FPGA Hunt

Syed Ahmed, Pete Kruskall, Yuetian Xu

December 15, 2007

1 Introduction

FPGA Hunt is a throwback to the games of yore. Much like the seminal game Duck Hunt, FPGA Hunt is a light gun controlled reflex-testing game where one must shoot objects that fly through the air. FPGA Hunt, however, introduces a modern twist onto the old game. In addition to a wide array of potential targets, each with their own personality and game effects, FPGA Hunt takes the 2D Duck Hunt engine and evolves it into a 2.5D engine, allowing what are typically static elements on the screen to become dynamic 3D elements, immersing the user in a semi-realistic environment. Furthermore, allow users to shoot the dog that used to mock them when they missed the duck. In the next few sections, we will explore how the zapper, game logic, and graphics modules worked together to make this gaming experience possible.

2 Zapper Overview - Syed Ahmed

The zapper we obtained for this project was the EMS TopGun. This zapper uses LED lights placed around the screen to determine the orientation of the device. A tiny camera at the end of the zapper is able to detect the LEDs and with some calibration, determine which portion of the screen the zapper is pointed at. The software that comes with the zapper allows for the user to control the PC's mouse with the device. When the trigger is pressed, the software interprets this as a mouse click. From there, a java program determines the mouse's coordinates and sends the information to the labkit via the serial port. The zapper module then interprets the data from the serial port and sends information related to the zapper's x and y position to the game logic module.

2.1 Zapper Logic: PC Output

The process of interpreting the PC's mouse coordinates and sending the information to the labkit via the serial port is completed entirely with java programming. The program first analyzes the computer's hardware to determine which communication ports are available. Depending upon the ports that are available, java displays a window to the user to allow him or her to select the port

they wish to use. After the user has selected their desired port, the program initializes the port and prepares for output. It then creates a blank 800×600 pixels window that encompasses the entire screen (when the screen resolution is 800×600). This window detects the coordinates for the mouse whenever it detects a mouse click. The software then sends a six digit number through the serial port. The first three digits relate to the x position, while the last three relate to the y position. This can be done indefinitely or until the window is closed by pressed Alt + F4.

2.2 Zapper Logic: Labkit Overview

The labkit receives the output from the PC through the RS-232 or serial port. The zapper module is responsible for interpreting this data and determining the information that it is carrying. The serial port sends a constant datastream of ones until it receives an output from the PC. At this point, the stream becomes zeroes for a period of time related to the baud speed of the port. This speed is set through the java software on the PC and for the purposes of this project, has been set to 115200. This means that every bit sent from the PC via the serial port lasts for approximately 235 clock cycles when using a 27 MHz clock on the labkit. After the initial start bit, the serial port then sends 8 bits related to the first digit in the x position. This is followed by a transition phase that consists of a '1' and a '0'. The stream then moves on to the second digit in the x position value, and proceeds in this manner until it reaches the final digit for the y position value. Upon completing this final digit, it immediately returns to a stream of ones until the next position signal is sent.

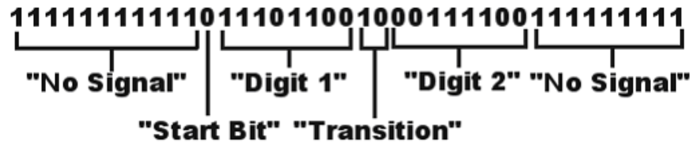

Figure 1: Sample Two Digit Serial Transmission Signal

2.3 Zapper Logic: Labkit: States

In order to facilitate the interpretation of the changing properties of the datastream, the zapper module has various states that relate to the information that it is currently analyzing. When the datastream is sending a continuous stream of ones, the zapper remains in a "no communication" state. In this state it only seeks to ensure that all the timers and expiration signals are reset or turned off. When the zapper module detects the start bit, it transitions to the "start_sequence" state. In this state, the module simply counts the number of clock cycles it takes a 27 MHz clock to count one bit from the serial port (which

happens to be approximately 235 with a baud speed of 115200). Instead of counting to this number, the module counts to a number slightly below it (approximately 220) to insure that all subsequent recordings are from the correct value, and then transitions to the next state, "x_digit_1". In this state, and all other states referring to a digit, the module has two timers, a macroscopic timer and a microscopic timer. The macroscopic timer is responsible for determining when the module has finished reading the 8 bits concerning the digit. This timer is responsible for triggering the expiration signal that transitions the module to the next state. The microscopic timer is responsible for determining which bit in the 8-bit sequence the module is currently reading. This will be discussed further later in the report.

