Tom Hughes
Joshua Ratcliff

ECE 385 Final Project
“Duck Hunt”

Block Diagram

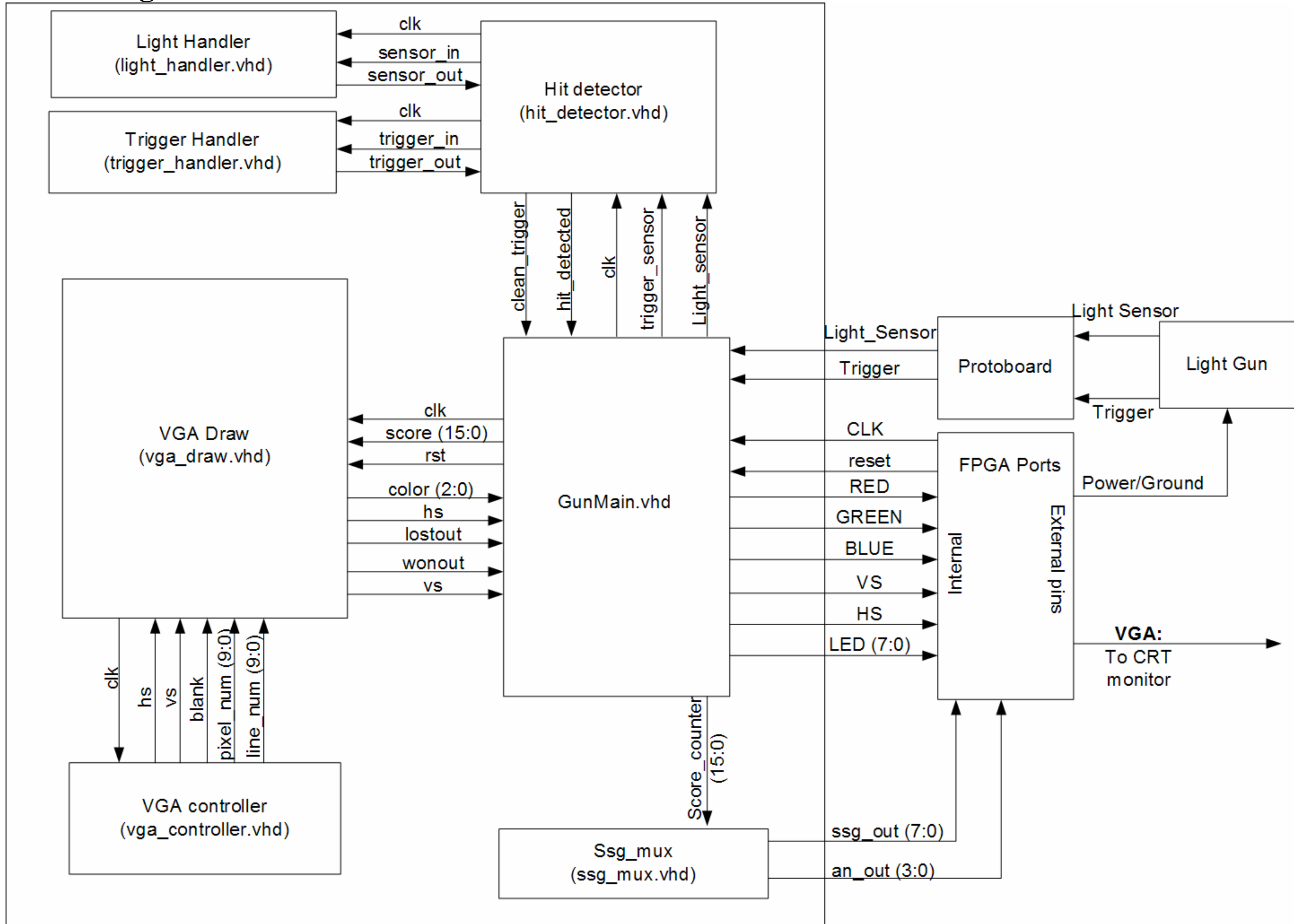

Figure 1 – Block Diagram

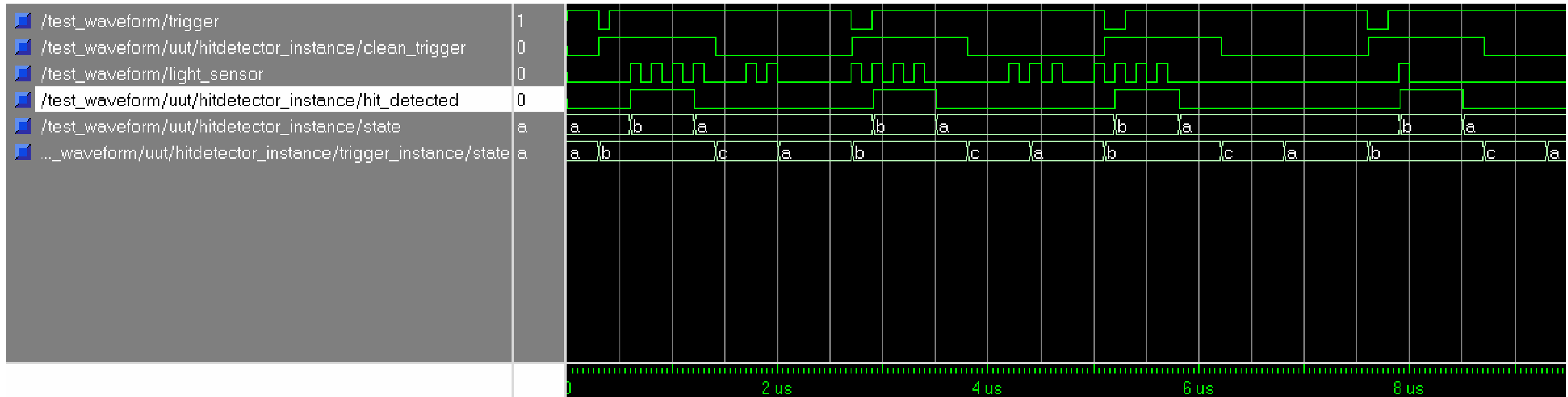

Figure 2 – Simulation waveform that illustrates how a hit is detected. Note that the time is not to scale (the free ModelSim won't simulate for milliseconds), but the ratio of signal lengths is the same. The process of hit detection proceeds as follows: The trigger on the gun ("trigger" above) is pressed (active low). This signal is adjusted with the "trigger_handler" and the fixed signal ("clean_trigger" above) is output high for 160 ms. If the light sensor on the gun picks up light while the "clean_trigger" is high, the "hit_detected" signal is raised for 80 ms. The states for both the trigger_handler ("/trigger_instance/state") and the hit_detector ("/hitdetector_instance/state") are shown above as well.

