Optical Input Targeting Game

Daniel Southern
Rachel Bainbridge

Abstract

Two of the more interesting applications possible with the 6.111 Labkit are video processing and 2D game systems. Our project integrates both of these ideas to create an interactive game experience which uses a video camera to generate most of the game input. Our design is composed of a 2-part system: a 2D graphics engine with video game logic and a video processing module to generate control signals for the video game.

The graphics system design will use sprites to draw both an arbitrary set of sprites to the screen along with the system's interpretation of the cursor position. Through this architecture, we will duck-hunt clone complete with the original sprites. The video processing module will be responsible for generating a set of coordinates for the cursor based on the position of a dot of red laser light in the camera image. This module will serve as the light gun from the original duck-hunt game, the object of the game will be to position the cursor over the flying duck, and then pull the trigger to shoot the duck.

Table of Contents

1	**Abstract**	**ii**
2	**List of Figures**	**iv**
3	**Overview**	**1**
4	**Description**	**2**
5	**Conclusions**	**14**
6	**Appendices**	**16**

2) List of Figures

Figure 1.	**System Overview**	**1**
Figure 2.	**Video Output and Game Logic Block Diagram**	**3**
Figure 3.	**Animated Living Duck in 4-bit Color**	**4**
Figure 4.	**Reading Pixel Data from a ROM**	**5**
Figure 5.	**Video Processing Block Diagram**	**8**
Figure 6.	**Frame Buffer Access Order**	**9**
Figure 7.	**Frame Buffer Reader Wiring Specification**	**10**
Figure 8.	**Frame Buffer Writer Wiring Specification**	**11**
Figure 9.	**Frame Buffer Data Configuration**	**11**
Figure 10.	**Frame Buffer Wiring Specification**	**12**
Figure 11.	**Pixel Analyzer Wiring Specification**	**13**
Figure 12.	**YCrCb To HSV Converter Wiring Specification**	**14**

3) Overview

The most important rule of thumb for organizing a technical description is ``Describe the whole before the parts''. This rule is based on the assumption that the device which you are describing is unknown to the reader and that a general view of the purpose and construction is needed before the details can be understood.
With this rule in mind, start your 6.111 report with an overview of the purpose, use and design of the device, what a user does with it and how he or she would do it. Describe in general the subsystem organization of the device. Emphasize those internal features which implement the main user-visible features.

Our Duck Hunt game will function similarly to the Duck Hunt of the original NES. A laser aimed at a computer screen will function as the "gun" and a camera will be used to locate this laser and a red dot will be displayed by the game at its location. The player will also be able to fire the gun by pressing the enter button on the lab kit. The computer screen will display a gaming environment similar to that of the original Duck Hunt. Animated ducks will fly across the screen one at a time and the score will also be displayed in the upper left hand corner of the screen.

Our implementation of the Duck Hunt game consists of two major parts: the video output to the screen which will be handled by Rachel Bainbridge and the input from the camera that locates the laser pointer which will be handled by Dan Southern.

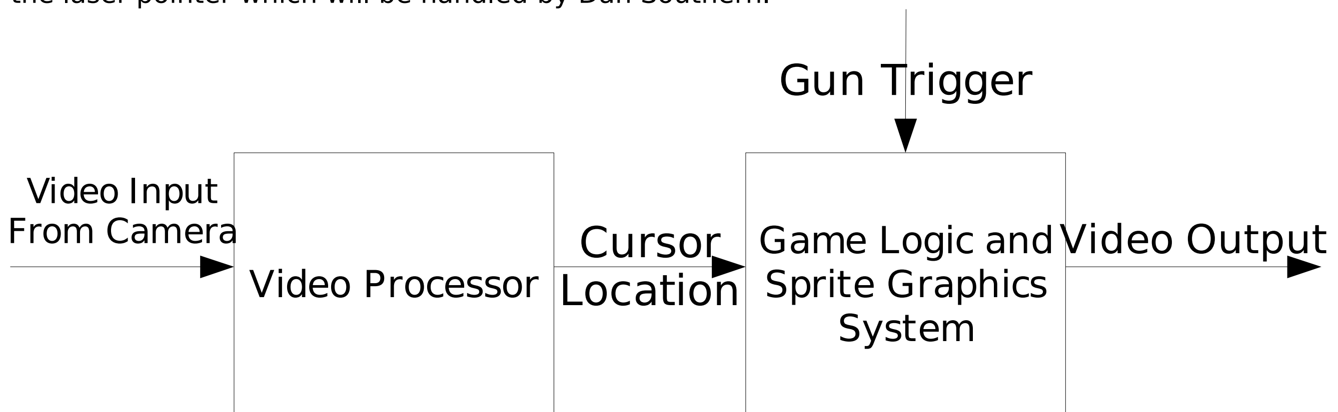

Figure 1 – System Overview

3.1) Game Logic and Sprite Graphics System (Rachel Bainbridge)

The video output for the duck hunt game is sprite-based system done in four bit color. The game logic is in charge of handling duck movement, score count, shot count, and checks if a duck has been hit by the player when the gun is fired. For the ducks and for the background tree and bush sprites, the original duck hunt sprites were used, with some minor color alterations. Two more sprites were specially made for the score and shot counters (as the original game sprites were not available), that are similar in appearance to the originals and function similarly as well. The duck will fly around the screen and flap its wings, and change the way it faces when it changes x direction. The player can hit a button to fire their laser pointer gun, and if they are within the bounds of the duck sprite, they will kill the duck and it will twirl gracefully to the ground, and they will receive a point for that duck. However, the player is limited to three shots per duck, as indicated by the shots display, and if they do not hit it by the time it has bounced on the top of the screen it will fly away and they will not receive a point for that duck. The game logic consists

of one module that runs on the pixel clock (65MHz) and miraculously needed no pipelining. The sprites also all run on the pixel clock and rely on the same basic code. My original goals were to have working game logic, background and animated duck sprites, and some kind of score keeping mechanism. I actually got implement an interesting score sprite and a shot sprite that kept track of the user's ammo.

3.2) Video Processing (Dan Southern)

One of the main goals of our project is to create a system which can interpret where a user is pointing with a laser pointer on a user interface. This ability has a variety of meaningful applications. We thought that one that worked well in the context of a two-part 6.111 project was to utilize the cursor information as an input to a game to replace the cursor. By aiming the laser pointer at a control surface, the user is moving the cursor to a desired position much in the same way we move the pointer using the mouse on a computer.

The basic idea of the system is that a video camera will capture the image of the control surface (the surface at which we are pointing the laser pointer) and store it in a frame buffer on the Labkit. This data will be analyzed in order to determine where in the image the laser pointer dot is located. This information is translated into a set of coordinates which are then passed on to an arbitrary system that can make use of this cursor data.

Analysis of the video data occurs so rapidly that to a user of the system, the response will feel instant, and a user watching the position of the cursor on the output display should have no problem controlling the cursor as this control is very similar to the visual feedback involved in moving the pointer on a computer.