The next state that the module goes to is the transition state. Once again, in this state the module only counts the number of clock cycles it takes for two bits to pass before transitioning the system to a state related to the next digit. In this manner, the state transitions continue, from a digit state to a transition state, until the module reaches the final y digit. When it has finished reading this final digit, the module enters its "transmit_data" state. In this state, the module sets the x and y position outputs to their new values and turns the "zapped" signal on. The output for the x and y position will remain constant until the value is changed, but the "zapped" signal will only last for as long as the module remains in the "transmit_data" state. Since the gaming logic works every 1/60th of a second, the zapper module is set to remain in the "transmit_data" state for that long.

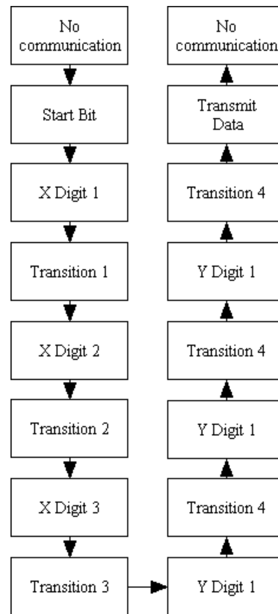

Figure 2: State Transition Diagram

2.4 Zapper Logic: Macroscopic Timings

There are four macroscopic timers running within the zapper module. The first timer, "counter1" is responsible for measuring the length of time for which the zapper module will be reading the start bit from the serial port. The second timer, "counter 2", is responsible for determining when the module is no longer reading a digit. The next timer, "counter3", counts for the amount of time that the serial port is outputting the transition one and zero bit. The final counter, "counter4", last for 1/60th of a second and is responsible for measuring the period of time for which the zapped signal will be outputted for.

2.5 Zapper Logic: Microscopic Timing

When the zapper module is in a state where it is reading a digit, it must read the values from the serial port and store them into appropriate registers. In order to this, a mini timer has been implemented that will count to approximately 235 (the length of time that one bit from the serial port is maintained) and then increment a position counter by 1. Depending on which digit is being read, the position counter will be used to store the value in its appropriate register. Once the module has finished reading all the positions for a given digit, each store integer will be placed in it appropriate position in a 4 bit value. When all of the digits for a given number have been recorded, the values will be multiplied by the appropriate value and stored in a 10 bit register. For example the number 762 will first be recorded as three separate numbers: a seven, a six, and a two. These numbers will then multiplied by the appropriate value and summed to digitally represent the number 762: $(7 \times 100) + (6 \times 10) + (2 \times 1)$.

2.6 Testing

The Zapper module was tested using the Logic Analyzer. When a signal was sent from the Java software on the PC, an output was displayed on the monitor indicating what the coordinates of the mouse position were. The x and y position outputs from the labkit's zapper module were displayed on the Logic Analyzer and this value was compared to the value on the PC to determine the modules accuracy. This process was repeated multiple times to ensure consistently correct behavior.

2.7 Integration Testing

When the zapper was first connected to the gaming logic, the hit margin was increased to a very high level to ensure that the "zapped" signal was functioning correctly. The hit margin was then slowly decreased so as to verify the accuracy of the zap.

2.8 Recommendations for Future Designers

Future programmers should look into creating a module that can read and interpret any output from the serial port and then verify whether that output makes sense or not. In the manner that the zapper module was implemented for this project, the output from the PC must be precisely what the zapper module expects for it to be. Otherwise, the zapper will completely misinterpret the signal. With a more robust module, this can be avoided.