Project Description/Purpose:

The goal of our project was to create a game similar to the old Nintendo DuckHunt game using the Nintendo Zapper gun, the XilinX Spartan 3 FPGA, and a monitor. The game has a target on the screen that moves around the screen and can be "shot" with the Zapper (light) gun. There is a timer bar that is displayed along the bottom of the screen that starts in the middle of the screen and decrements about every two seconds. When the player shoots the target correctly with the gun, the timer bar increments by one segment. The goal of the game is to make the timer bar progress until it reaches the right side of the screen, which is when the player has effectively "beat the clock" and wins the game. However, if the timer bar decrements completely and reaches the left side of the screen the player loses the game.

Nintendo Zapper Gun Details:

The most complex portion of this project was determining how exactly the Nintendo Zapper gun worked and how to correctly interface it with our game design on the FPGA. Going into the project all we knew about the gun was that in the original Duck Hunt game pressing the trigger causes the screen to go black and replaces the duck with a white square. We knew that the gun "saw" the white square and that is how it determined whether there was a hit (see Figure 7).

We were able to find little documentation about the gun, but did discover that the Nintendo controllers had specific Vcc (+5V) and ground (0V) ports that needed to be connected. This left 5 other pins on the gun for us to test. After not getting any response from any of the pins on the gun, we tried attaching pull-up resistors to the pins. We brought the gun into the lab and displayed values on the oscilloscope, from which we determined that the gun uses two pins to output the gun signals (see Figure 3). For this to work correctly we needed a 10 kΩ resistor connected to each input (see Figure 13).

The "Trigger" signal is an active low signal that is normally at 5V until the trigger on the gun is pressed. After the trigger is pressed, the trigger signal stays low for about 20 ms (see Figure 4). The other signal is the "Light Sensor" signal. The light sensor in the gun emits pulses from low (0V) to high (5V) that last about 1ms and have 20ms between each pulse (see Figure 5).

During our original testing of the light gun, we tried to point it at the LCD monitors in the lab and discovered that nothing registered. We soon discovered that this was due to the difference in the way an LCD and CRT behave. An LCD only emits light, while a CRT has shoots electrons at the screen and has a more noticeable refresh rate. When testing the gun in the lab, we noticed that aiming the gun at the oscilloscope would actually make the gun register light. Knowing this information and also knowing that the Nintendo Zapper gun was designed to work on a TV screen, we then decided to try to use a CRT monitor instead of an LCD screen.

Getting the gun to register images on the CRT monitor turned out to be challenging as well. We initially displayed a black screen with a white box in the center and connected a wire from the light gun to the oscilloscope. The idea was that when we pointed the gun at the white box, we expected to see the pulses generated by the gun being displayed on the oscilloscope. However, this did not happen. The next step we did

was try a moving image on the screen because we thought that perhaps motion would cause enough change for the gun to register the white box. This did not work either.

Our next step was to connect the CRT monitor to a computer, so that we could draw different shapes and sizes of images as well as varying refresh rates, with which to test the light gun. We started with a black background and drew many different sizes of white shapes on the screen (see Figure 8). We then pointed the gun and watched the oscilloscope for pulses. None of the all-white shapes we tried registered any activity, so we decided to draw different types of shapes. One of the shapes that we were able to get the light gun to "see" was a rectangle consisting of alternating white and black rectangles inside of each other (see Figure 9). Our thought was that because our hand holding the gun was not very steady, the movement was enough to cause the light sensor in the gun to move between the dark areas and the light areas. This led us to try another test in which we drew thin horizontal white bars across the screen (see Figure 10). The light gun did detect light when we pointed it at the area with the white horizontal lines, which led us to further believe that the unsteadiness of our hand was somehow causing this.

We also tested pointing the light gun at a TV screen and the fluorescent lights in the lab. Doing both of these things caused pulses on the oscilloscope indicating that the gun was actually "seeing" the light. Given the results of the horizontal lines test, we then considered what the fundamental differences between a television setup, which did consistently work, and our CRT monitor setup, which did not consistently work were. It turns out that the key to the problem was the NTSC and interlacing standard of the television screen. The NTSC specification draws alternating odd and even lines to make a full frame. Drawing alternating odd and even lines at 60 Hz means that any given pixel is drawn at about 30 Hz. However, the CRT monitor we were doing testing on was drawing to the VGA at 60 Hz. This was too fast for the Nintendo Zapper, which was designed for an NTSC television screen. The light gun "sees" objects by responding to a change from little or no light to bright light. To solve this problem, we essentially "interlaced" the target by drawing every other line during each draw of the screen. Once we did this it corrected the issues we had with the gun not registering the target. We also managed to use the interlacing and the gun not "seeing" our normal 60 Hz screen to our advantage. Instead of blacking out the screen and only drawing the target like the original Duck Hunt game does, we just interlace the target and draw the rest of the screen with progressive scan. The gun does not pick up anything but the target because it is the only thing interlaced on the screen. It also adds a feature to the game. In the original Duck Hunt the screen flashing to black when the trigger is pressed is very noticeable. However, in our version we have removed that flash, so the player has a better gaming experience.

Component Descriptions:

VHDL file/component structure:

GunMain (gunmain.vhd)
- Hit Detector (hitdetector.vhd)
  - Light Handler (lighthandler.vhd)
  - Trigger Handler (triggerhandler.vhd)
- SSG Mux (ssgmux.vhd)
- VGA Draw (vgadraw.vhd)
  - VGA Controller (VGA_controller.vhd)

GunMain (gunmain.vhd) – This is the main entity for the project and serves as the entity that links all the subcomponents. It takes the inputs for the trigger and light sensor signals from the light gun (Trigger and Light_Sensor) and has the pin connections associated with it to use the VGA (RED, GREEN, BLUE, VS, HS). It also connects the LEDs so that they all light up when a "hit" has been made and sends the score to the ssg_mux, so the score shows up on the HEX display.

We decided to make our version of "Duck Hunt" playable in a slightly different way than the original version because we thought it would be more fun. Our game has a status bar that starts in the middle of the screen and gives you an initial score of 10 (0x0A). The status bar then decrements about every 2.5 seconds and takes a point off of your score. When you make a hit, your score increments by one and the status bar moves to the right on the screen. The goal is to get the status bar all the way to the rightmost part of the screen.

This GunMain section of code keeps track of the scoring through the "score_count" process. Resetting, losing, or winning are all asynchronous and allow support of replaying the game or staying in the win state or lose state until reset has been pressed. This allows us to show a green screen when you win (and keep the score constant until the reset button has been pressed) and to show a red screen (and keep the score constant until the reset button has been pressed).