4) Description

4. Game Logic and Sprite Graphics System

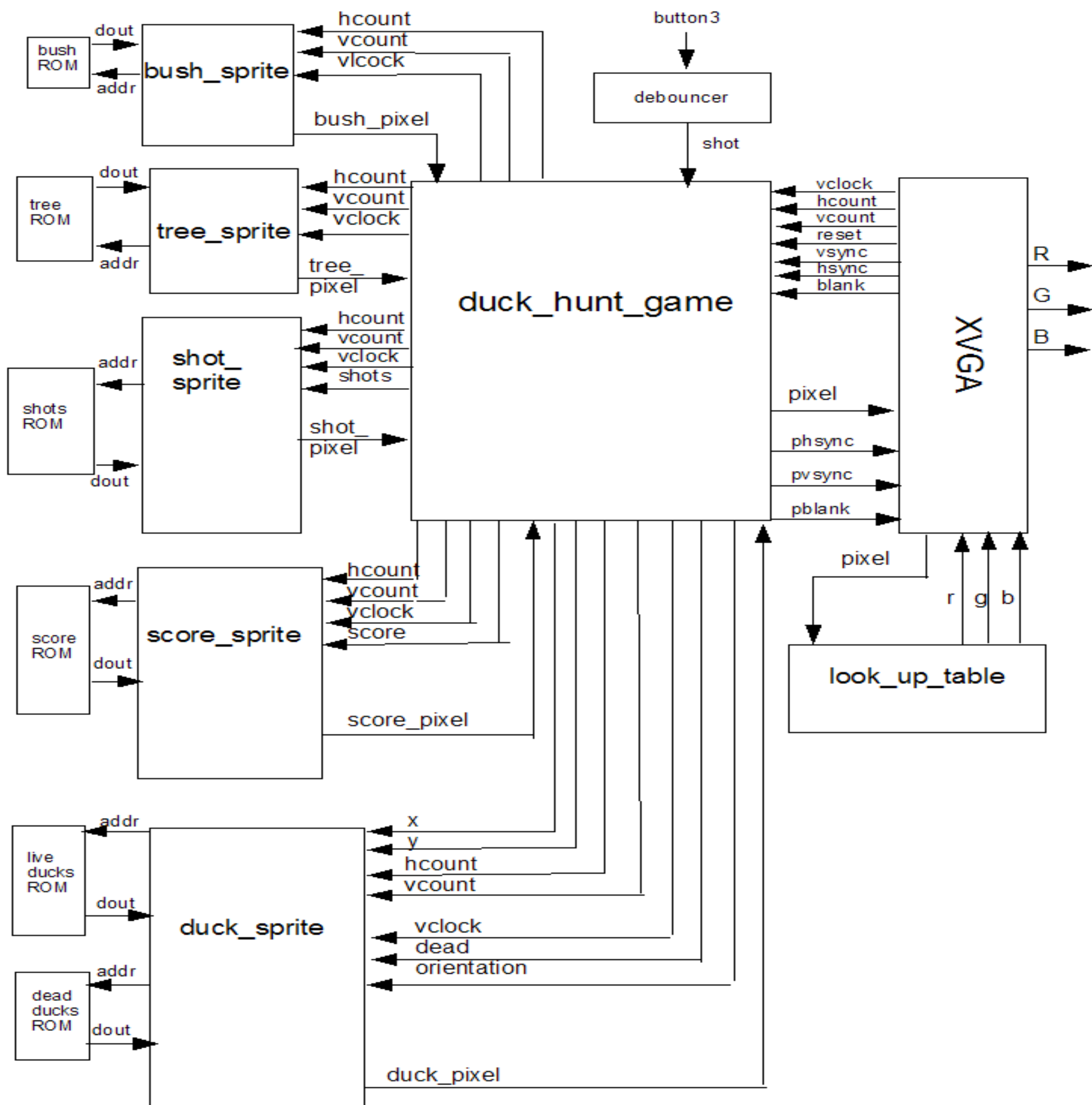

Figure 2 – Video and Game Logic Block Diagram

4.1.1) Sprite Module (Rachel Bainbridge)

To being making a sprite ROM in four bit color, one must first start with a sprite image (almost any format that paint can open will do). Then image can be opened in an image processing program (paint worked best for me, even better than Photoshop which was causing my sprites to come out weird colors mostly likely due to strange color mapping) and then save the image as a 16 color bitmap image (there are several options ranging between monochrome and 24-bit true color for all kinds of sprites you could want to do). Paint will then tell say some color image data may be lost if you save in this format. This warning is fine, as the shape of sprite will be preserved but some of the colors may change, because they were not available in the 4-bit color palette. One can then change the colors as one wants using the available colors and save the image. Then using MATLAB code borrowed from older projects, the bitmap can be converted into a .coe file to be used as the

initial loading file for the BRAM. When one creates the BRAM it is important to remember to load the file and to make sure the read only option is selected (unless you plan on doing later alterations to the BRAM in with the circuit.) The width of the ROM should be the number of bits per pixel (four in this case) and the depth should be the total number of pixels in the image (width of the sprite multiplied by the height). The address of the information for the sprite will then go from zero to the depth and output a value with the number of bits specified by width. For animated sprites, one can create one ROM with multiple sprites on it and simply cycle through them by making sure they are all the same size and then adding the depth of one sprite to the starting address when one wants to change the current sprite.

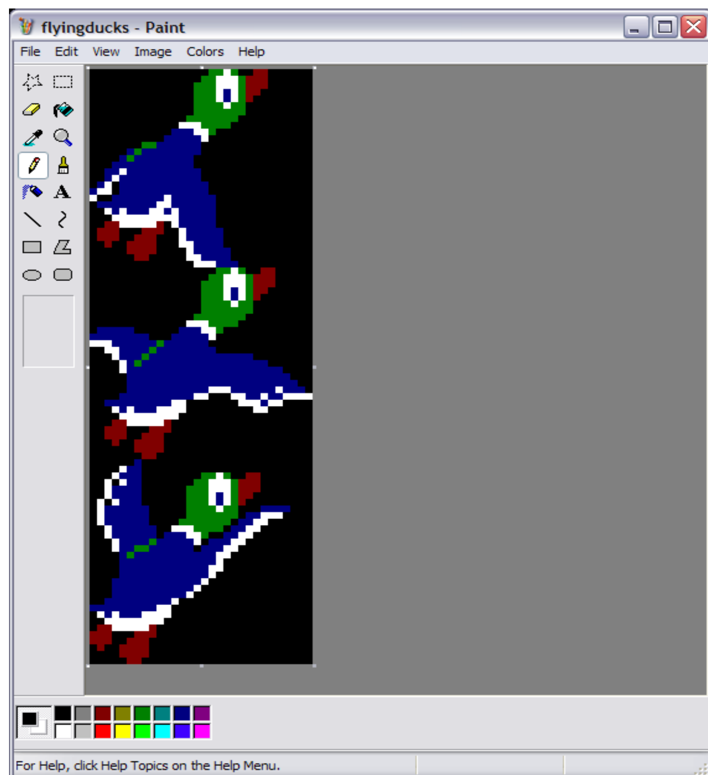

Figure 3 – Animated Living Duck Sprite in 4-bit Color

4.1.2) Lookup Table Module (Rachel Bainbridge)

The color palette used in this project was 4-bit color and so a look up table was needed. 4-bit color is not true color, that is the bits do not correspond directly to RGB values, but rather is indexed color, in which a certain value corresponds to a predetermined color. The look up table module takes an input of a 4-bit pixel value and outputs an 8-bit for each red, green and blue value to be used by the xvga module to create the output the screen. The pixel value is simply stuck into a giant case statement that behaves like combinational logic. Depending on the value of pixel one of sixteen colors will be output.