3 Game Logic - Yuetian (Peak) Xu

The game logic module takes the inputs from the zapper module and changes the state of the objects which are outputted to the graphics module. The inputs are the trigger, the x position, and the y position of the shot being fired. The objects outputted to the graphics module are bit strings which are the object type, object status, object x location, object y location, and the object depth concatenated together.

3.1 Timing

There are several clock speeds that are used throughout this system. The input from the light gun is transmitted over serial which is one speed, most of the modules ran at 27MHz, the graphics portion ran at 65MHz, and the game logic runs at 60Hz. The timing problems were handled by having signals being held as levels.

Our functional spec became that the zapper module would hold each trigger pull received high for at least 1/60 of a second, and output x and y position as levels until it changes. Similarly, the game logic module will constantly output the various object bit strings as levels that change at most 60 times a second to the game logic module. These design decisions enabled things to work together without worrying about timing too much. A small divider module was also made to generate a 60Hz clock signal from the 27Mhz signal.

3.2 Main Game Logic

This module is the main logic unit. This module's design is primarily a finite state machine to select which stage the game is currently in. In each stage, all of the shootable objects have their own finite state machines.

The original design contained a provision for both background and foreground objects. However, we soon realized our definitions for both objects utilized common bit string formats. Similarly, the compile times got very high. Thus, we chose to merge the two and just support 8 objects total. We also originally envisioned having an environment list which would store all of the different objects in memory and act as an intermediary between game logic and

graphics. Implementationally, it was more convenient to have game logic module store internal memories ofthe status of various objects. Thus, it was decided that we would hook up the game logic and graphics modules directly.

One of the design concerns is that we would not want to have objects such as geese be spawned more than once. The way I handled this was via spawn states where the necessary objects are initialized and the state variable is unconditionally updated to become the next state. This way, we would never spend more than a single clock period in any given state. While this design was great initially, it turned out to be less than stellar for more complicated logic such as having geese only spawn when the dog is near etc. In order to make sure the geese wouldn't spawn more than once, I added some registers that kept track of whether a particular goose has spawned during this stage. This worked similarly and provided some additional flexibility. This enabled behavior such as the dog patrolling the ground and rousing nearby geese.

Another issue is the matter of making sure that a sprite would appear for a certain amount of time. If the goose being shot, the dog laughing at you, or the dog holding up shot geese passed too quickly, one would miss out on the cool animation. The solution to this was to use the in game timer. Though originally planned to be part of a more FPS approach or to add more difficulty where one would fail the level if not completed within a certain time limit, this is now used for measuring the time. I had a few registers around which would store the time when the state was first entered. For example, if the goose fell to the ground, it would set the state of the dog to be that of holding up the dead goose. At the same time, it would store the system time into the dog animation time counter. If the time is now more than a certain number of seconds beyond when the dog first started holding up the dead goose, the dog would transition to the appropriate next state and disable the timer.

This function was made into an idiom in the game logic module and used very frequently so as to enable dog/goose interactions. For example, if the goose reaches the top of the screen and escapes, the dog would come out and taunt you by laughing. If you kill the goose and it falls to the ground, the dog would show up with a goose and hold it up for some time. In addition, it allowed us to show the animations for both the goose and dog being shot.

Another interesting feature was the fact that I would store a half way point for what the midway point of the dog and goose is when the goose is shot. As soon as the goose is shot, the dog jumps in that direction. Before it passes through the halfway point, it will go up into the air. After the halfway point, it will start dropping. This enables it to make a cool looking arc through the air for the jump.

The triggers for transition onto the next game stage is simply a check to see if the score has surpassed some threshold. This threshold is proportional to all the geese spawned so far. We currently award one point for each goose killed. If you should happen to miss a goose, don't despair and reach for the reset button. Due to the fact that the ability to shoot the dog is considered an important feature, we actively encourage violence towards the dog. Shooting the dog will also net you one point. This makes it especially tempting to shoot

the dog while it is laughing at you for missing a target.

