i) <=  col(i)*40 + row(i); --facing right
else
row(i) <= to_integer(Hcount) - hcenter(i) + 4;
index(i) <=  col(i)*40 - row(i); --facing left
end if;
end loop;
for i in 7 downto 4 loop
 col(i) <= to_integer(Vcount) - vcenter(i) ;
if wing_position ='0' then
row(i) <= to_integer(Hcount) - hcenter(i) + 3;
index(i) <=  col(i)*40 + row(i); --facing right
else
row(i) <= to_integer(Hcount) - hcenter(i) + 4;
index(i) <=  col(i)*40 - row(i); --facing left
end if;
end loop;

 sr_spriteAddress0 <= index(0);
 sr_spriteAddress1 <= index(1);
 sr_spriteAddress2 <= index(2);
 sr_spriteAddress3 <= index(3);
 sr_spriteAddress4 <= index(4);
 sr_spriteAddress5 <= index(5);
 sr_spriteAddress6 <= index(6);
 sr_spriteAddress7 <= index(7);


for i in 7 downto 0 loop
colScore(i) <= to_integer(Vcount) - vScore(i) ;
rowScore(i) <= to_integer(Hcount) - hScore(i) + 3;
```

```
indexscore(i) <=  colScore(i)*40 + rowScore(i);
end loop;
--for i in 15 downto 0 loop
--colFish(i) <= to_integer(Vcount) - vFish(i) ;
--rowFish(i) <= to_integer(Hcount) - hFish(i) + 3;
--indexFish(i) <=  colFish(i)*20 + rowFish(i);
--end loop;

end if;
end process;


-- Registered video signals going to the video DAC
VideoOut: process (clk25, reset) begin
--Asyncronous reset, pixels are black and all instructions are zero
if reset = '1' then
VGA_R <= "0000000000";
VGA_G <= "0000000000";
VGA_B <= "0000000000";
backgroundPixelsRemaining <= 0;
sr_instructionAddress <= "0000000000000000";
numInstructions <= numElements;

elsif rising_edge(clk25) then
--If there are no instructions left, reset the instruction counter
if numInstructions = 0 then
sr_instructionAddress <= "0000000000000000";
end if;

--If we are in the actual pixel part of the screen
if vga_hblank = '0' and vga_vblank = '0' then
--Handle the background run length encoding stuff.
--Backgroundpixelsremaining refers to remaining pixels in the current instruction
--numInstructios refers to the number of instructions left
--read_sr_red is the actual color coming out of the ROM
if backgroundPixelsRemaining = 2 then
sr_instructionAddress <= sr_instructionAddress + 1;
backgroundPixelsRemaining <= 1;
incrementedAddress <= '1';
elsif backgroundPixelsRemaining = 1 then
if incrementedAddress = '0' then
sr_instructionAddress <= sr_instructionAddress + 1;
end if;
incrementedAddress <= '0';
backgroundPixelsRemaining <= 0;
elsif backgroundPixelsRemaining = 0 then
currColor <= read_sr_red(15 downto 12);
backgroundPixelsRemaining <= to_integer(read_sr_red(11 downto 0))-1;
numInstructions <= numInstructions -1;
else
backgroundPixelsRemaining <= backgroundPixelsRemaining-1;
end if;

if blackScreen = '1' then
```

```
VGA_R <= "0000000000";
VGA_G <= "0000000000";
VGA_B <= "0000000000";
elsif showTarget=4 then
if (square(0)='1' and opaqueSprite(0)='1')
or (square(1)='1' and opaqueSprite(1)='1')
or (square(2)='1' and opaqueSprite(2)='1')
or (square(3)='1' and opaqueSprite(3)='1')
or (square(4)='1' and opaqueSprite(4)='1')
or (square(5)='1' and opaqueSprite(5)='1')
or (square(6)='1' and opaqueSprite(6)='1')
or (square(7)='1' and opaqueSprite(7)='1')
then
VGA_R <= "1111111111";
VGA_G <= "1111111111";
VGA_B <= "1111111111";
else
VGA_R <= "0000000000";
VGA_G <= "0000000000";
VGA_B <= "0000000000";
end if;
elsif showTarget=3 then
if (square(4)='1' and opaqueSprite(4)='1')
or (square(5)='1' and opaqueSprite(5)='1')
or (square(6)='1' and opaqueSprite(6)='1')
or (square(7)='1' and opaqueSprite(7)='1')
then
VGA_R <= "1111111111";
VGA_G <= "1111111111";
VGA_B <= "1111111111";
else
VGA_R <= "0000000000";
VGA_G <= "0000000000";
VGA_B <= "0000000000";
end if;
elsif showTarget=2 then
if (square(2)='1' and opaqueSprite(2)='1')
or (square(3)='1' and opaqueSprite(3)='1')
or (square(6)='1' and opaqueSprite(6)='1')
or (square(7)='1' and opaqueSprite(7)='1')
then
VGA_R <= "1111111111";
VGA_G <= "1111111111";
VGA_B <= "1111111111";
else
VGA_R <= "0000000000";
VGA_G <= "0000000000";
VGA_B <= "0000000000";
end if;
elsif showTarget=1 then
if (square(1)='1' and opaqueSprite(1)='1')
or (square(3)='1' and opaqueSprite(3)='1')
or (square(5)='1' and opaqueSprite(5)='1')
or (square(7)='1' and opaqueSprite(7)='1')
```