4.1.3) Stationary Sprite Module (Rachel Bainbridge)

There are two sprites in our Duck Hunt implementation which are stationary background sprites. These are the bush and tree sprites. Each has its own BRAM which stores it pixel data and parameters specifying its height, width, and location. The inputs to the module are hcount, vcount and the pixel clock, and the output is the pixel data the sprite has for the current location. If hcount and vcount are within the sprite, that is if hcount is between the sprite's beginning x

location and end location, which is simply the beginning x location added to the sprites width comma, and the same is true of vcount, except with the y locations, then the pixel output is assigned to the output from the ROM and the address is incremented by one. If the current hcount and vcount are outside the sprite, then address is assigned zero - it is basically reset- and the pixel output is also given a zero (transparent). Implementing more stationary sprites is as easy as adding more location parameters and modifying the if statement to include the new locations.

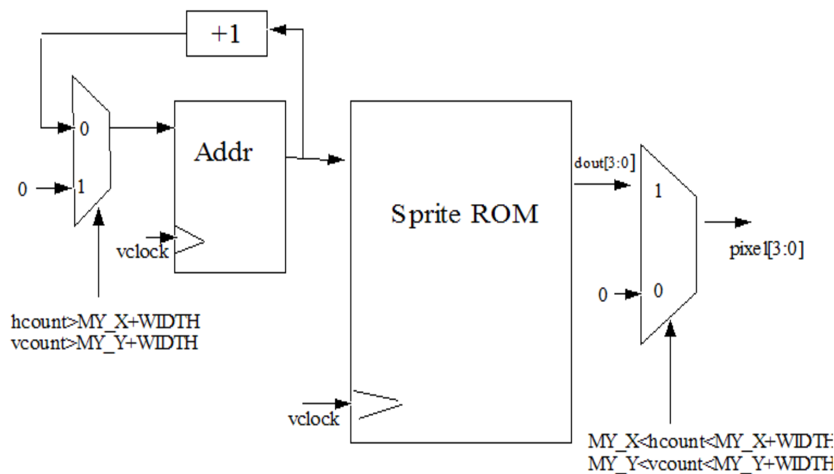

Figure 4 - Reading Pixel Data from a ROM

4.1.4) Divider Module (Rachel Bainbridge)

The Divider module is used to time the flapping of the duck. It increments a counter once each clock cycle until it reaches one fourth of 65 million and then it sets an enbl signal high and resets the counter to zero. The module basically counts out one fourth of a second, which is an easy time to count and a speed at which the duck flapping looks smooth.

4.1.5) Duck Sprite Module (Rachel Bainbridge)

The Duck Sprite Module is much more complex than it's stationary counter parts, because the sprite itself must move while the whole thing moves around the screen. The module takes as inputs the x and y of its current location, hcount and vcount, the pixel clock, its x orientation and the dead signal, and outputs its pixel data for the current x y coordinate. The module has parameters specifying the width and height The duck sprite module has two BRAMs, one for dead ducks and one for live ducks. There are three versions of the live duck, to create a duck that appears to be flapping and two versions for the dead duck, to create a duck that appears to be spinning to the ground. The sprites are laid out vertically in the ROM one after another, and are all identical in size. The basic reading from the ROM is the same for both dead and live ducks, except that the dead duck code has been modified so that dduck_no has only two values. For both versions duck_no has the starting address in the ROM for the current duck that is being displayed on the screen. If hcount and vcount are between and the current x, y of the duck and x added to the width and y added to the height, then the pixel output is given dout and the address is

increased, otherwise it is given zero. Every time the enable signal from the divider module goes high, the duck_no is increased by 900 (or wraps around back to zero if it is at the highest possible starting address) in order to get to the next duck in the ROM (in this case ducks are 30 pixels by 30 pixels, for a total of 900 locations needed for each duck). Switching ducks creates, the flapping/twirling motion. For the flying ducks, the same ROM was used for ducks flying both left and right. In order to switch the orientation of the duck, one must read from the end of the row first. In order to do this two variables, line and pix were used. Pix begins at zero and increments up to the width of the sprite and is then reset to zero while line is incremented. Line starts at one less than the width of the sprite and increments by the width of the sprite every time pix reaches the width of the sprite. The address to be read from the ROM is then assigned to duck_no plus line minus pix, which will effectively read a mirror image of what is in the ROM. If hcount and vcount are outside the sprite, then line is reset to one less than the width and pix is reset to zero.

4.1.6) Score and Shots Sprite Module (Rachel Bainbridge)

The Score and Shots modules are stationary sprite modules that take extra inputs and function quite similarly. The score module takes a four-bit input score in addition to vcount, hcount, vclock, and outputs a pixel for the score display and shots takes the additional two-bit input shots. The Score and Sprite modules are both hand-made sprites that have shapes that represent either bullets or ducks. The shots sprite has three circles representing the bullets that the player is using and the score sprite has four duck shapes that represent the ducks the user is shooting at. When a duck is hit, it is white, when it is missed it is black. When a shot is used, it is black, the remaining bullets are white. Each white object corresponds to a bit in register (three bits for the shots and four bits for the ducks), and these bits are anded bit wise with white pixel and so if the bit is zero the object is black and if the bit is one the object is white. In order to determine where the object is the module makes use of parameters (d1, d2, etc.) that mark where the one object ends and the next begins. The output of the rest of the sprite is exactly the same as the bush and tree sprites.