The "hit_holder" process allows us to correctly increment the score counter and prevent multiple "hits" from being counted. This is necessary because the signal coming from the "hit_detector" is held high for 160 ms. This was initially done to support changing the screen to black because about 160 ms was needed to get the gun to respond. However we later were able to make it so we didn't have to black out the screen at all, when we discovered how to interlace the target only (see "Nintendo Zapper Description"). This process monitors the "intern_hit" signal, which is the output from the hit detector. One thing that we discovered was that even with this cleaned-up signal from the trigger we would still get fluctuations in the signal and increment the score multiple times. This was also due to the fact that if the gun trigger is partially pressed it will hold the trigger signal low for as long as it is held rather than the normal 20ms. This process rectifies that problem by monitoring that cleaned up signal and adding an extra buffer of time after a single hit has been detected where we don't count any more hits.

The hit2assign process then takes the result from the "hit_holder" process and uses it to create a hit2 signal that is asserted long enough to increment the score_count correctly. All of these timings were very sensitive and it took a lot of time to get it exactly right without access to an oscilloscope or be able to simulate the gun exactly with the ModelSim tools. We initially simulated our timing with the simulator, but discovered that the combination of using the gun, monitor, and FPGA led to inconsistencies that could not be accounted for in the simulation. We ended up starting with theoretical values that should have worked and then adjusting them until they did.

The "timeoutholder" process is used to provide a signal to the score_counter to decrement the score about every 2.5 seconds. Again, this was something that we had to test considerably to get good timing. Synchronizing the score_counter so that it would increment and decrement at precisely the right times was a fragile process and somewhat difficult to do in VHDL because of the "bad synchronous description" limitations.

Hit Detector (hitdetector.vhd) – This is the entity that takes the signals from both the light_handler and the trigger_hanlder and determines if an actual hit has occurred. It consists of a two-state finite state machine. State A is a wait state where it waits and checks to see if both the output from the trigger handler and the light handler are high. If they are then it means that both the trigger has been pressed and the gun is "seeing" light, so a hit should be registered. This is done by moving to state B and asserting the hit_detected signal for about 80ms (see Figure 12). Again, the timing of signals was very important and 80ms was initially chosen for when we were making the screen go black. When we made the game play with the screen going black, the first 80ms the screen was entirely black to make sure the light detector would not detect any extraneous noise and the next 80ms were when a target was displayed. The hit detector was then supposed to output a high signal for 80ms, so only one hit would register for each time the white target on a black screen was displayed. We ended up keeping the signal the same even after we changed to the interlacing of the target because we found that it consistently worked.

Light Handler (light_handler.vhd) – The light handler was initially created so that we could do processing on the light sensor signal (see Figure 5), but it turned out that we just needed to handle the oscillating light sensor in the main file. We decided to leave it in the code in case we decided at a later point to do some sort of processing on it.

Trigger Handler (trigger_handler.vhd) – The trigger handler processes the incoming signal from the gun so that we can have a useable signal. The raw trigger signal is active low (drops to 0V when the trigger is pressed) and lasts about 20ms (see Figure 6). We wanted an inverted trigger signal that would be held for 160ms so that we could use it for blacking out the screen. We also wanted to limit how quickly the user could press the trigger and have it register with our game. We didn't want the player to be able to continually press the trigger extremely fast and beat the game in an instant, so we use this entity to limit how often we will register that the trigger has been pressed. We initially had this set to 1 second, but found that our game was not beatable. (The status bar decrements every 2.5 seconds and so 1 hit registered a second required a perfect hit every second to win). We eventually changed this limit to be about 0.33 seconds because it seemed like playable timing.

The trigger_handler starts in state A, where it waits until the trigger on the gun has been pressed (Trigger_in = '0'). When the trigger is pressed we immediately move to state B, where the "Trigger_out" signal is asserted for 80ms. After 80ms, we move to state C, where we do nothing for about .33 seconds. This prevents another trigger press from being registered until a total of about .41 seconds after the trigger has initially been pressed.

SSG MUX (ssgmux.vhd) – The SSG MUX enables us to display signals from the register unit on the hex displays. In this lab we used our SSG MUX to display the current score. Inside the SSG MUX, we created a 10 bit counter, from which we use the top 2 bits to select whether we are displaying the input signals A (3:0), B (3:0), C (3:0), or D (3:0) by sending the correct an_out value. This lets us slow down how often we are writing to hex displays. If we just tried to do it on every clock pulse, it would not have

time to display. We can only actually write one display at a time (out of four), but we are still doing it quickly enough that it is not visible. With the clock division we are doing, each display is enabled for (2^8 cycles)/(50 MHz) = 5.12 microseconds and the counter starts over every (2^10 cycles)/(50 MHz) = 20.5 microseconds. Also, if the reset button is pressed, the counter will be reset to 0. In this code we also have a conversion from a 4 bit number (that comes in on signal A,B,C or D) to the corresponding 7-bit 7 segment display values. The top bit (7) of the ssg_out signal that is output to the displays is the decimal point signal (1 means show the decimal point, 0 means don't show it).

VGA Draw (vgadraw.vhd) – This is the VHDL code that handles drawing our background to the screen. It has one process (haslost) to determine if we are in the "won" or "lost" state, in which we draw a red or green screen, respectively. It also has many signals that we declared to allow us to draw the status bar, the dithered lake and grass, the dithered sunset, and the dithered blue sky. These values are positions of where we draw all those objects on the 640x480 screen. The "haslost" process checks to see if the user has lost the game and if so asserts the "lost" signal. The "moveball" process handles the bouncing of our target around the screen by checking the horizontal and vertical coordinates of the target and inverting them. We made the horizontal limit align with the middle of the lake so that it looks like our "duck" is flying in and out of the lake. Also, the "InterlaceToggler" process creates the signal that we use in "ComplexDraw" to do interlacing.

The "ComplexDraw" process handles the actual drawing of our dithered image to the screen, our interlaced target, the green screen and the red screen. The highest priority events to draw are a win or a lose and so those are the first "if" statements. The next highest priority is the interlaced target. Interlacing is accomplished by checking if "interlace_toggle" is a 1 or a 0. Interlace_toggle is set in another process by dividing the clock to the correct 25 MHz pixel clock and then inverting the "interlace_toggle" signal if we are drawing a new screen. This makes interlace toggle alternate between 1 and 0 every time an entire screen is drawn. Then in the actual drawing process we compare the least significant bit of the line number and interlace_toggle to draw odd lines white on one entire screen draw and draw even lines white on the next screen draw. This effectively gives an instantaneous image like the one in Figure 9 for the target, but because it is done so quickly is not apparent to the human eye. This effectively simulates NTSC and allows the gun to register light properly.

The status bar is another thing we draw to the screen and the size is calculated by multiplying the score by a constant value ($1/20^{th}$ of the screen width). We made the status bar white so that it would be distinguishable from the other images on the screen.