```
then
VGA_R <= "1111111111";
VGA_G <= "1111111111";
VGA_B <= "1111111111";
else
VGA_R <= "0000000000";
VGA_G <= "0000000000";
VGA_B <= "0000000000";
end if;
elsif showTarget = 0 then
if square(0)= '1' and opaqueSprite(0) ='1' then
VGA_R <= rCurrentSprite(0);
VGA_G <= gCurrentSprite(0);
VGA_B <= bCurrentSprite(0);
elsif square(1)='1' and opaqueSprite(1) ='1' then
VGA_R <= rCurrentSprite(1);
VGA_G <= gCurrentSprite(1);
VGA_B <= bCurrentSprite(1);
elsif square(2)='1' and opaqueSprite(2) ='1' then
VGA_R <= rCurrentSprite(2);
VGA_G <= gCurrentSprite(2);
VGA_B <= bCurrentSprite(2);
elsif square(3)='1' and opaqueSprite(3) ='1' then
VGA_R <= rCurrentSprite(3);
VGA_G <= gCurrentSprite(3);
VGA_B <= bCurrentSprite(3);
elsif square(4)='1' and opaqueSprite(4) ='1' then
VGA_R <= rCurrentSprite(4);
VGA_G <= gCurrentSprite(4);
VGA_B <= bCurrentSprite(4);
elsif square(5)='1' and opaqueSprite(5) ='1' then
VGA_R <= rCurrentSprite(5);
VGA_G <= gCurrentSprite(5);
VGA_B <= bCurrentSprite(5);
elsif square(6)='1'  and opaqueSprite(6) ='1' then
VGA_R <= rCurrentSprite(6);
VGA_G <= gCurrentSprite(6);
VGA_B <= bCurrentSprite(6);
elsif square(7)='1' and opaqueSprite(7) ='1' then
VGA_R <= rCurrentSprite(7);
VGA_G <= gCurrentSprite(7);
VGA_B <= bCurrentSprite(7);
elsif squareScore(0) = '1' and opaqueScore(0) = '1' then
VGA_R <= rCurrentScore(0);
VGA_G <= gCurrentScore(0);
VGA_B <= bCurrentScore(0);
elsif squareScore(1) = '1' and opaqueScore(1) = '1' then
VGA_R <= rCurrentScore(1);
VGA_G <= gCurrentScore(1);
VGA_B <= bCurrentScore(1);
elsif squareScore(2) = '1' and opaqueScore(2) = '1' then
VGA_R <= rCurrentScore(2);
VGA_G <= gCurrentScore(2);
VGA_B <= bCurrentScore(2);
```

```
elsif fishOpaque='1' and fishSquare = '1' then
VGA_R <= "0000000000";
VGA_G <= "0000000000";
VGA_B <= "1111111111";