4.1.7) Duck Hunt Game Logic Module (Rachel Bainbridge)

The Duck Hunt Game Logic is very similar to the pong game logic from lab five. The module takes inputs from the pixel clock - vclock, hcount, vcount, hsync, vsync, blank, and outputs the game's hsync, vsync, and the pixel to be written to the screen. The duck moves once per frame. The frame pulse is created by taking vsync and negating it and delaying it one clock cycle by putting it in a register. A pulse is then assigned to the current vsync negated and anded with the negated register. Similarly, a bullet pulse is created for when the user presses the fire button, but this pulse must be high for one frame. The value of shot is put into a register when pulse is high to delay it a frame and then anded with the negated value to create the bullet pulse. The duck travels in diagonal lines around the screen and bounces off the sides and top. It does not bounce off the bottom, but rather a specified height from the bottom of the screen called HORIZON leaving space for the shots and score counters. The duck's speed is set with the register pspeed, which does not vary for this game. The duck's current location are stored in one x and one y register. The duck is moved by subtracting or adding pspeed to the x and y coordinates of the duck. The duck's current direction is stored in two one bit registers. Every frame, the bits are concatenated and put into a case statement to determine which way the duck will move. A one indicates the pspeed must be subtracted to the corresponding coordinate and a zero indicates that pspeed must be added. If the duck hits the lower bound, it's current position must be reset to pspeed minus one, so that its position does not roll around to a higher coordinate and cause odd things to happen. The duck will also leave the game and a new one will be released if the duck bounces more than three times on the top edge of the screen. The game logic keeps track of how many times the ducks has hit the top of the screen by incrementing a register bounces each time

the duck's y direction changes from up to down. If it hits the top and bounces is equal to three then it flies off the top of the screen, the score bit corresponding to the current duck is set to zero, and a new duck is started. A new duck will also be started if the duck is shot by the player. Every time the player fires the gun the xy coordinate of the cursor which comes in from Dan's half of the project is compared with the duck xy coordinate. If the cursor is within the duck (a 30 by 30 square) then the duck is killed and dead goes high and the score bit corresponding to the current duck is set to one. When the duck is dead it only has negative y velocity and it falls at the same speed it flaps until it hits the horizon, and then a new duck is released. A register called totalducks keeps track of the total number of ducks the player has gone through. On reset or a new duck, the xy coordinates and direction of the duck are also set back to the originals. However, on reset score and totalducks are set to zero, while on a new duck, the score is untouched and totalducks is incremented by one. The game logic also decides which pixel is displayed from the pixels being output from the individual sprite modules. If a duck pixel is being output for the square, then it will be the one output to the screen, otherwise either the score, shot, bush or tree pixel will be output (the four can be or'd together as it is assured they will never overlap because they are stationary).

4.2) Video Processing System (Dan Southern)

The following block diagram represents an overall view of the video processing system. Data is streamed in the NTSC decoder on the Labkit in the upper left corner. The data is propagated through into the frame buffer. The Frame Buffer Reader reads the data into the pixel analyzer, which consists of several operations on each pixel. The system's ultimate output is a set of (X, Y) coordinates which are subsequently passed into the Game Logic module.

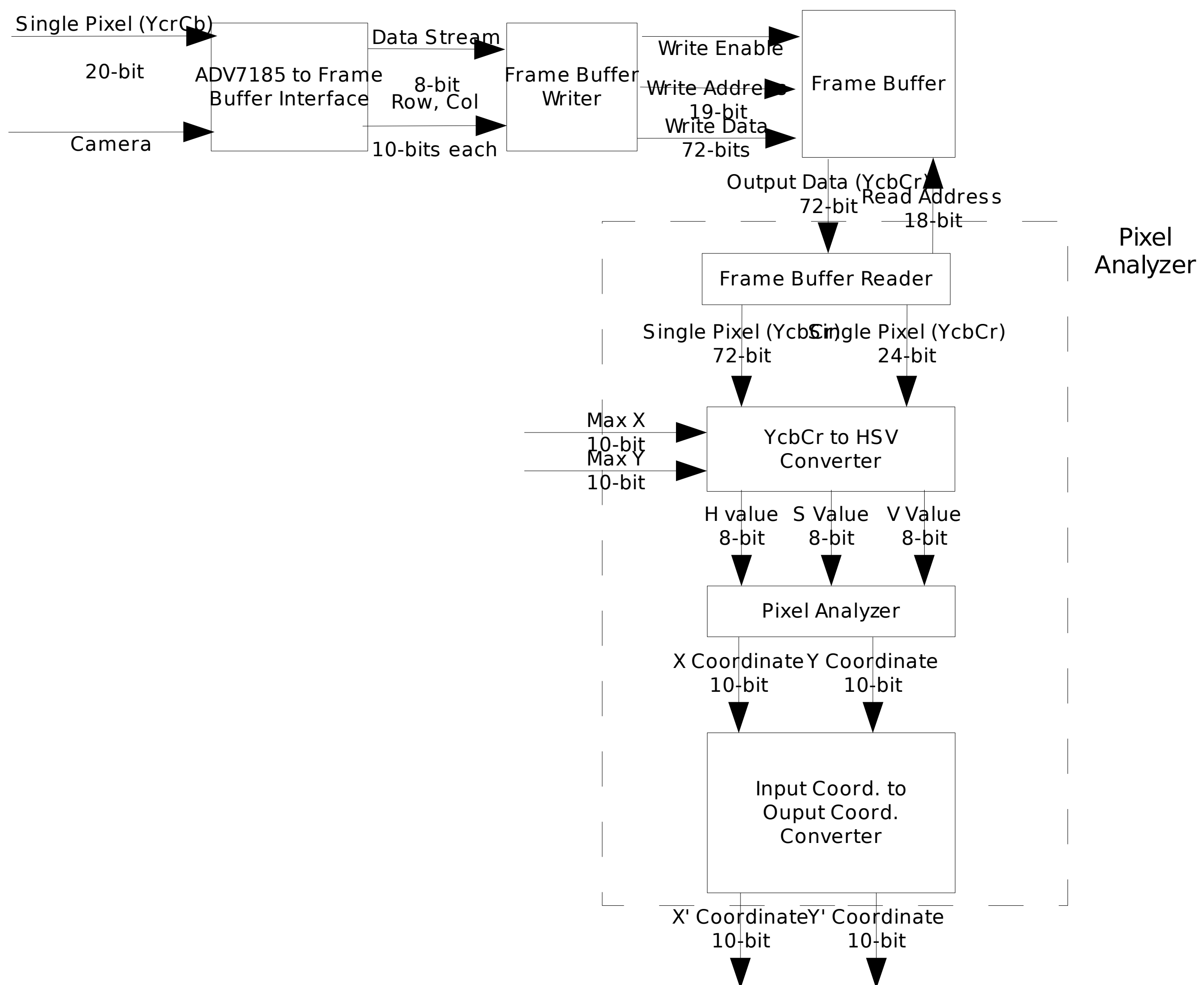

Figure 5 – Video Processing Block Diagram

4.2.1)Frame Buffer Reader (Dan Southern)

In order to determine the location of the laser pointer in the camera image, the system must iterate through the image to examine each pixel. To this end, the Frame Buffer Reader module generates control signals that are fed to the Frame Buffer to read each pixel and present it to the next stage of pixel analysis.

There are several intricacies to this process, we have to work around two other modules that also interface with the frame buffer while handling the 2 clock cycle delay in the ZBT ram's behavior. The other two modules accessing the frame buffer are the XVGA module generating pixel output data and the Frame Buffer Writer module which writes data coming in from the camera in real time. Both of these applications cannot wait to access the frame buffer in the way that this module can, since they are subject to other external timing constraints which prevent them from waiting for access to the ZBT.