In the interest of providing variable difficulty levels and perhaps tricking your friends, a set of switches was added to adjust the hit margin. By hit margin, we mean how much far outside of the target sprites you can shoot and still have the shot count as a hit. This is necessary as the laser pointer on the light gun is not perfectly accurate and can be off by as much as 80-100 pixels off at certain points due to parallax problems. We made it configurable in increments of 8 between 0 and 128. At 128, it's quite possible to blow multiple geese out of the air simultaneously. At 0, it's virtually impossible to score a hit. This feature was fairly simple to implement and entertaining to use on one's unsuspecting friends.

3.3 Testing

A unit testing module was set up fairly early on for the game logic module to allow for testing independent of the zapper and graphics modules. This is the most important for the game logic module due to the need to depend upon both other modules.

On the input side, a simple switch was wired to be the trigger and the target's x and y coordinates were fed to what the zapper's output x and y coordinates were with a variable offset controlled by other switches. This allowed for testing whether the trigger logic worked properly.

On the output side, the blob modules from lab 5 were used to show the position of the various sprites on the screen. The color of the rectangles reflected the type of object it was. This was quite successful at making sure movement code worked properly without depending upon the graphics module. This aspect was used well beyond when the graphics module was first integrated as it greatly reduced compile times and enabled faster debugging turn around time. Unfortunately, The status of the object was not easy to reflect and this test suite thus didn't quite fulfill the highest potential it could have had. One of the primary limitations was the number of colors available which could have been used to reflect not only object type but status as well.

3.4 Integration Process

The integration with graphics module was fairly smooth judging from the relatively simple specifications. We both agreed to hook up our modules together directly. Furthermore, use of level signals made things easy. We had slight miscommunication where we had opposite orders for concatenating the various values: type, status, x position, y position, and depth together to form the bit string. This led to the object teleporting around the screen. This was fairly simple to figure out.

The integration with the zapper module wasn't quite as smooth due to some timing issues. We soon decided to use a level pulse scheme much like the graphics module and things worked out fine. Also, the game logic and graphics both ran at 1024×768 resolution while the zapper ran at 800×600 resolution. For this, a

conversion factor had to be applied. For some reason, the proper ratio of 32/25 couldn't be used due to some Verilog limitation. I had to settle for 41/32 which was a very good approximation.

3.5 Advice for Future Implementations

One thing that we found very useful during integration with the graphics module was to abstract out a set of parameter for the object types and object statuses which are shared between our modules. Using the include directive to add these to both works well and made synchronization considerably easier.

Another recommmendation is to always use trigger and hold to produce level signals for inter module communication. These are far more forgiving with regard to different clock rates etc.

4 Graphics - Pete Kruskall

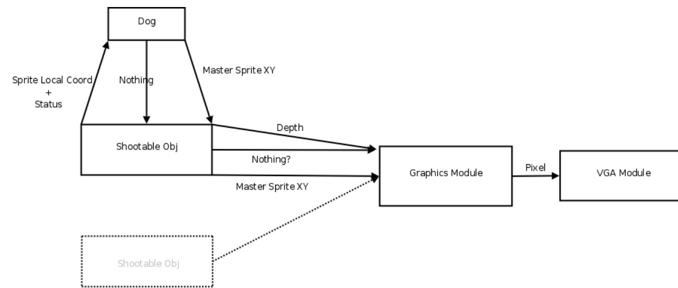

Figure 3: Block Diagram for Graphics Components Showing just one Object Module

The graphics collection of modules serves to take information on the state of the objects from the Game Logic Unit and to convert them into graphical information to display on a VGA screen.

4.1 Master Sprite

To keep the design simple, sprites are implemented using a master sprite bitmap, which contains each animation frame in one memory system. An example master sprite image is shown in figure 4. When communicating information about pixel colors from various sprites, rather than communicate actual colors, we choose to communciate between modules using the position of the pixel in this master sprite image.