--elsif squareFish(0)='1' and opaqueFish(0) = '1'then
--VGA_R <= rFish(0);
--VGA_G <= gFish(0);
--VGA_B <= bfish(0);
--elsif squareFish(1)='1' and opaqueFish(1) = '1'then
--VGA_R <= rFish(1);
--VGA_G <= gFish(1);
--VGA_B <= bfish(1);
--elsif squareFish(2)='1'and opaqueFish(2) = '1' then
--VGA_R <= rFish(2);
--VGA_G <= gFish(2);
--VGA_B <= bfish(2);
--elsif squareFish(3)='1' and opaqueFish(3) = '1'then
--VGA_R <= rFish(3);
--VGA_G <= gFish(3);
--VGA_B <= bfish(3);
--elsif squareFish(4)='1' and opaqueFish(4) = '1'then
--VGA_R <= rFish(4);
--VGA_G <= gFish(4);
--VGA_B <= bfish(4);
--elsif squareFish(5)='1' and opaqueFish(5) = '1'then
--VGA_R <= rFish(5);
--VGA_G <= gFish(5);
--VGA_B <= bfish(5);
--elsif squareFish(6)='1' and opaqueFish(6) = '1'then
--VGA_R <= rFish(6);
--VGA_G <= gFish(6);
--VGA_B <= bfish(6);
--elsif squareFish(7)='1' and opaqueFish(7) = '1'then
--VGA_R <= rFish(7);
--VGA_G <= gFish(7);
--VGA_B <= bfish(7);
--elsif squareFish(8)='1' and opaqueFish(8) = '1'then
--VGA_R <= rFish(8);
--VGA_G <= gFish(8);
--VGA_B <= bfish(8);
--elsif squareFish(9)='1' and opaqueFish(9) = '1'then
--VGA_R <= rFish(9);
--VGA_G <= gFish(9);
--VGA_B <= bfish(9);
--elsif squareFish(10)='1' and opaqueFish(10) = '1'then
--VGA_R <= rFish(10);
--VGA_G <= gFish(10);
--VGA_B <= bfish(10);
--elsif squareFish(11)='1' and opaqueFish(11) = '1'then
--VGA_R <= rFish(11);
--VGA_G <= gFish(11);
--VGA_B <= bfish(11);
--elsif squareFish(12)='1' and opaqueFish(12) = '1'then
```

```
--VGA_R <= rFish(12);
--VGA_G <= gFish(12);
--VGA_B <= bfish(12);
--elsif squareFish(13)='1' and opaqueFish(13) = '1'then
--VGA_R <= rFish(13);
--VGA_G <= gFish(13);
--VGA_B <= bfish(13);
--elsif squareFish(14)='1' and opaqueFish(14) = '1'then
--VGA_R <= rFish(14);
--VGA_G <= gFish(14);
--VGA_B <= bfish(14);
--elsif squareFish(15)='1' and opaqueFish(15) = '1'then
--VGA_R <= rFish(15);
--VGA_G <= gFish(15);
--VGA_B <= bfish(15);

elsif vCount > VSYNC + VBACK_PORCH + VACTIVE - 12 and hCount < HSYNC + HBACK_PORCH + timer then
VGA_R <= "1111111111";
VGA_G <= "1111111111";
VGA_B <= "1111111111";
else
--otherwise, just print the background
VGA_R <= rCurrent;
VGA_G <= gCurrent;
VGA_B <= bCurrent;
end if;
end if; --blackscreen
elsif EndOfField = '1' then
--This completely resets upon a new screen
backgroundPixelsRemaining <= 0;
sr_instructionAddress <= "0000000000000000";
VGA_R <= "0000000000";
VGA_G <= "0000000000";
VGA_B <= "0000000000";
else
VGA_R <= "0000000000";
VGA_G <= "0000000000";
VGA_B <= "0000000000";
end if; -- vgablankl end if
end if; -- reset and clock end if
end process VideoOut;


VGA_CLK <= clk25;
VGA_HS <= not vga_hsync;
VGA_VS <= not vga_vsync;
VGA_SYNC <= '0';
VGA_BLANK <= not (vga_hsync or vga_vsync);

readData <= "00000000" & duckHit;

end rtl;
```

A.3 Software

```
#include <io.h>
#include <system.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <time.h>

#define RADIUS 60

#define HWIDTH 640
#define HSYNC 96
#define HBACK_PORCH 48

#define VWIDTH 480
#define VSYNC 2
#define VBACK_PORCH 33
#define GROUND_HEIGHT 110

#define LEFT_MAX HSYNC+HBACK_PORCH
#define RIGHT_MAX HSYNC+HBACK_PORCH+HWIDTH-RADIUS
#define TOP_MAX VSYNC+VBACK_PORCH
#define BOTTOM_MAX VSYNC+VBACK_PORCH+VWIDTH-RADIUS-GROUND_HEIGHT


typedef struct {
    int timeLimit;
    int numDucks;
    int numShots;
    int speed;
    } LevelType;