Therefore, the Frame Buffer Reader module is only allowed to access the Frame Buffer when neither of the other two modules are accessing it. This is implemented through a signal passed into the Frame Buffer Reader module called Read_Success which indicates whether the module was able to add it's frame buffer operation to the queue or whether one of the other modules

performed a frame buffer operation in that clock cycle. If another module did access the frame buffer, such that the Read_Success signal is false, then the Frame Buffer Reader Module waits and halts its pipeline until it is able to queue a read command into the Frame Buffer.

By engineering this module to handle arbitrary delays in access to the Frame Buffer, the timing constraints on the XVGA and Frame Buffer Writer modules are significantly reduced. The performance of this module is affected by the fact that it has effectively the lowest priority when accessing the frame buffer; however, this function is the least time critical. A user of the system is unlikely to notice any performance degradation as long as the module is able to iterate through the frame buffer at about 20Hz. This level of performance should be achievable since the Frame Buffer Writer accesses the Frame Buffer relatively rarely, and the XVGA module accesses it exactly 1 out of every 4 clock cycles. The module will iterate through the frame buffer much faster than this at a rate slightly less than ¾ * 65MHz / (1024 * 768) ≈ 62Hz as a worst case figure since the dimensions of the NTSC image will be less than 1024x768.

The image is read out of the frame buffer starting in the upper left corner, which corresponds to the pixel address (0, 0). The module iterates first across each column and the down to the next row until it has reached the max values programmed in for the size of the image with this particular camera. The camera produced a usable image size of approximately 720 pixels wide by 460 pixels high, so the Frame Buffer reader module only iterated through the first 720 columns and 460 rows of the frame buffer. Figure 6 shows the order in which pixels are read from the frame buffer.

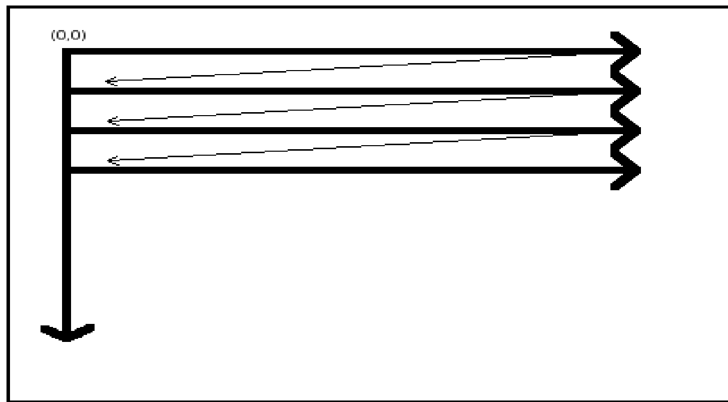

Figure 6 – Frame Buffer Access Order

Figure 7 below details the connection specification of the Frame Buffer Reader Module. The module is synchronized with the 65MHz system clock (clock_65mhz), has connections for interfacing with the frame buffer (ZBT_Data, Read_Address), and an interface for presenting data to the Pixel Analyzer module in a form that is easily dealt with. The pixel information is output on Read_Data, the pixel coordinates are output on Read_X and Read_Y, and the New_Frame signal indicates that analysis of a new frame should be started because the end of the frame buffer has been reached.

Type	Name	Bus Width	Description
Input	reset	1	FSM should return to default initial state when high
Input	clock_65mhz	1	System clock input to synchronize with Main FSM
Input	ZBT_Data	72	Input from the read port of the Frame Buffer
Input	Read_Success	1	High Value indicates that the module was able to queue a read on the Frame Buffer.
Output	Read_Address	19	The Address to read from the Frame Buffer

Output	Read_Data	72	The data from the frame buffer corresponding to the coordinates (Read_X, Read_Y)
Output	Read_X	11	X Coordinate of the current pixel being output by the module
Output	Read_Y	10	Y Coordinate of the current pixel being output by the module
Output	New_Frame	1	High Value indicates that this clock cycle starts us iterating through the Frame Buffer again from the start

Figure 7 – Frame Buffer Reader Wiring Specification

4.2.2) Frame Buffer Writer (Dan Southern)

This module is derived from an example on the course website, however I have made significant modifications to the source, including addition of handling the color data, reworking the timing of access to the frame buffer (the original version wrote to the frame buffer 4 times as much as was necessary), and handling frame buffer access collisions.

The Frame Buffer Writer module serves as the interface between the NTSC decoder on the Labkit and the frame buffer. As Data is streamed from the NTSC decoder, this module aggregates it into packets describing individual pixels before generating the control signals to write the data packets into the frame buffer.

The pixels are output from the NTSC decoder on the Labkit as a sequence of luminance, or the Y component, and chrominance, the Cr and Cb components. The data that is presented serially on the output data bus of the decoder chip, with each Y component sharing a Cr and Cb data point with a neighboring Y component. The data stream takes on the following format:

Y Cr Y Cb Y Cr Y Cb Y Cr...

As soon as the Frame Buffer Writer module has collected four Y data points and a set of two each of Cr and Cb data points, a packet of data is complete and the information is ready to be written into the frame buffer. This includes generating the address where the data will be written, outputting the pixel data on the frame buffer interface, and asserting the write enable signal. This process may take more than one clock cycle, as the XVGA module is given higher priority for accessing the frame buffer. If the module attempts a write while the XVGA module is reading the frame buffer, the module will be informed of this by receiving a high value on its Write_Failed input. In this situation the module will perform the write again in the next cycle, in which case it is guaranteed to succeed since the XVGA module accesses the frame buffer at most every 4th clock cycle and the Frame Buffer Write Module has the next highest priority for frame buffer access.

The following is a complete list of the connections to the Frame Buffer Writer Module. It's inputs are basically the set out of outputs from the NTSC decoder module, other than the control signals Write_Failed and Write_Completed which coordinate access to the frame buffer with other modules.

Type	Name	Bus Width	Description
Input	clock_65mhz	1	FSM should return to default initial state when high
Input	Video_Clock	1	System clock input to synchronize with Main FSM
Input	fvh	3	A concatenation of the field, vsync, and hsync signals
Input	data_valid	1	High indicates data is ready to be read from ADV7185
Input	Yin	8	Luminance value from ADV7185
Input	Crin	8	Cr chrominance value from ADV7185
Input	Cbin	8	Cb chrominance value from ADV7185
Output	ZBT_Write_Address	19	Address to write to in the frame buffer
Output	ZBT_Write_Data	36	Packet of information to put in one cell of the frame buffer

Output	ZBT_Write_Enable	1	High value indicates the module is ready to write to the frame buffer
Input	Write_Failed	1	High value indicates that the frame buffer was busy when we tried to write, and we should try to write again in the next cycle
Input	Write_Completed	1	High value indicates that data was successfully written into the frame buffer in the last clock cycle.

