Duckhunter
6.111 Final Project Report

Taylor Barton and Andrew Lisy

December 14, 2005

Abstract

Duck Hunt is a classic first-person shooter game for the Nintendo Entertainment System, in which the player uses a special light gun to "shoot" images of either one or two ducks or clay pigeons, depending on the game mode. Our *Duckhunter* project focused on the third mode of this game: clay shooting. In this mode, two clay pigeons are launched from the bottom of the screen into the distance, and the player is given three shots to hit them both before they are too far away.

In an attempt to beat this game, we have built a servomechanism to automatically track and shoot down the clay pigeons. The labkit's FPGA was programmed to "look" at the video output of the Nintendo System by receiving its video outputs, find the moving targets, and calculate the angle at which to aim the light gun. It outputs the appropriate signals to two servomechanisms to control up-down and left-right aiming, and fire the light gun. In addition, *Duckhunter* has a calibration mode to improve the accuracy of aim.

Contents

List of Figures

List of Tables

1 Overview

*Duck Hunt*is a classic first-person shooter originally released in 1984 for the Nintendo Entertainment System (NES). The game has three modes: one duck, two ducks, and skeet. In the single and dual duck modes, the player aims at sprites of one or two ducks as they fly through the air. In the third mode, the player attempts to hit two clay discs as they move across the screen. In each mode, points are awarded and levels are passed based on the number of successful hits in a given 10-target trial.

Because of the computation problem inherent in predicting flight trajectories of high-order polynomials in real-time, *Duckhunter* focuses on the third mode of play. The project itself is divided into two subsystems: the disc tracking system and the mechanical system. The tracking system, at its most basic, takes in an NTSC signal generated by the NES and outputs appropriate time-based coordinates. The mechanical system receives these coordinates as well as a "fire" signal from the central FSM, and positions and triggers the gun at the appropriate time.

The tracking system is composed of several modules and sub-FSMs used to convert the NTSC signal and track the disc. An incoming NTSC signal is first passed to the NTSC-VGA conversion system which handles the logic necessary to display a 256x240 NTSC signal on a 1024x768 computer monitor. From the conversion block, individual pixels are passed to the tracking engine which converts a stream of pixels into trajectories for each disc. Once the trajectories are discovered, two points, (x_1, y_1) and (x_2, y_2) for each disc are passed to the interpolation component, which predicts a linear-approximation to the flight path at which to aim the gun.

The mechanical system performs two major duties. The first, calibration, is necessary due to the non-stationary nature of the gun. Since the labkit, TV and gun mechanism may be repositioned from demo-to-demo or session-to-session, a calibration routine is necessary to keep the gun aimed properly. Since a small deviation in aiming caused by movement of the labkit or gun could cause the system to fail, proper aiming is crucial to the success of the system. Once calibration is performed, the gun is ready to accept coordinates from the tracking system. Coordinates are passed as a function of time, $< x(t), y(t) >$, and the gun mechanism converts these cartesian coordinates into a pair of angles, (θ, ϕ) which correspond to the horizontal and vertical orientation of the two servo motors in the system. Once the gun is aimed properly, the the fire signal is asserted from the FSM into the NES.

2 Description of *Duckhunter*

2.1 Input System and Disc Tracking DSP

The input system is the primary interface between the game itself and the "player" created by *Duckhunter*. The labkit receives an analog NTSC signal from the Nintendo corresponding to the *Duck Hunt* gameplay. The first part of the system digitizes this signal and stores it in a dual-port RAM. Since the NTSC signal and the VGA signal are clocked differently, the RAM is used to allow these two separate clocks to act on the same set of data. The data in the RAM is then read on the other end by the `vga_with_ram` module. Individual pixels are pulled out of the RAM by this module and passed to the `trajectory` module where they are converted into a 4-point path. The paths is then returned to the `vga_with_ram` module where it is used to perform a fixed interpolation to predict the new position, $(x + \frac{\Delta x}{\Delta t}, y + \frac{\Delta y}{\Delta t})$.