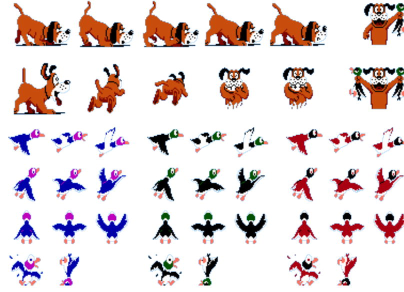

Figure 4: Master Sprite image from Version 1 of FPGA Hunt

4.2 Shootable Object

A shootable object represents any object displayed on screen whose state is communicated from the GLU. It effectively maintains state, including position, status, depth (for layering sprites), and type of object– all by maintaining a persistent connection to the GLU.

A shootable object's job is to return to the Graphics module indicators as to whether or not the given object has a candidate pixel to be drawn on the screen. This is accomplished by sending the Graphics module 3 pieces of information per screen pixel: a pixel position for the master sprite, a depth value, and an indicator as to whether or not this object actually has a sprite pixel at the requested location.

To get this information, each shootable object maintains an instance of each of the types of objects allowed in the game (whose types are listed in figure 4.3), but only requests information from that which matches the Shootable Object's type field.

The Shootable Object module forwards the screen pixel requested by the VGA unit to the object instance that the Shootable Object is supposed to represent.

4.3 Object Modules

Each object module serves to convert screen coordinates to coordinates in the master sprite image. It accomplishes this by taking in a coordinate denoting the requested pixel in terms of local object space (0,0 being the upper left hand corner of the object), as well as a number indicating the status of the object (i.e., which direction it may be facing). By summing hard-wired master-sprite locations and local object space coordinate requested, the Object modules return coordinates in the master sprite which correspond to the pixel requested.

Each object module also stores it's own height, and knows to assert a signal 'NOTHING' if the pixel requested is beyond its dimensions.

Binary ID	Type
0001	Goose
0010	Red Goose
0011	Blue Goose
0100	Dog
0101	Tree
0110	Cloud

Table 1: Object Types and their Respective Bit String IDs

4.3.1 Goose / Red Goose / Blue Goose

There are 3 separate object modules to represent geese in our game. The only difference between the three is color; however, rather than recolor one sprite collection, we chose to use 3 different collections, each already colored differently, to represent the different colors. Each goose object has a myriad of status values that can be asserted to change the look of the goose. Besides status values to denot flying in all 8 directions of the compass rose, the goose object also allows passing a 'goose-shot' status, as well as a 'goose-falling' status. Some directions of flight were not represented in the master sprite file. For instance, for a goose to be able to fly south (down), we mirrored the y-image of the sprite when sending back master sprite coordinates to the Shootable Object module.

4.3.2 Dog

The dog, from the original Nintendo game, is stored as its own shootable object. Like the goose object, the Dog allows for status values that determine its direction. We limited the Dog's directions to east and west. However, we allow for many different animation states, including: a simple "walk-and-sniff", for when the dog is searching for fallen geese; a jumping animation, for when the dog is grabbing a goose; a laughing dog animation, as well as two stationary images for when the dog has found respectively one and two geese. Much like the Goose module, we mirror sprite images for certain status value s to allow for directionality in animation.

4.3.3 Tree

The tree, unlike the Dog or Goose modules, does not have any animations. However, a different problem comes up for the Tree object. The sprite we had available for the Tree object was comparable in size to the Dog and Goose sprites, which did not appear realistic when placed on screen. As such, we decided to scale up the image when it was rendered on the screen. To accomplish this, the Tree module is told not to assert the 'nothing' signal unless the local coordinate requested is more than $(SCALE_FACTOR * SIZE_X, SCALE_FACTOR * SIZE_Y)$. When returning the master sprite coordinate, we divide by the scale

factor to simply stretch pixel values over a $SCALE_FACTOR * SCALE_FACTOR$ number of pixels. This created a grainy appearance which could have been remedied by an antialiasing filter. As division can only occur with powers of 2, we chose a scale factor of 4 for the Tree module.