The rest of the process draws the screen normally (non-interlaced) and also checks to see if it is supposed to draw the lake, grass, sunset, or sky. To make it appear as if we had more than 8 colors, we implemented dithering. Dithering alternates pixels of several different colors on alternating lines in order to make it look like there is a new color. We used this technique on the grass, lake, sunset, and sky to make them appear more natural and less bright. The lake is dithered between blue and cyan, the grass is dithered with black and green, the sun is dithered with yellow and red, the sunset is dithered with purple and red, and the top of the sky is dithered with blue and black.

VGA Controller (VGA_controller.vhd) – This entity controls the VGA by drawing each pixel with a 25 MHz pixel clock and outputting horizontal sync (hs), vertical sync (vs), blank, pixel_num, and line_num signals. We used the provided VGA controller and did not modify it. The controller uses the standard VGA specs found at *http://www.epanorama.net/documents/pc/vga_timing.html*. It uses a 25 MHz pixel clock to draw a total of 800x525 pixels. Only 640x480 of these pixels are displayed though because the other pixels occur in the "porches" and are needed to allow the electron beam to shift to the next row.

Post-lab Comments:

One component that we tried to add to our game was the ability to load a background image. In our original project proposal we suggested loading an image of a duck from SRAM, but we underestimated the feasibility of this task with the tools provided to us and found that a statically drawn duck produced better results.

The idea was to use the serial port to send an image from the computer to the FPGA SRAM and have the VGA drawing code pull each pixel color from the SRAM as it was drawing each pixel. We decided that we would create an image in 8-bit bitmap (bmp) format. We created an image in Adobe Photoshop using a palette of 8 colors (the same as the 8 colors available on the Spartan 3 FPGA) and encoded it into the 8-bit bitmap format.

The next step we took was to write the VHDL code to send the image over the serial port and save it in the SRAM. This was a very complex task because it consisted of syncing the data being sent from the computer with the data being received on the FPGA and loading it correctly into the SRAM.

To write a correct serial driver for the FPGA, we had to look up the serial port communication specifications. This is the RS-232 specification, which allows data to be transmitted by sending a start bit followed by the data and then terminating that frame of data with a stop bit. There is also the availability of a parity bit. Furthermore, there also needs to be a specific baud rate that the data is sent at. The difficulty with the baud rate was dividing the 50 MHz clock on the FPGA into a rate that would match the baud rate. This turned out to be difficult to do because the clock could not be divided well into the available baud rates (9600, 38400, 76800, 115200). Also, we did not want to lower the baud rate too much or it would take too long to send an image to use for the background. (The image was about 300 KB, which takes about 7 minutes at 9600 baud).

We tested sending data by running gtkterm in Linux and sending ASCII characters over the serial port to be displayed on the LEDs on the FPGA. We initially tried a baud rate of 115200, but this seemed too fast when we tested sending data. When the baud rate was high, we seemed to be getting framing errors because the LEDs would not light up in a consistent pattern when we pressed the same key multiple times. We looked up the ASCII codes and checked to make sure the correct ASCII code was being displayed when we sent it over the serial port. After testing all of the available baud rates with this method, we determined that the most reliable baud rate was 9600. This was most likely due to the fact that we were able to divide the clock closest to 9600. When we output our divided clock and measured it with the oscilloscope, the frequency was around 9550 Hz. Unfortunately this was as close as we could get the frequencies to match, but it seemed to be within the smallest margin of error of all the frequencies we

tested. (We later discovered that the errors at 115200 baud were due to the PC hardware we were using, rather than a limitation of the FPGA.)

The next step after we were able to send data over the serial port was to load it into SRAM. We basically wrote a memory controller that loaded the data onto the SRAM data bus and incremented the address consecutively as each byte of data was received over the serial port. The SRAM writes in 10ns, so we knew that there would be no problem in not being able to write the data quickly enough since we were only receiving the data at 9600 baud. We enabled both chips of SRAM (256K x 16 each) on the FPGA, but only used the lower byte of each 16 bit word because the image was encoded in 8-bit format. Our memory writing code consists of a state machine that waits for an entire byte of data from the serial port and then writes that to the SRAM. Then the address increments and we wait for the next byte. This continues until all the data has been sent. 640x480 images encode in 8-bit bitmap format were approximately 300KB and took about 7 minutes to send over the serial port and load into SRAM.

After loading the SRAM with the image, the next step was to display the image on the monitor. To do this we wrote VHDL code that accessed the SRAM to get the color as it was drawing each pixel. The 8-bit bitmap format we used encoded the image with a header followed by a byte of data for each color in which it only used 3 of the bits. This is because our FPGA only supports 3-bit color (8 total colors). In our VGA drawing code we had a pixel counter that was drawing each pixel at a rate of 25 MHz. Starting from position (0,0) on the screen, we kept an address counter and incremented it each time we loaded data from the SRAM. Each time the pixel number changed we loaded the lower 3-bits of the data in SRAM into our "color" signal which went out to the monitor. The SRAM access time is 10ns, so there was plenty of time for the data to be accessed during each pixel draw. Each time we drew an entire screen (when the position was back at (0,0)), we reset the address back to the starting address. Another feature we added was an "offset" that allowed us to start at a certain address in SRAM in order to skip the header information. We also made sure not to fetch any data or increment the address counter during the "front porch" and "back porch" sections of the screen drawing because nothing is actually displayed on the screen during these times.

When we initially tried loading an image over the serial port and displaying it on the screen we saw a bunch of flashing colors and garbage. After debugging for a while we discovered that it was due to the fact that we weren't resetting the address counter back to the initial value when we reached the (0,0) pixel of the screen. Once we corrected this we managed to get a recognizable image, but one that was shifted by several pixels every few rows. Our initial thought was that we were still displaying header information and not actually skipping it, so we opened our image in a hex editor and calculated the exact length of the header. However, this did not solve our problem and the image was still skewed. We then connected the "offset" signal to the switches to allow us to manually adjust how much header information to skip. After going through all 256 combinations we discovered that all this was doing was shifting the entire skewed image back and forth across the screen and not actually solving any problems.

Even after 40 more hours of testing our image displaying code, we were unable to draw the image completely correctly. We continued to try various combinations of calculating the address of SRAM data to load and tried loading data into different locations of SRAM as well. Unfortunately, we were unable to tell exactly if the problem was due to not sending the image correctly over the serial port, not loading it into the SRAM exactly correctly, not reading it correctly, or not displaying it correctly. Without

being able to actually look at the data in SRAM, we were not able to verify that things were working entirely correctly. However, loading the image was not the integral part of our project and was only something that we thought of to do as an addition to the project.