2.1.1 detection_toplevel

The `detection_toplevel` module is an abstraction layer that allows for the easy integration of the two separate parts of the project: the gun system and the tracking system. By "hiding" the code for the tracking system behind `detection_toplevel,` it encapsulates the individual control signals used to drive the various modules involved in the tracking system and allows implementation with a smaller number of necessary

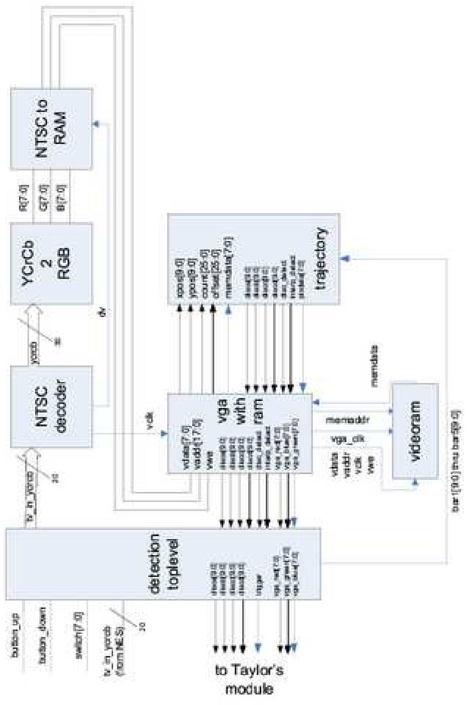


Figure 1: Input System Block Diagram

The input system block diagram. Inputs and outputs are routed to the toplevel file.

signals. `detection_toplevel` handles all the subprocedure calls, interprets the data and hides the private registers and wires. Essentially, it provides a "black box" to the `labkit.v` file, and allows changnes to be made to the internal workings without any changes to the overarching project, provided that there are no abstraction violations or inteface changes. By using this module, we greatly simplied integration, allowed for efficient incremental improvement, and maintained code usability and maintainability.

The inputs used by `detection_toplevel`, aside from the standard input and clock, were synchronized versions of `button_up` `button_down` as well as `tv_in_ycrcb` and `tv_in_line_clock1`. The first two inputs, `button_up` and `button_down` were used directly in the module to control the position of the detection zones for the interpolation. The coordinates, once adjusted, were passed to `trajectory` module for detection purposes and also up to the `labkit` module to be displayed. The tv-in data and clock signals were passed to the `NTSC_decoder` module to be converted into RGB data and used.

The outputs of the module were the `trigger` signal, gun coordinates `x_coord` and `y_coord`, RGB display signals, NTSC display signals for passthru, and `ledbank` to control the LEDs. These were chosen so that as little output from the `trajectory` module as possible could be passed up to `labkit`, to simplify integration. All of the outputs, with the exceptions of `trigger`, `x_coord` and `y_coord` could be directly "plugged in" to the `labkit` interface, thus avoiding many namespace conflicts.

2.1.2 NTSC_decode, adv7185init, i2c

These modules were provided in the collection of "black box" modules for 6.111. The ADV7185 video decoder, integrated in the labkit, decodes the analog stream into a stream of digital LLC data. Modules adv7185 and i2c interface with the decoder chip to provide and generate the necessary clock and data signals. The stream is then passed to the NTSC_decode module, written by Javier Castro. This module generates pixels encoded in `YCrCb` format and appropriate hsync, vsync, and `field` signals. Because the output will be viewed on a VGA monitor, the interlace `field` signal is not used for this project.

2.1.3 YCrCb2RGB

Before the output of the `NTSC_decode` module can be used in the rest of the system, it must be converted from the tv-friendly luminance/chrominance (`YCrCb`) values to a more VGA-compatible `RGB` signal. The conversion is primarily mathematical in nature, so the details will not be presented here. The `YCrCb2RGB` module, provided by Xilinx in their IP Cores library [1], acts between the `NTSC_decode` module and the `NTSC_to_ram` module.

2.1.4 NTSC_to_ram

The first stage of the dual-port RAM hand-off scheme is storing the NTSC pixel data in the RAM. The RAM itself is 18-bit x 8-bit, allowing for 262144 pixels in 8-bit color. The data itself is arranged in a scheme whereby the first 9-bits in the address correspond to the y-axis row, and the next 9-bits indicate the x-axis column. Therefore, given an 18-bit address, the RAM will return or store the RGB color in an 8-bit format, `RRRGGGBB`.