//The level array
LevelType levels[10] =
    {{30,2,14, 3000},
     {30,2,14, 5000},
     {30,4,14, 4000},
     {30,4,8, 3000},
     {20,4,8, 2500},
     {15,6,10, 2000},
     {15,8,12, 1500},
     {15,8,10, 1500},
     {15,8,8, 1000},
     {8,8,8, 1000}};

int playLevel(int level) {
    //Standard
    unsigned int i;


    ////SET HARDWARE VARIABLES////

    //Reset game immediately
    //Yes, 4 times is essential for these writes.  I hate NIOS.
    IOWR_16DIRECT(VGA_BASE, 0, 0x6005);
    IOWR_16DIRECT(VGA_BASE, 0, 0x6005);
    IOWR_16DIRECT(VGA_BASE, 0, 0x6005);
    IOWR_16DIRECT(VGA_BASE, 0, 0x6005);

    //A short delay
    usleep(500000);

    //Reset shot count
    IOWR_16DIRECT(ZAPPER_BASE, 0, 0x0001);
    IOWR_16DIRECT(ZAPPER_BASE, 0, 0x0001);
    IOWR_16DIRECT(ZAPPER_BASE, 0, 0x0001);
    IOWR_16DIRECT(ZAPPER_BASE, 0, 0x0001);

    usleep(500000);
    //Set reset game to 0
    IOWR_16DIRECT(VGA_BASE, 0, 0x6000);
    IOWR_16DIRECT(VGA_BASE, 0, 0x6000);
    IOWR_16DIRECT(VGA_BASE, 0, 0x6000);
    IOWR_16DIRECT(VGA_BASE, 0, 0x6000);

    usleep(500000);

    IOWR_16DIRECT(ZAPPER_BASE, 0, 0);
    IOWR_16DIRECT(ZAPPER_BASE, 0, 0);
    IOWR_16DIRECT(ZAPPER_BASE, 0, 0);
    IOWR_16DIRECT(ZAPPER_BASE, 0, 0);

    usleep(500000);


    ////INITIALIZE SOFTWARE VARIABLES////

    //Initialize the countdown and shot counter
    double count = 0;
    int shots = 0;

    //Has any given duck been hit?
    //Only holds 1 or 0 values
    int duckHit[8];
    int hitCount = 0;

    //Horizontal and vertical positions and directions
    unsigned int h[8], v[8];
    int hdir[8], vdir[8];

    //Is any given duck visible? (depends on initial level)
    //Once invisible, does not change back
    volatile int duckVisible = 0x4000;

    //Initially, all of the level's ducks are visible
for (i=0; i<levels[level].numDucks; i++) {
duckVisible |= 1<<i;
```

```
}

    //Initialize some default values
    for (i=0; i<8; i++) {
        v[i]=BOTTOM_MAX;    //Ducks will start at the ground
        vdir[i] = 1;        //They will all start by flying up
        h[i]=rand()%400+LEFT_MAX;   //Their horizontal position will be random
        hdir[i]=rand()%2;   //Their horizontal direction will be random
        duckHit[i]=0;       //No ducks are hit so far
        }




    //Initialize the game with duckVisibility
    IOWR_16DIRECT(VGA_BASE, 0, duckVisible);
    IOWR_16DIRECT(VGA_BASE, 0, duckVisible);
    IOWR_16DIRECT(VGA_BASE, 0, duckVisible);
    IOWR_16DIRECT(VGA_BASE, 0, duckVisible);



////PLAY THE LEVEL WHILE////
    //1. the level has not been won
    //2. and the countdown > 0
    //3. and there are shots left
    int win=0;
while (win==0 && count < levels[level].timeLimit && shots < levels[level].numShots) {

        //Increment count
        // determined this value by experiment
        count+= (double)levels[level].speed/(double)300000;

        //For each duck
for (i=0; i<8; i++){
//If the duck has been registered as hit, make it fly off the screen
if (duckHit[i]==1) {
if (v[i]>TOP_MAX)
         v[i]-=4;
         //Once the duck hits the top of the screen, make it invisible
     else if (v[i]==TOP_MAX || v[i]==TOP_MAX-1 || v[i]==TOP_MAX-2 || v[i]==TOP_MAX-3) {
     int tmpMask = 0xffff;
         duckVisible &= tmpMask^(1<<i);
        }
     }
//Otherwise, the duck has not been hit, make it fly randomly
else {
     //Horizontal movement
     if (hdir[i] == 0) {
         if (h[i]>RIGHT_MAX || rand()%250==0)
             hdir[i] = 1;
         else
             h[i]++;
         }
     else {
         if (h[i]<LEFT_MAX || rand()%250==0)
             hdir[i] = 0;
         else
            h[i]--;
         }