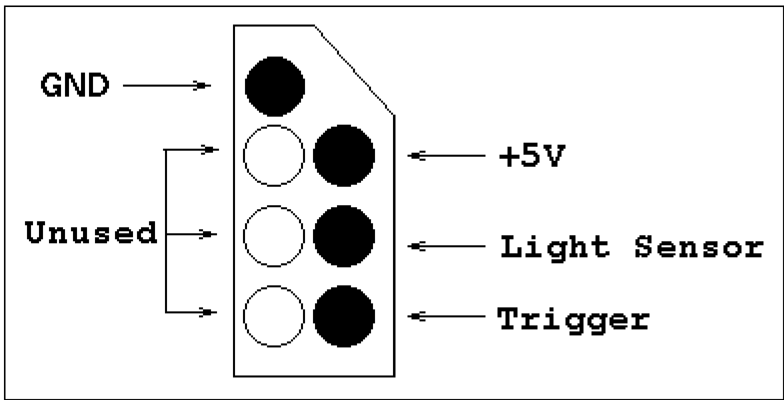

Figure 3 – Diagram of Nintendo Zapper Connector

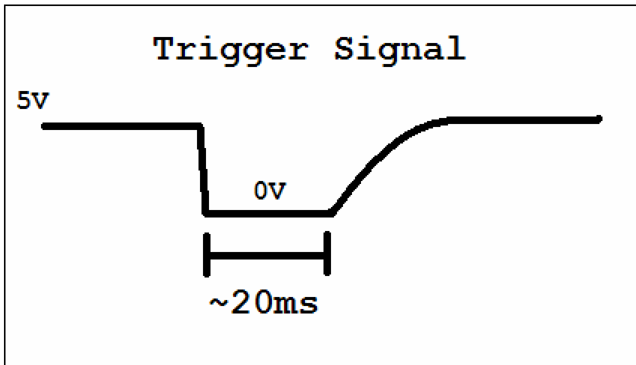

Figure 4 – Signal from Nintendo Zapper after trigger has been pressed (before processing)

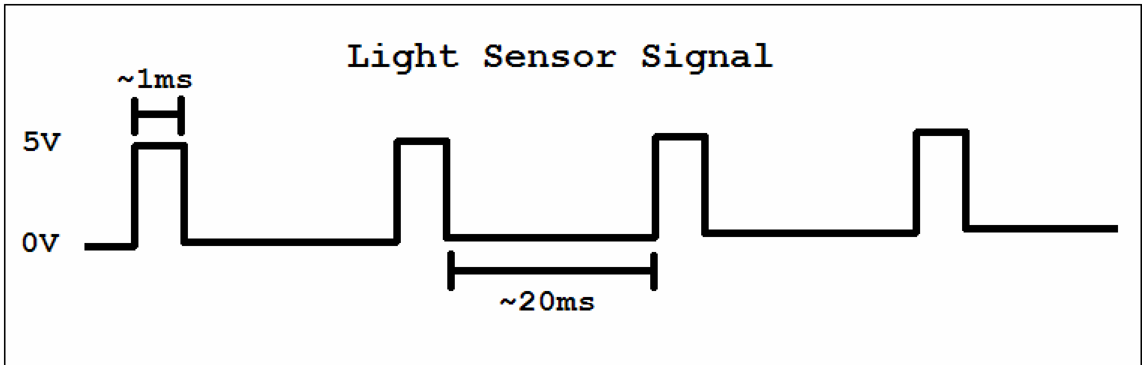

Figure 5 – Signal from Nintendo Zapper when the light sensor "sees" light

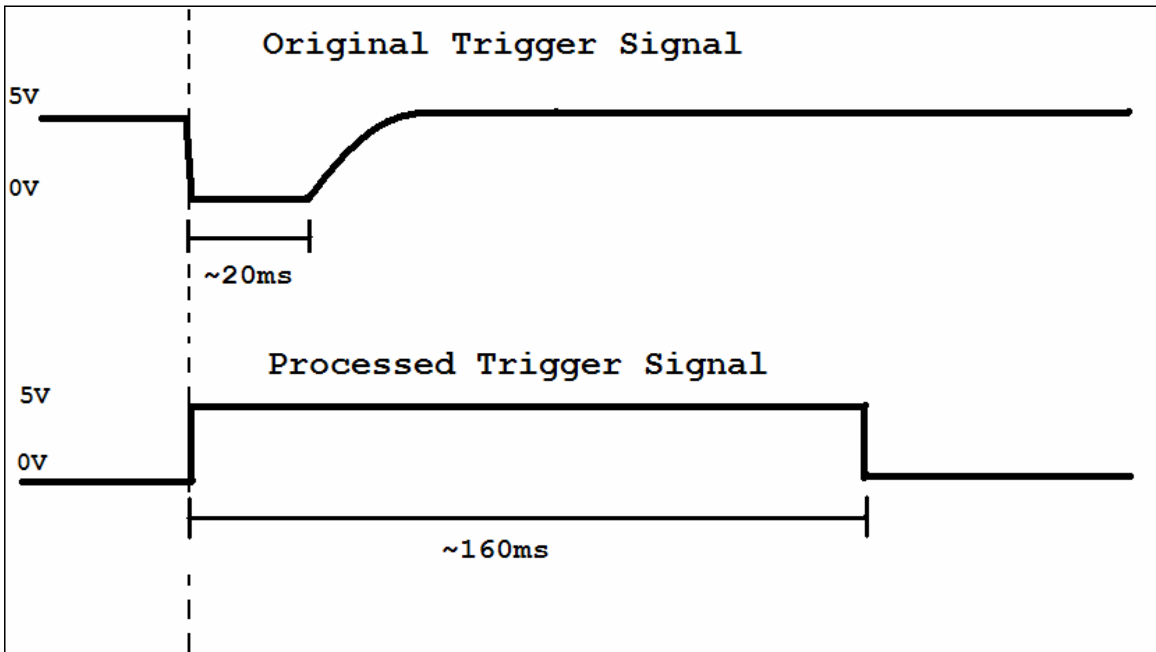

Figure 6 – Diagram of processed Nintendo Zapper trigger signal compared to actual signal coming form the gun