4.3.4 Cloud

The cloud represents another object which, like the tree, needed to be resized to appear natural in the environment. The cloud was scaled by the same factor as the tree, and though we ended up not using it in the full implementation of FPGA hunt, was verified to work through graphics module testing.

4.4 Graphics Module

The graphics module takes in data from each of the object instantations, and on each tick of the vga pixel clock, chooses the color of the current pixel. If no object exists at the current pixel, a background color is selected based on the y-coordinate of the pixel. Low y-coordinates receive a blue color (representing the sky), and high y-coordinates receive a brown color (the ground).

Because reading from the sprite memory is a relatively slow process (requiring one full clock cycle), we do not perform any memory reads until the last possible moment. As such, the Graphics and Object modules only communicate using pixel coordinates. To further increase the number of reads we can do, we use a dual port BRAM to store the master sprite information.

At each clock tick, the graphics module determines the top two objects by depth and checks their pixel color values to ensure that the pixel color does not represent a transparent pixel.

4.4.1 Color Keying

To keep track of the two highest objects at any pixel, at each clock cycle, we iterate through each of the eight objects, maintaining at each step the two highest objects up to that point. By the end of the iteration, in two separate registers, we maintain the master sprite pixel coordinates of the two highest objects, and use these for color key calculations and color selection.

Color keying is performed to excise the solid color around each of the rectangularly shaped sprites. Once the sprite coordinates are stored for the two most prominent objects, we query the memory for both pixels in question, and determine which should be drawn. If the top most object, at the current pixel, is supposed to be transparent, the second highest object's pixel is drawn to screen. If the second highest object's pixel is also meant to be transparent, the pixel color is left as the background color.

This implementation of color keying has a weakness in that it only maintains two layers of color key information. Because color keying is in essence an iterative process, blocking assignments are needed for the logic to work correctly. To ensure that an answer is complete by the next tick of the pixel clock, we chose

to limit the number of layers we kept track of for color keying, keeping in mind that there are few situations in duck hunt for which there are more than two objects overlayed on one pixel.

4.5 Testing

To test the entire graphics module, a Graphics Test Suite module was created, which served to connect FPGA controls (switches, buttons, etc) directly to the inputs for Shootable Object modules. Correct operation was verified visually.

Verifiying color keying required an example situation in which two different shootable objects were directly overlayed with one another, allowing the depth of one object to be directly controlled by user input.

4.5.1 Integration Testing

As the only required integration was that with the GL unit, there was little possibility for error in connecting each of the 8 connections between the two units. However, due to a lack of communication in specification, when we finally connected the GL and Graphics unit, we witnessed what can only be called schizophrenic behavior. A couple of iterations later, as well as mutual assurance that each of our modules passed our own tests, we were able to figure out that we had both miscommunicated which data value resided in the most significant bit. A quick fix later, and all was working.

Most other problems were avoided by defining globally the parameters we planned to use for status values, object types, and other constants that were to be used by multiple modules.

4.6 Recommendations For Future Designers

Those wishing to implement a sprite system as I've described above would be wise to quickly verify correct operation of their BRAM memory files. Converting a 24 bit color bitmap to a 16 color COE file took a lot of work and error checking, as many of the scripts published on the internet are not entirely error-proof.

On the other hand, it is very easy to make a small mistake and attribute it to 3rd party applications, when it really is your own fault. We had a fully working implementation with all of the original sprite animations, when we decided to add on a few new objects. I had figured that by increasing the width of the sprite, that little more had to be done to ensure correct operation. However, by increasing the width, I also needed to modify the method of access for data in the BRAM (as we multiply our y coordinate by the width of the image at one point to access the data).

Working with 16 colors helped greatly in debugging this problem, as we were able to read the memory file line by line to verify the correct data was stored, and that the error was likely to lie in the verilog code itself.

5 Conclusion

With all three elements working together, we were able to produce a fully functioning and compelling game. Our game successfully took the coordinates from the zapper, interpreted them to determine which creature was being shot (or not) and displayed this on the screen. It also successfully determined and displayed other actions related to gameplay such as dog-geese interactions as well as graphical elements such as occlusion.