                //Vertical movement
                if (vdir[i] == 0) {
                    if (v[i]>BOTTOM_MAX || rand()%250==0)
                        vdir[i] = 1;
                    else
                        v[i]++;
                    }
                else {
                    if (v[i]<TOP_MAX || rand()%250==0)
                        vdir[i] = 0;
                    else
                        v[i]--;
                    }
                }//else, duck not hit

            //Update horizontal position
            IOWR_16DIRECT(VGA_BASE, 0, i*0x400 | h[i]);

            //Update vertical position
            //the &0x2000 puts a 1 in the 13 spot to update the vertical
            IOWR_16DIRECT(VGA_BASE, 0, i*0x400 | 0x2000 | v[i]);
            }//for each duck

        //Update duck visibility
        IOWR_16DIRECT(VGA_BASE, 0, duckVisible);
        IOWR_16DIRECT(VGA_BASE, 0, duckVisible);
        IOWR_16DIRECT(VGA_BASE, 0, duckVisible);
        IOWR_16DIRECT(VGA_BASE, 0, duckVisible);

        //Check for duck hits
        unsigned int duckHitsHardware = IORD_16DIRECT(VGA_BASE, 0);
        for (i=0; i<8; i++) {
            if ((duckHitsHardware&0x1)==1) {
                if (duckHit[i] == 0) {
                    duckHit[i]=1;
                    hitCount++;
                    }
                }
            else
                duckHit[i]=0;
            duckHitsHardware=duckHitsHardware>>1;
            }

        //Update the number of shots
        shots=IORD_16DIRECT(ZAPPER_BASE, 0);
        printf("shots: %d\n",shots);
```

```
  //Update the shots and timer on the vga
  IOWR_16DIRECT(VGA_BASE,0,0xa000|(unsigned int)(levels[level].numShots-shots));
  IOWR_16DIRECT(VGA_BASE,0,0x8000|(unsigned int)((levels[level].timeLimit-count)*HWIDTH/levels[level].ti

        //Win if all ducks are hit.
        if (hitCount == levels[level].numDucks)
            win = 1;

        //Sleep to slow down the movement
        usleep(levels[level].speed);

}//While level not won or lost

    //Be sure that the rest of the ducks fly off the level before advancing
    if (win) {
        while (duckVisible!=0x4000) {
            //For each duck
            for (i=0; i<8; i++){
                if (v[i]>TOP_MAX)
                    v[i]-=3;
                //Once the duck hits the top of the screen, make it invisible
                else if (v[i]==TOP_MAX || v[i]==TOP_MAX-1 || v[i]==TOP_MAX-2) {
                    int tmpMask = 0xffff;
                    duckVisible &= tmpMask^(1<<i);
                    }
                IOWR_16DIRECT(VGA_BASE, 0, i*0x400 | 0x2000 | v[i]);
                }
            IOWR_16DIRECT(VGA_BASE, 0, duckVisible);
            usleep(levels[level].speed);
            }
        }

    return win;
    }



int main() {
    //Standard
    int i;

    //Initialize random
srand(time(NULL));

    //Play the game over and over again
    while (1) {
        //Assume that we won so that we can play the first level
        int win = 1;

        //Reset the game
        IOWR_16DIRECT(VGA_BASE, 0, 0xc005);
        IOWR_16DIRECT(VGA_BASE, 0, 0xc005);
        IOWR_16DIRECT(VGA_BASE, 0, 0xc005);
        IOWR_16DIRECT(VGA_BASE, 0, 0xc005);

        usleep(10000);

        IOWR_16DIRECT(VGA_BASE, 0, 0xc000);
        IOWR_16DIRECT(VGA_BASE, 0, 0xc000);
        IOWR_16DIRECT(VGA_BASE, 0, 0xc000);
        IOWR_16DIRECT(VGA_BASE, 0, 0xc000);