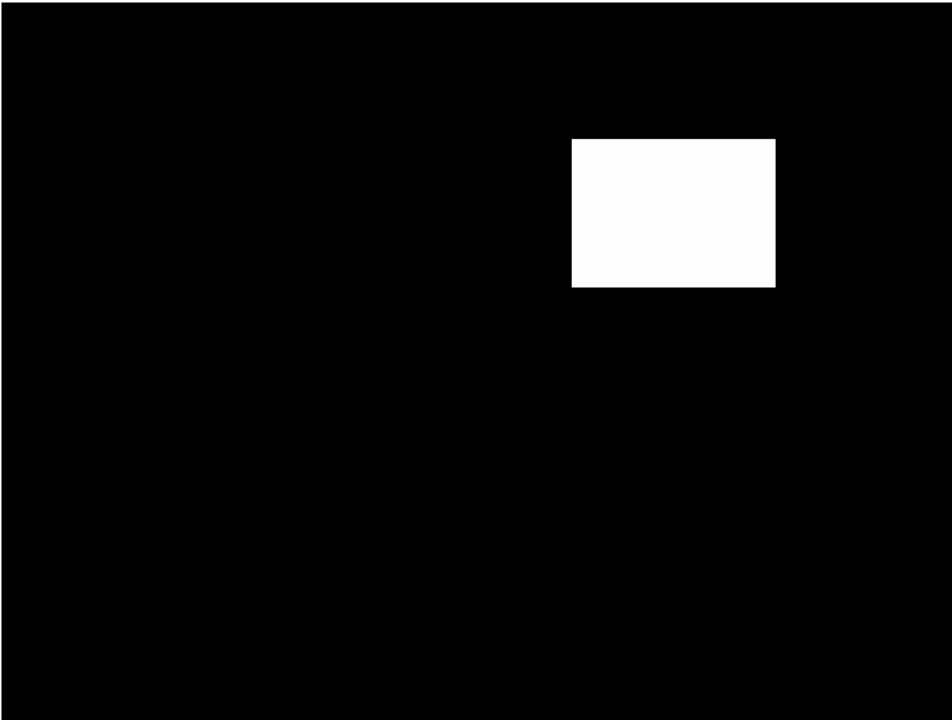

Figure 7 – Screenshot of a normal Duck Hunt screen after the trigger is pressed

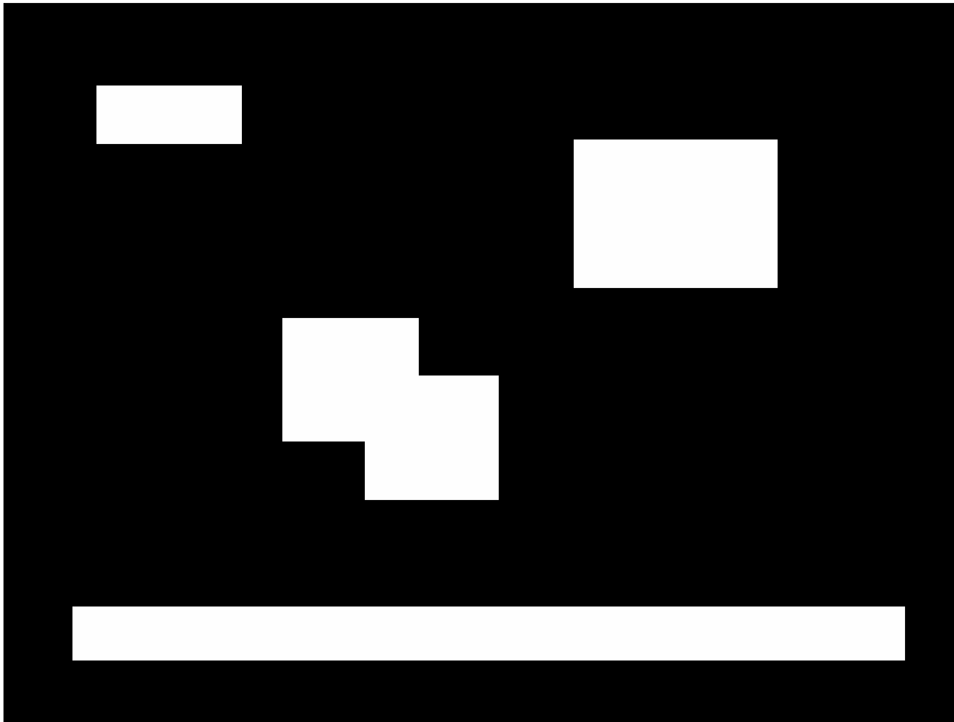

Figure 8 – Example of different sizes and shapes of targets that the Zapper did not "see"