Figure 8 – Frame Buffer Writer Wiring Specification

4.2.3) Frame Buffer (Dan Southern)

The implementation of this module is based on the ZBT interface borrowed from the course website.

The frame buffer stores a single NTSC frame of data in order to facilitate asynchronous data accumulation processing. By reading video data into the frame buffer, other modules are granted access to the entire image at any time. The data in the frame buffer will always be a mixture of two video frames as the contents of the buffer are updated; however, for this application we can assume that the image data is continuous enough across a couple of video frames that this will not affect the performance of the system in any way.

The Frame buffer is implemented as the two ZBT's concatenated together, with each pixel in the NTSC image receiving one byte in each ZBT ram, or 2-bytes total. Since the ZBT ram is 36-bits wide, or 72-bits wide using both, this allows us to store 4 pixel of data in each location in the ZBT ram. As it turns out, we can get away with storing only 2-bytes per pixel since adjacent pixels share some color information such that on average each pixel is only 2-bytes. Although there are actually 3 values that describe each pixel, a set of 8 values can approximately describe 4 pixels. We accomplish this by storing a set of chrominance (Cr, Cb) bytes for every two pixels in one of the ZBT rams while storing each pixels individual Luminance (Y) byte in the other ZBT ram.

In Figure 9 shown below, we can see that one ZBT ram (on the left) stores only luminance (Y) data for each pixel. Each location in the ZBT is outlined in a thick black line, so there are 4 pieces of luminance data stored in each location in the ram. The other ZBT ram stores chrominance data that is shared between two pixels. The coordinates of the pixels with which the data is associated with is indicated in parenthesis.

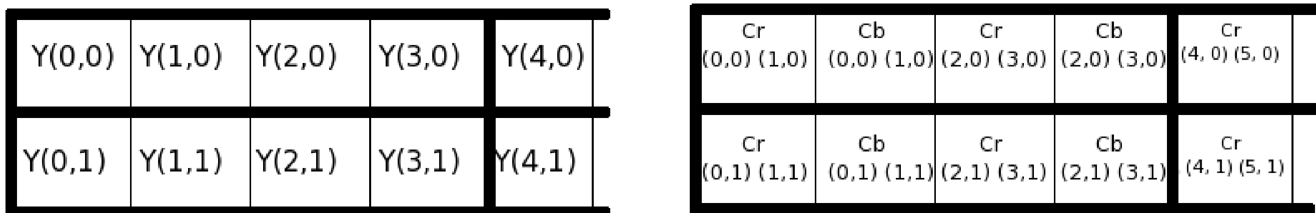

Figure 9 – Frame Buffer Data Configuration

The frame buffer is implemented in Verilog as a pair of interfaces to the ZBT rams, which share an address port, and write enable signal. The write and read data are the concatenation of the the two 36-bit connections. The set of connections to the frame buffer is straightforward. The module takes an address, and Write Enable signal and (optionally, if the write enable is high) data to write. The following figure outlines all of the connections to the module.

Type	Name	Bus Width	Description
Input	clock_65mhz	1	Ram clock
Input	ZBT_Enable	1	Low value causes RAM to “sleep”, we tie this value to high since we constantly access the ram

Input	ZBT_Write_Enable	1	A high value enqueues a write operation. The data at the write port and the address are clocked in. The write will happen two cycles later
Input	ZBT_Address	2	The address at which the operation on the ram will take place, whether it be a read or write
Input	ZBT_Write_Data	72	Data that will be written at the address specified if ZBT_Write_Enable is set high
Output	ZBT_Read_Data	72	Selects which timer parameter to drive onto the timer_value output pins

Figure 10 – Frame Buffer Wiring Specification

4.2.4) Pixel Analyzer (Dan Southern)

As the pixel data is sequentially read from the frame buffer by the Frame Buffer Reader Module, the Pixel Analyzer module examines each pixel in order to determine whether it matches the description of the cursor pixel. In this case we can distinguish the cursor pixels both from their relatively high luminosity and from their color. Red pixels with an above average luminosity are considered to be part of the cursor.

There are several methods employed to make the distinction between cursor pixel and non-cursor pixels. The Pixel Analyzer module contains another module which converts the YCrCb video data from the frame buffer into other color spaces which are more easily worked with, specifically the RGB color space and the HSV color space. The module which performs the conversion is documented below.

There are several strategies that can be employed here: we can put the data through the converter, which is pipelined, and deal with the pipeline delay in order to analyze the data in either the RGB or HSV color spaces. This method should theoretically lead to the most accurate cursor detection. Another strategy is to accurately detect areas of the image with a discontinuous spot of high luminosity, which may be sufficient without any color detection.

After experimenting with several combinations of the methods mentioned above, I found that against a white background, which is the least optimal environment, searching for relative luminosities proved to be the best method. The best solution would be to parameterize this module such that one could easily adjust the requirements for matching pixels to the cursor, however I left the module optimized for the application at hand with the following functionality.

The module computes an average luminosity over the entire image by summing the luminosities of each individual pixel as the data is fed in. Once the module receives the signal that the end of the frame has been reached, we take the 8 highest order bits of the sum to represent the average luminosity. On top of this average, we can program in an additional threshold using the switches on the Labkit, and only pixels with luminosities higher than the sum of the average plus the threshold value are considered for the next stage of analysis. By setting a correct threshold value, we can achieve performance where only the cursor pixels are passed to the next stage of analysis. In this regard, spending extra time working on luminosity analysis paid off, as it was really irrelevant how the cursor pixels values would be averaged together. The module originally included a center of mass calculator for the pixels matching the description, however I was able to replace this with a trivial method of simply finding the pixel with the highest luminosity. During the analysis of each frame this highest value may change several times, so we only update the output coordinates of the module when the New_Frame signal is received.

The final output X and Y coordinate, which are subsequently passed on to the Game Logic, are first scaled appropriately so that each corner of the image corresponds to each corner of the VGA display. This conversion is basically an affine transformation of X from the range [0..720] to

[0..1024] and for Y from [0..450] to [0..768]. Both functions were approximated using excel, and then turned converted into a look up table module. The output (X, Y) coordinates of the Pixel Analyzer Module are ultimately connected to the output of this set of look up tables.

Type	Name	Bus Width	Description
Input	clock_65mhz	1	System clock
Input	reset	1	Module reset signal, high value resets module
Input	Read_Data	72	Pixel information from the Read Frame Buffer Module
Input	Read_X	11	The X coordinate of the pixel information
Input	Read_Y	10	The Y coordinate of the pixel information
Input	Read_Success	1	Indicates whether the last read operation on the frame buffer was successful. If this signal is low, the pipeline is halted for one cycle
Input	New_Frame	1	High value indicates that the Frame Buffer Reader has just presented this module with the last piece of information for the frame. The next values will be from the next frame.
Input	Threshold	8	This is a luminosity value above the average luminosity for which we consider pixels. This allows us to adjust the module performance for best results in many ambient light settings
Output	X	11	The X coordinate of the cursor in the image
Output	Y	10	The Y coordinate of the cursor in the image
Output	Average_Brightness	8	The Average brightness of all pixels in the image. This is output to the LEDs for some visual feedback during operation.