      //Go through all 10 levels
     for (i=0; i<10 && win==1; i++) {
      win=playLevel(i);
     }
        }


   return 0;
   }
```

A.4 Java Image Pixel and RLE Generator

```
import java.awt.Graphics2D;
import java.util.Set;
import java.util.List;
import java.util.ArrayList;
import java.util.HashSet;

import java.awt.Image;
import java.awt.RenderingHints;
import java.awt.Transparency;
import java.awt.image.BufferedImage;
import java.awt.image.PixelGrabber;
import java.awt.Color;
import java.io.File;

import javax.imageio.ImageIO;


/**
 * Generates a static array in VHDL that contains all of the pixel values
 * for the image given. Resizes the image to 40x40 if necessary.
 *
 * @author kr
 */
public class VHDLGen {

    public VHDLGen() { }

    static int WIDTH = 40;
    static int HEIGHT = 40;
    static int EXPECTED_NUM_COLORS = 10;

    /**
     * Grabs the image Pixels.
     * @param image image to use (should be size 40x40)
     */
    public static void processImage(Image image) {

        try {

            PixelGrabber grabber =
                    new PixelGrabber(image, 0, 0, WIDTH, HEIGHT, false);

            if (grabber.grabPixels()) {
            int width = grabber.getWidth();
            int height = grabber.getHeight();

                int[] data = (int[]) grabber.getPixels();

                Set<Color> colors = new HashSet<Color>();
                Color lastColor;

                lastColor = null;

                int curCount = 1;

                List<Integer> counts = new ArrayList<Integer>();
                List<Color> colorList = new ArrayList<Color>();
                boolean firstIteration = true;
                List<Color> pixels = new ArrayList<Color>();

                for (int y = 0; y < height; y++) {
                 for (int x = 0; x < width; x++) {
                 int pixel = data[y * width + x];
                 int red = (pixel >> 16) & 0xff;
                 int green = (pixel >> 8) & 0xff;
                 int blue = pixel & 0xff;
                 boolean addNow = false;

                 // For Printing Individual Pixels
//                 System.out.println(red + "\n" + green + "\n" + blue);

                 Color curColor = new Color(red, green, blue);
                 colors.add(curColor);
                 pixels.add(curColor);

//                 if (firstIteration) {
//                 firstIteration = false;
//                 lastColor = curColor;
//                 }
//
//                 if (curColor.equals(lastColor)) {
//
//                 if (++curCount == 4000) {
```

```
//                  addNow = true;
//                  }
//
//                  } else {
//                  addNow = true;
//                  }
//
//                  if (addNow) {
//                  colorList.add(lastColor);
//                  counts.add(curCount);
//                  curCount = 1;
//                  }
//
//                  lastColor = curColor;

                 }
                }

                // ensure that we have the right # of colors
//                if (colors.size() != EXPECTED_NUM_COLORS)
//                 throw new RuntimeException("Number of colors (" + colors.size() + ") is " +
//                 "not expected(" + EXPECTED_NUM_COLORS + ")");

                // remove all with count 1, this works fine
//                for (int i = 1; i < colorList.size(); i++) {
//                 if (counts.get(i) == 1) {
//
//                 int prevCount = counts.get(i - 1);
//
//                 colorList.remove(i);
//                 counts.remove(i - 1);
//                 counts.remove(i - 1);
//                 counts.add(i - 1, prevCount + 1);
//                 i--;
//                 }
//                }

//                if (colorList.size() != counts.size())
//                 throw new RuntimeException("ColorList Size = " + colorList.size() +
//                 "but Counts Size = " + counts.size());
//
//                // Convert all colors to instructions
//                int i = 0;
//                System.out.print("(");
//                for (Color c : colorList) {
//                 System.out.print("\"");
//
//                 System.out.print(Pad4(Integer.toBinaryString((transformColor(c, colors)))));
//                 System.out.print(Pad12(Integer.toBinaryString(counts.get(i))));
//
//
//                 System.out.print("\",");
//                 i++;
//                }
//                System.out.print(");");


                // Process Color image
//                System.out.println("Width: " + width + ", Height:" + height);
//                System.out.println("Distinct Colors: " + colors.size());
//                System.out.println("Num of Instructions: " + colorList.size());
//                System.out.println("Lists Agree: " + (colorList.size() == counts.size()));
//                System.out.println("Instructions: ");

                // print color lookup table
//                for (Color c : colors) {
//                 System.out.println("RGB:" + (Integer.toHexString(c.getRed()))
//                 + ", " + (Integer.toHexString(c.getGreen()))
//                 + ", " + (Integer.toHexString(c.getBlue())));
//                }