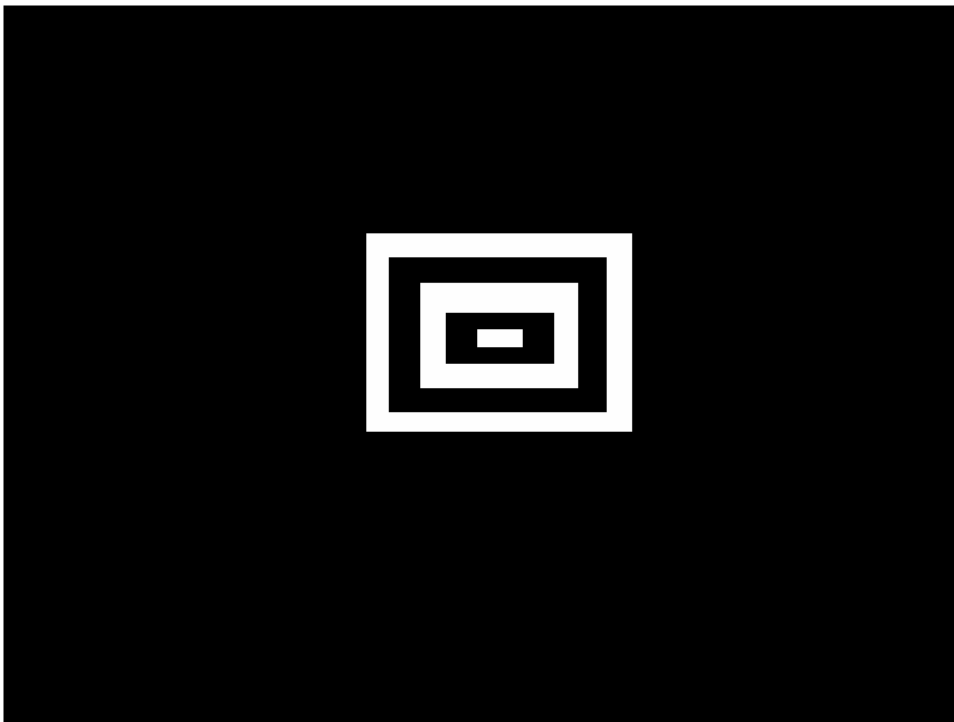

Figure 9 – Screenshot of a target that the Nintendo Zapper registered

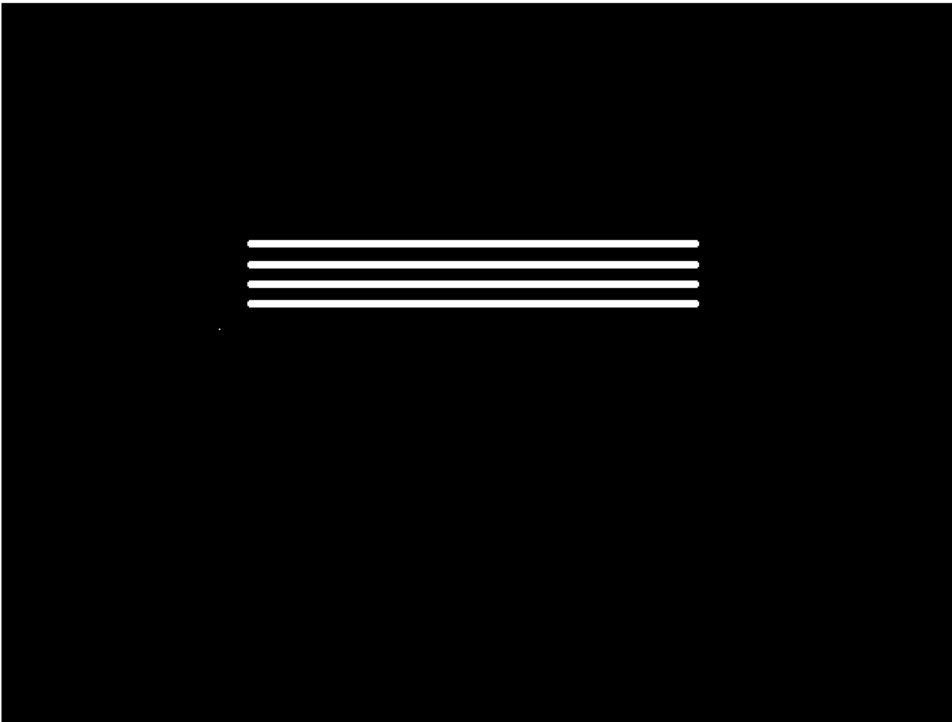

Figure 10 – Example of horizontal lines that the Zapper gun could "see"

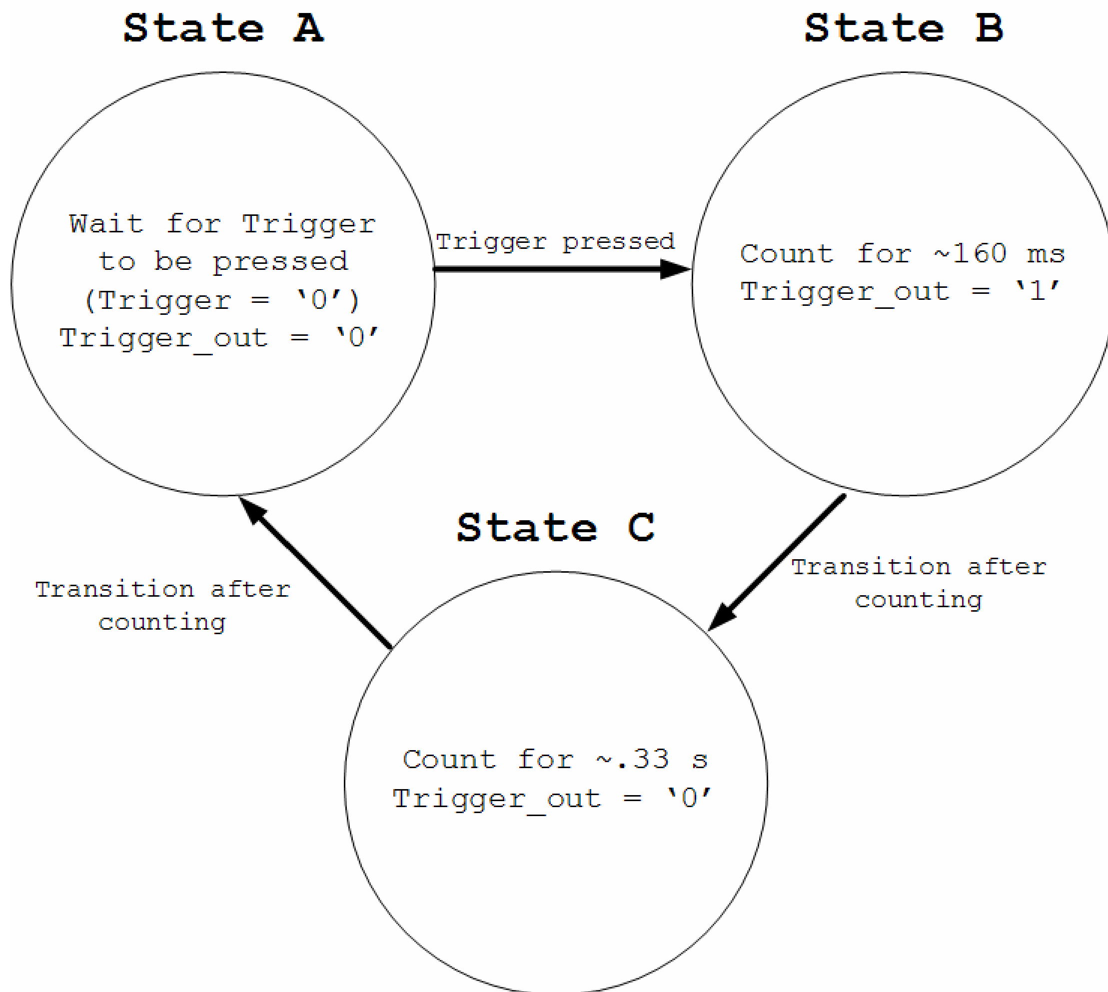

Figure 11 – Finite State Machine for Trigger Handler (trigger_handler.vhd)

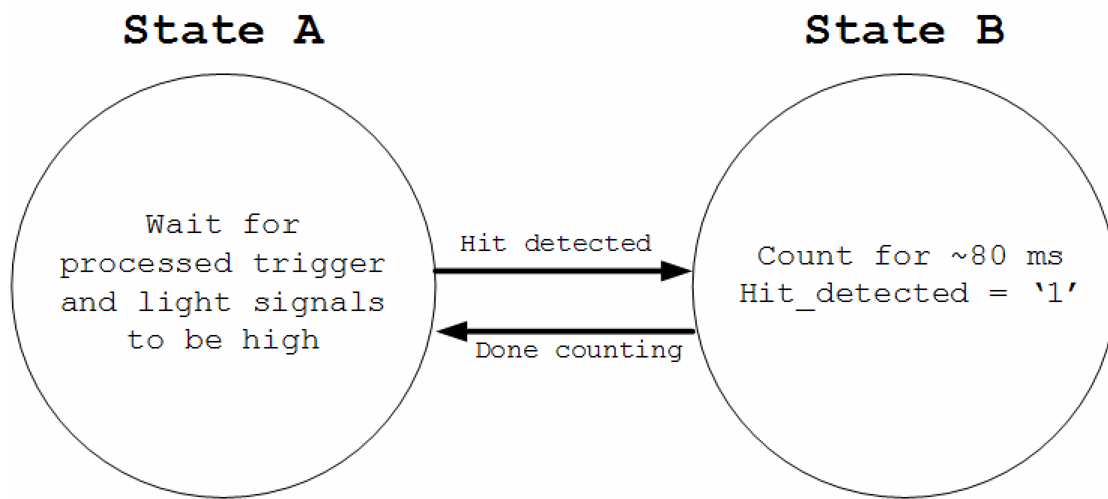

Figure 12 – Finite State Machine for detecting "hits" (hit_detector.vhd)