Figure 11 – Pixel Analyzer Wiring Specification

4.2.5) YCrCb to HSV Converter (Dan Southern)

The purpose of this module is to take YCrCb pixel information at its inputs and output the corresponding RGB and HSV color space data. The design of this module posed many interesting design problems. In the calculations for the conversions, there are several division required and several multiplications as well. The process actually lends itself well to pipelining, as the transformation is carried out in several discrete operations over the 3 values.

First the data is converted from the 3 8-bit Y Cr Cb values into three 10-bit R G B values. These intermediate results are also part of the output of this module, as we may want to generate RGB data for display with the VGA or for analysis of the pixel information in this color space. This step of the conversion includes only a single set of multiplications carried out in parallel and then two additions. Therefore this step will benefit only marginally from pipelining by separating the multiplications and additions into two stages.

The next step, conversion from the RGB values into a Hue, Saturation, and Value is the more difficult and interesting problem. This conversion includes divisions by 8-bit values. Rather than implementing this division, I created a lookup table for the function f(x) = 256/x. In order to approximate the value a / b, we can multiply a * f(b), and then divide by 256 by shifting right 8 bits. This method is precise enough for our purposes, and reduces the delay for this calculation down to about the delay of a multiply operation.

This module takes as input the YCrCb pixel value and produces as output both an RGB value and an HSV value. The details of the connections to this module are outlined in the table below.

Type	Name	Bus Width	Description
Input	clock	1	System clock
Input	Y	8	Y value of pixel to convert

Input	Cr	8	Cr chrominance of the pixel to convert
Input	Cb	8	Cb chrominance of the pixel to convert
Output	R	10	Red value of the pixel (on a 10-bit scale)
Output	G	10	Green vale of the pixel (on a 10-bit scale)
Output	B	10	Blue value of the pixel (on a 10-bit scale)
Output	H	8	Hue value of the pixel
Output	S	8	Saturation value of the pixel
Output	V	8	The “Value” of the pixel

Figure 12 – YCrCb To HSV Converter Wiring Specification

4.2.6) NTSC Decoder

This module was borrowed from the course website. The purpose of this module is to read the output of the ADV7185 to grab the pixel data and coordinates. This data is then fed into the Frame Buffer Write Module to be aggregated and written into the frame buffer. This module also includes and interface to setup the ADV7185 into the correct mode when the Labkit is powered on.

4.2.7) Frame Buffer Display

This module was borrowed from the course website. This module reads the pixel information out of the frame buffer for use in outputting in the VGA data stream. This is primarily used for debugging purposes, as in the final product there is no need to output the image from the camera.

4.8) XVGA

This module was borrowed from the course website. This module generates the control signals for a 1024x768 VGA display.

5) Conclusions

5.1) System Integration

We experienced minimal problems integrating our two modules. We were able to develop independently until we each had achieved our desired functionalities. We were then able to simply combine our Verilog sources, and connect the output of the Video Processing Module (the cursor location) straight into the game logic.

Our integration went smoothly because our project was divisible into two very distinct pieces. The independent operation of both pieces, and coordination on which resources on the Labkit were available to each of us (I.e. ZBT Ram) resulted in the prevention of integration issues.

5.2) Testing and Debugging

5.2.1) Game Logic and Sprites (Rachel Bainbridge)

Testing the sprites was very easy, because there was visual feed back. In the beginning, when the sprites were not showing up on the screen at all, or just as noisy blue blobs, test ROMs were created and the address was specified by the lab kit switches and the output of the ROM was

wired up to be displayed on the led hex display. By changing the switches, one could see whether the correct information was on the ROM and the output was just being handled incorrectly, or that the ROM was not functioning properly or had the wrong information on it. Before actually using a sprite, the module was tested to see if it would output just a block of color to the screen in the specified place. Also the game logic was created and tested using a blob instead of a sprite from a ROM to see if it would work functionally without interference from the sprite.

5.2.1) Video Processing (Dan Southern)

Testing the video processing proved challenging at first until a had the system in a stable state where I was able to display video onto the screen. Once I had achieved a form of visual feedback, I was able to start working on testing various filter applied to the pixel data in order to get a visual sense of how accurately I could distinguish interesting pixels from the background image.

I used code borrowed from a past project to handle the complicated task of initializing the NTSC decoder on the Labkit and then interpreting its output. Although the code was already written, I found that it was necessary to develop an understanding of almost the entire process in order to make the modifications to the system I needed to make. I expanded the system to handle the chrominance data as well as just the luminance, and I had to implement a priority system for modules to access the frame buffer. This proved to be the single biggest challenge I encountered as an error in this process would generally disable all of the operations happening on the frame buffer so that I was stuck with just a blank screen, which made it difficult to diagnose the problem. The logic analyzer proved invaluable in this situation.

Overall I found that I was able to spend most of my time developing the modules rather than tracking down bugs, although I was forced to deal with strange behavior, where I would make a seemingly innocuous change to the source code (for example, something simple like renaming a variable) which would cause inexplicable behavior such as static in the video signal, or the hex display on the Labkit to stop working. I suspect that this was a product of how the hardware was organized on the Labkit which could vary from compile to compile, and would occasionally result in sub-optimal configurations.

5.3) Final Thoughts

5.3.1) Rachel's Final Thoughts

If I had to do this project over again, there wouldn't be much I would change. I planned ahead as much as I could and tried to add on very small bits at a time. Implementing the sprites separately from the game logic at first helped a lot too. If someone were to do a similar project I would tell them that the hardest part to do is make the sprites actually appear on screen.

5.3.2) Dan's Final Thoughts

I enjoyed developing the video modules. The engineering problems were interesting, although it is somewhat frustrating to have to wait through the delay to compile code in order to test a bug fix. If faced with developing another project on the Labkit, I would spend more time setting up useful debugging facilities to eliminate as many extra compilation delays as possible. Also, I think I would develop my modules more independently in order to reduce the overall complexity throughout the module design process. This would have saved me some time, since were many more variables in my system than there could have been that only slowed down my debugging efforts.