                System.out.print("(");
                for (int i = 0; i < pixels.size(); i++) {
                 Short s = new Short(transformColor(pixels.get(i), colors));
                 System.out.print ("\"" + Pad4(Integer.toBinaryString(s.intValue())) + "\",");
                }

                System.out.println(")");

                System.out.println("\nColors:");
                for (Color c : colors) {
                 System.out.print(c + ",");
                }
                System.out.println();

//                for (i = 0; i < colorList.size(); i++)
//                 System.out.println(colorList.get(i) + ", " + counts.get(i));

            } else {
             System.err.println("Could not grab pixels");
            }
        }
        catch (InterruptedException e1) {
            e1.printStackTrace();
        }
    }

    static String Pad4(String s) { return Pad(s, 4); }
    static String Pad12(String s) { return Pad(s, 12); }

    static String Pad(String s, int num) {

     // this should NEVER happen
     if (s.length() > num)
     throw new RuntimeException("The length of the string is " + s.length() +
     " but we are trying to pad to " + num);

     while (s.length() < num)
```

```
 s = "0" + s;
 return s;
}

static String Pad10(String s) {
 while (s.length() != 10)
 s = "0" + s;
 return s;
}

/**
 * Translates a color into an index into the lookup table.
 *
 * @param c the color
 * @param allColors the lookup table
 * @return the index
 */
static short transformColor(Color c, Set<Color> allColors)
{
 boolean found = false;
 short counter = 0;
 for (Color col : allColors) {
 if (!c.equals(col)) {
 counter++;
 } else {
 found = true;
 break;
 }
 }

 // should NEVER happen
 if (!found)
 throw new RuntimeException("Color not found in lookup table!");

 return counter;
}

/**
 * Convenience method that returns a scaled instance of the
 * provided {@code BufferedImage}.
 *
 * @param img the original image to be scaled
 * @param targetWidth the desired width of the scaled instance,
 *    in pixels
 * @param targetHeight the desired height of the scaled instance,
 *    in pixels
 * @param hint one of the rendering hints that corresponds to
 *    {@code RenderingHints.KEY_INTERPOLATION} (e.g.
 *    {@code RenderingHints.VALUE_INTERPOLATION_NEAREST_NEIGHBOR},
 *    {@code RenderingHints.VALUE_INTERPOLATION_BILINEAR},
 *    {@code RenderingHints.VALUE_INTERPOLATION_BICUBIC})
 * @param higherQuality if true, this method will use a multi-step
 *    scaling technique that provides higher quality than the usual
 *    one-step technique (only useful in downscaling cases, where
 *    {@code targetWidth} or {@code targetHeight} is
 *    smaller than the original dimensions, and generally only when
 *    the {@code BILINEAR} hint is specified)
 * @return a scaled version of the original {@code BufferedImage}
 */
public static BufferedImage getScaledInstance(BufferedImage img,
                                       int targetWidth,
                                       int targetHeight,
                                       Object hint,
                                       boolean higherQuality)
{
    int type = (img.getTransparency() == Transparency.OPAQUE) ?
        BufferedImage.TYPE_INT_RGB : BufferedImage.TYPE_INT_ARGB;
    BufferedImage ret = (BufferedImage)img;
    int w, h;
    if (higherQuality) {
        // Use multi-step technique: start with original size, then
        // scale down in multiple passes with drawImage()
        // until the target size is reached
        w = img.getWidth();
        h = img.getHeight();
    } else {
        // Use one-step technique: scale directly from original
        // size to target size with a single drawImage() call
        w = targetWidth;
        h = targetHeight;
    }

    do {
        if (higherQuality && w > targetWidth) {
            w /= 2;
            if (w < targetWidth) {
                w = targetWidth;
            }
        }

        if (higherQuality && h > targetHeight) {
            h /= 2;
            if (h < targetHeight) {
                h = targetHeight;
            }
        }

        BufferedImage tmp = new BufferedImage(w, h, type);
        Graphics2D g2 = tmp.createGraphics();
        g2.setRenderingHint(RenderingHints.KEY_INTERPOLATION, hint);
        g2.drawImage(ret, 0, 0, w, h, null);
        g2.dispose();

        ret = tmp;
    } while (w != targetWidth || h != targetHeight);

    return ret;
}
```

```
    /**
     * Command line main function accepts two arguments (input file and output
     * file for 40x40 version).
     *
     * @param args Arguments to main
     */
    public static void main(String args[]) {

     if (args.length <= 1) {
     System.err.println("usage: java VHDLGen <infile.jpg> <40x40file.jpg>");
     return;
     }

     try {
     BufferedImage img = ImageIO.read(new File(args[0]));
     BufferedImage resized = getScaledInstance(img, WIDTH, HEIGHT,
     RenderingHints.VALUE_INTERPOLATION_NEAREST_NEIGHBOR,
     false);