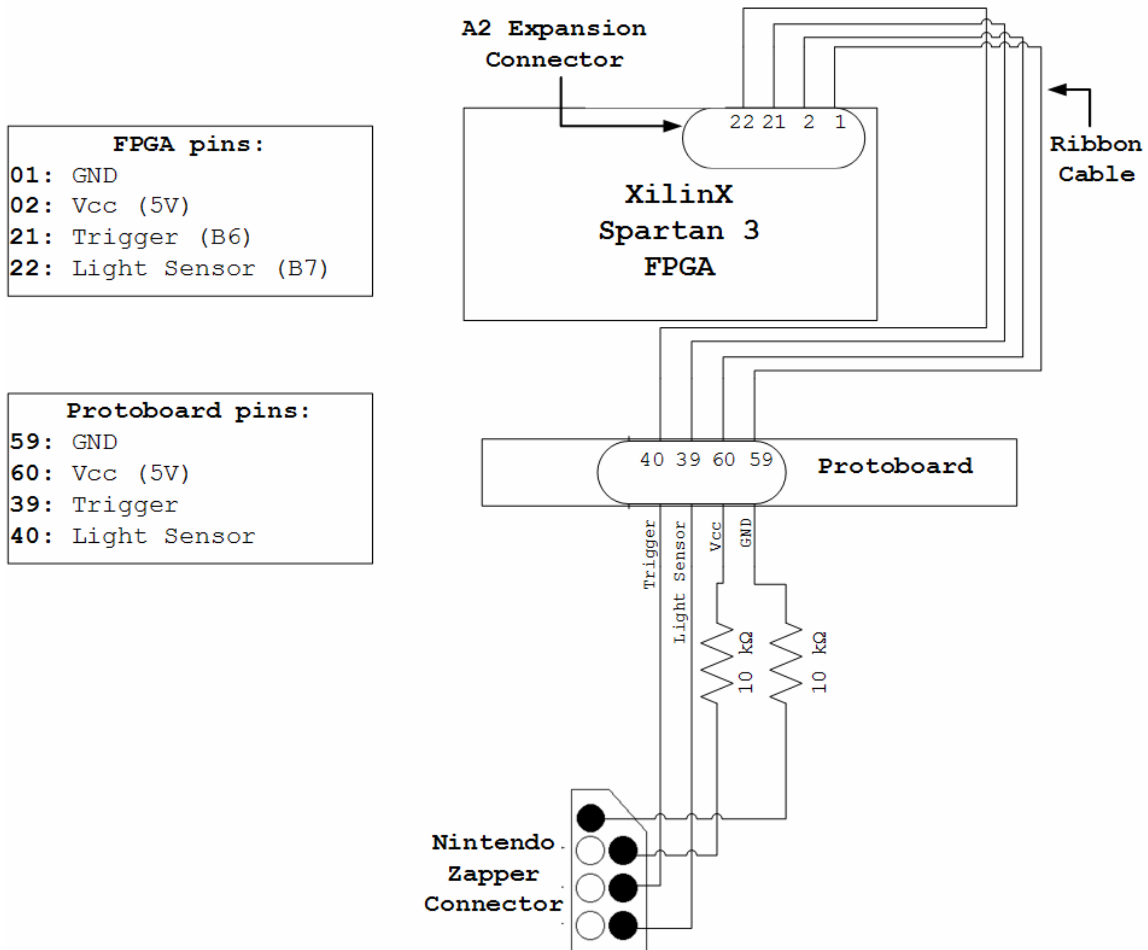

Figure 13 – Circuit Diagram

Port Mappings

Signal Name	Port
an_out<0>	D14
an_out<1>	G14
an_out<2>	F14
an_out<3>	E13
BLUE	R11
Clk	T9
GREEN	T12
HS	R9
LED<0>	K12
LED<1>	P14
LED<2>	L12
LED<3>	N14
LED<4>	P13
LED<5>	N12
LED<6>	P12
LED<7>	P11
Light_Sensor	B7
RED	R12
reset	L14
ssg_out<0>	N16
ssg_out<1>	F13
ssg_out<2>	R16
ssg_out<3>	P15
ssg_out<4>	N15
ssg_out<5>	G13
ssg_out<6>	E14
ssg_out<7>	P16
Trigger	B6
VS	T10

Design Statistics:

TIMING REPORT
Clock Information:

```
-----------------------------------+------------------------+-------+
Clock Signal                       | Clock buffer(FF name)  | Load  |
-----------------------------------+------------------------+-------+
Clk                                | BUFGP                  | 164   |
clkdiv_1:Q                         | NONE                   | 16    |
vga_draw_instance_won:Q            | NONE                   | 6     |
vga_draw_instance_clkdiv_0:Q       | NONE                   | 1     |
vga_draw_instance_clkdiv_18:Q      | NONE                   | 40    |
vga_draw_instance_vga_clkdiv:Q     | NONE                   | 20    |
-----------------------------------+------------------------+-------+
```

Timing Summary:

Speed Grade: -5

Minimum period: 6.840ns (Maximum Frequency: 146.199MHz)
Minimum input arrival time before clock: 4.012ns
Maximum output required time after clock: 14.072ns
Maximum combinational path delay: No path found

Design Summary

```
Number of errors:      0
Number of warnings:    0
Logic Utilization:
  Total Number Slice Registers:       247 out of   3,840    6%
    Number used as Flip Flops:                     241
    Number used as Latches:                          6
  Number of 4 input LUTs:             401 out of   3,840   10%
Logic Distribution:
  Number of occupied Slices:                       309 out of   1,920   16%
    Number of Slices containing only related logic:     309 out of     309  100%
    Number of Slices containing unrelated logic:          0 out of     309    0%
      *See NOTES below for an explanation of the effects of unrelated logic
Total Number 4 input LUTs:            557 out of   3,840   14%
  Number used as logic:               401
  Number used as a route-thru:        156
  Number of bonded IOBs:               29 out of     173   16%
  Number of GCLKs:                      1 out of       8   12%

Total equivalent gate count for design:  5,852
Additional JTAG gate count for IOBs:  1,392
Peak Memory Usage:  71 MB
```