6) Appendices

Appendix A: Matlab BMP to COE code

```
function BMPtoCOE(image_name)
%Converts a 16 color bitmap image to a Xilinx .COE file
%Was written so students could use a FPGA to display images on a VGA
%monitor
%read bmp data in and display it to the screen
[imdata,immap]=imread(image_name);
image(imdata);
colormap(immap);
numpixels=numel(imdata);
%create .COE file
COE_file=image_name;
COE_file(end-2:end)='coe';
fid=fopen(COE_file,'w');
%write header information
fprintf(fid,';******************************************************************\n'
)
;
fprintf(fid,';**** BMP file in .COE Format *****\n');
fprintf(fid,';******************************************************************\n'
)
;
fprintf(fid,'; This .COE file specifies initialization values for a\n');
fprintf(fid,'; block memory of depth= %d, and width=4. In this case,\n',numpixels);
fprintf(fid,'; values are specified in hexadecimal format.\n');
%start writing data to the file
fprintf(fid,'memory_initialization_radix=16;\n');
fprintf(fid,'memory_initialization_vector=\n');
%convert image data to row major
newimdata=transpose(double(imdata));
%write image data to file
for j=1:(numpixels-1)
fprintf(fid,'%s,\n',dec2hex(newimdata(j)));
end
%last data value supposed to have a semicolon instead of a comma
fprintf(fid,'%s;\n',dec2hex(newimdata(numpixels)));
%clean shutdown
fclose(fid)
```

Appendix B: Look Up Table Verilog

```
////////////////////////////////////////////////////////////////////////////////
//
// look up table for 4-bit RGB color
//
////////////////////////////////////////////////////////////////////////////////


module look_up_table (pixel, r, g, b);
```

```
	input [3:0] pixel;
	output [7:0] r, g, b;
	reg [7:0] r, g, b;

	always@(*) begin
		case(pixel)
			4'b0000: begin b = 0; g = 0; r = 0; end // black
			4'b0001: begin b = 0; g = 0; r = 132; end // red brown
			4'b0010: begin b = 0; g = 132; r = 0; end // green
			4'b0011: begin b = 0; g = 132; r = 132; end // brown green
			4'b0100: begin b = 132; g = 0; r = 0; end // dark blue
			4'b0101: begin b = 132; g = 0; r = 132; end // purple
			4'b0110: begin b = 132; g = 132; r = 0; end // blue green
			4'b0111: begin b = 132; g = 132; r = 132; end // dark grey
			4'b1000: begin b = 198; g = 198; r = 198; end // light grey
			4'b1001: begin b = 0; g = 0; r = 255; end // orange red
			4'b1010: begin b = 0; g = 255; r = 0; end // vivid green
			4'b1011: begin b = 0; g = 255; r = 255; end // yellow
			4'b1100: begin b = 255; g = 0; r = 0; end // blue
			4'b1101: begin b = 255; g = 0; r = 255; end // pink purple
			4'b1110: begin b = 255; g = 255; r = 0; end // light blue
			4'b1111: begin b = 255; g = 255; r = 255; end // white
			default: begin b = 0; g = 0; r = 0; end // default black
		endcase
end
endmodule
```

Appendix C: Score Sprite Module Verilog

```
//////////////////////////////////////////////////////////////////////////////
//
// score sprite: generate score on screen
//
//////////////////////////////////////////////////////////////////////////////

module score_sprite(hcount, vcount, clk, score, pixel);
	parameter WIDTH = 90;
	parameter HEIGHT = 60;
	parameter MY_X = 900;
	parameter MY_Y = 700;
	parameter d1 = 32; parameter d2 = 44; parameter d3 = 56; //locations of the ends of
different ducks

	input [10:0] hcount;
   input [9:0] vcount;
	input [3:0] score;
	input clk;
   output [3:0] pixel;

	reg [3:0] pixel;
	reg [12:0] addr;
```

```
        wire[3:0] dout;
        score5400x4 scoreram1(addr, clk, dout);
        always@(posedge clk) begin
                                if((hcount > (MY_X + WIDTH))&&(vcount > (MY_Y + HEIGHT)))
                        addr <= 0;

         if((hcount >= MY_X && hcount < (MY_X+WIDTH)) &&
         (vcount >= MY_Y && vcount < (MY_Y+HEIGHT))) begin
                if(dout == 15) begin
                        if(hcount < d1+MY_X) pixel = dout&{4{score[0]}};
                        else if(hcount < d2+MY_X) pixel = dout&{4{score[1]}};
                        else if(hcount < d3+MY_X) pixel = dout&{4{score[2]}};
                        else pixel = dout&{4{score[3]}};
                        end
                else pixel = dout;

                addr <= addr+1;
          end
          else pixel = 0;
        end


endmodule
```

Appendix D: Duck Sprite Module

```
////////////////////////////////////////////////////////////////////////////////
//
// duck sprite: generate tree on screen
//
////////////////////////////////////////////////////////////////////////////////

module duck_sprite(x,y,hcount,vcount,clk,dead,orientation,pixel);
   parameter WIDTH = 30;
   parameter HEIGHT = 30;

   input [10:0] x,hcount;
   input [9:0] y,vcount;
        input clk,dead, orientation;

   output [3:0] pixel;

        reg [3:0] pixel;
        reg[11:0] faddr;
        reg[10:0] daddr;
        reg[10:0] fduck_no;
        reg[9:0] dduck_no;
        reg[9:0] line = 30;
        reg[4:0] pix;

        wire[3:0] dout;
        flyingducks2730x4 duckram1(faddr, clk, dout);

        wire[3:0] ddout;
```

```
	deadducks1800x4 duckram2(daddr, clk, ddout);

	wire enbl;
	Divider duckdivider(clk, enbl);


   always @ (posedge clk) begin
		if(dead) begin //duck is dead use dead duck sprites
			faddr <= 0;  //return flying duck sprites to original state
			fduck_no <= 0;

			if (enbl) begin
				if(dduck_no == 900) dduck_no <= 0; //increase duck number (starting
addr) once per frame
				else dduck_no <= 900;
			end

			if((hcount > (x + WIDTH))&&(vcount > (y + HEIGHT)))
				daddr <= dduck_no; //last pixel reached, reset address


			if ((hcount >= x && hcount < (x+WIDTH)) &&
		(vcount >= y && vcount < (y+HEIGHT))) begin
				daddr <= daddr + 1;
			pixel = ddout;
			end
			else pixel = 0;
		end

		else begin //duck is alive use flying duck sprites

			daddr <= 0;  // return dead duck sprites to orginal state
			dduck_no <= 0;
			if (enbl) begin
				if(fduck_no == 1800) fduck_no <= 0; //increase duck number (starting
addr) once per frame
				else fduck_no <= fduck_no+900;
			end
			if((hcount > (x + WIDTH))&&(vcount > (y + HEIGHT)))begin
				faddr <= fduck_no; //last pixel reached, reset address
				line <= WIDTH-1;
				pix<=0;
				end



			if ((hcount >= x && hcount < (x+WIDTH)) &&
			(vcount >= y && vcount < (y+HEIGHT))) begin
				if(orientation) begin //read from rom backwards (this is a bit tricky)
					if(pix == WIDTH-1) begin line <= line + WIDTH; pix <= 0; end
					else pix <= pix + 1;
					faddr <= fduck_no + line - pix;
				end
				else faddr <= faddr + 1;
```

```
                pixel = dout;
            end
            else pixel = 0;
        end
    end
endmodule
```