     ImageIO.write(resized, "bmp", new File(args[1]));

     processImage(resized);
         System.exit(0);

     } catch (Exception ex) {
     ex.printStackTrace();
     }
    }
}
```

A.5 Java Pixel Generator Tester

```
import java.awt.image.BufferedImage;
import java.io.BufferedReader;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

import javax.imageio.ImageIO;

public class InstructionTest
{

private static List<String> instructions = new ArrayList<String>();
```

```
static List<Integer> r = new ArrayList<Integer>();
static List<Integer> g = new ArrayList<Integer>();
static List<Integer> b = new ArrayList<Integer>();

private static int WIDTH = 40;
private static int HEIGHT = 40;

/**
 * @param args
 */
public static void main(String[] args)
{
if (args.length != 1) {
System.err.println("Please supply file to read");
return;
}

String fileName = args[0];

// Read file into array
try {

BufferedReader f = new BufferedReader(
new InputStreamReader(new FileInputStream(fileName)));




String line;
while ((line = f.readLine()) != null) {
instructions.add(line);

// BY PIXELS TEST - passes
// r.add(Integer.parseInt(line));
// line = f.readLine();
// g.add(Integer.parseInt(line));
// line = f.readLine();
// b.add(Integer.parseInt(line));

}

} catch (FileNotFoundException e) {
e.printStackTrace();
} catch (IOException e) {
e.printStackTrace();
}

// Print Instructions
// printInstructions();
createImage();
}

public static String Pad2(String s) {
```

```
if (s.length() == 1)
return "0" + s;
else
return s;
}

public static void createImage()
{
BufferedImage recreation = new BufferedImage(WIDTH, HEIGHT,
 BufferedImage.TYPE_INT_RGB);

int x = 0;
int y = 0;

// by pixels test - PASSES, PIXELS ARE OK, RLE IS NOT.
// Set <String> uniques = new HashSet<String>();
// for (int i = 0; i < r.size(); i++) {
//
// int red = r.get(i);
// int green = g.get(i);
// int blue = b.get(i);
//
// String color = Pad2(Integer.toHexString(red)) + Pad2(Integer.toHexString(green)) +
//    Pad2(Integer.toHexString(blue));
//
// uniques.add(color);
//
// System.out.println("RGB: " + color + ", " + red + ", " + green + ", " + blue);
// recreation.setRGB(x, y, Integer.parseInt(color, 16));
// if (++x == WIDTH) {
// x = 0;
// y++;
// }
// }
// System.out.println("Unique Colors: " + uniques.size());

// go through each instruction
for (String s : instructions) {

int color = Integer.parseInt(s.substring(0, 4), 2);
int length = Integer.parseInt(s.substring(4), 2);

System.out.println("Color: " + color + ", Length: " + length);

for (int i = 0; i < length; i++) {

recreation.setRGB(x, y, getRGBFromCode(color));

if (++x == WIDTH) {
x = 0;
y++;
}
}
}
```

```
System.out.println("At the end, X = " + x + ", Y = " + y);
System.out.println("Number of Instructions: " + instructions.size());

try {
ImageIO.write(recreation, "jpg", new File("recreation.jpg"));
} catch (IOException e) {
e.printStackTrace();
}

}

/**
 * This is populated from the lookup table generated by VHDLGen
 *
 * @param code  Color code
 * @return RGB Hex value
 */
static int getRGBFromCode(int code)
{
switch(code) {
case 0:
return 0x41402b;
case 1:
return 0x307311;
case 2:
return 0x2bcbff;
case 3:
return 0x8bd265;
case 4:
return 0x51bac6;
case 5:
return 0x841a05;
case 6:
return 0xb4f715;
case 7:
return 0x5bc09f;
case 8:
return 0xad7d00;
case 9:
return 0x8cd42d;
default:
return 0x00000000;

}
}

public static void printInstructions()
{
for (String s : instructions) {
System.out.println(s);
}
}
```

```
}
```