The `NTSC_to_ram` module receives RGB data from the `YCrCb` module. Since the module is set up to allow for up to 24-bits of RGB color, yet our system only uses 8-bit color, only the high-order bits of the R, G, B signals are used. The red and green signals contribute 3 high-order bits, while the blue signal contributes 2 high-order bits. The data is then put into the RAM in sync with the NTSC clock.

[1]See the IP Cores library at http://www.xilinx.com/ipcenter/. The YCrCb2RGB datasheet and module can be found at http://direct.xilinx.com/bvdocs/appnotes/xapp637.pdf

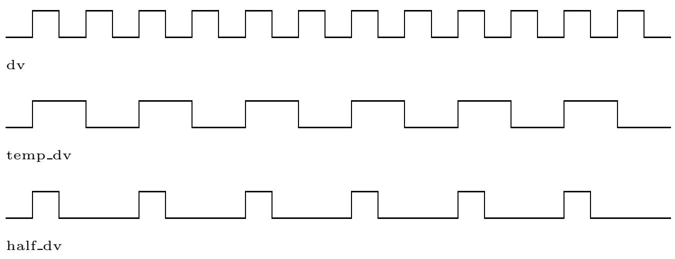


Figure 2: NTSC_to_ram Timing Diagram

The original enable, `dv`, caused the image to appear stretched across the screen. The lengthened signal, `half_dv`, is high every other `dv`, thus storing every other pixel and compressing the image horizontally.

Modifications to Module Since the system was originally set up to display a 256x192 pixel signal, it needed substantial modification to correctly handle the 256x240 signal. The first issue to surface was a problem with the aspect ratio. When the image appeared immediately after the module was modified, it was stretched abnormally in the horizontal plane. This was likely due to the module only sampling the first 192 pixels, then displaying them across the entire screen. To correct this, the `data valid` signal needed to be modified to be high one 'tick' out of every four.

To create this signal, as shown in Figure 2, first a temporary signal, `temp_dv` is used to divide the frequency of `dv` in half. Then, the original `dv` and `temp_dv` are put through an AND gate to obtain the final `half_dv` signal which is asserted every fourth clock transition. Since this new signal is high half the time of the original, this causes the RAM to receive only every other pixel, which compresses the image in the horizontal plane. Although the image is still distorted somewhat (240px has been compressed to 120px), since the entire image is now stored in RAM, the final calculations can compensate for the distortion. This compensation was not possible earlier, since only the first 192 pixels were ever stored.

2.1.5 vga_to_ram

Once the pixels have been successfully stored in the RAM, they must then be read and sent to the processing unit. The `vga_to_ram` module handles the retrieval of pixel data and dispatches the processing unit. To do this, the module first generates a `pixel_clock` signal at 78.780MHz, which is used throughout the module to process data at an appropriate rate for a 1024x768 VGA display. The data itself is retrieved by a pair of incrementing counts, `xpos` and `ypos`, which correspond to the horizontal and vertical position of the current pixel. As previously mentioned, the pixel data is saved in memory at an address formed from the first 9-bits of the `ypos` signal followed by the first 9-bits of the `xpos` signal, in a `{yyyyyyyyyxxxxxxxxx}` scheme. The counts increment every cycle of `pixel_clock` so that a new pixel is retrieved on each positive edge.

The `vga_to_ram` ensures that the `xpos` and `ypos` signals are within the appropriate specification for the resolution. Parameters are defined for the horizontal and vertical front porch, back porch, sync, active, and total which control the incrementing of the `xpos` and `ypos` signals. This portion of the module was provided courtesy of Javier Castro, so little modification was necessary.

Modifications to Module The majority of the modifications to this module involved the dispatch and handling of the data received from the memory. Since only one pixel was sent for a given clock cycle, a collection of FSMs was needed to correctly handle the pixel given the current state of the detection subsystem. Details about the specific calculations are given below in the `trajectory` module section. The `vga_to_ram` acted as the controlling module, choosing how to send data to and interpret data from the `trajectory` module. The submission portion is very straightforward – data coming out of the ram is passed into two instances of the `trajectory` module, corresponding to the two discs.

Along with the pixel data, the module also passes an offset, a count, `xpos`,`ypos`, and detection zones. The specific function of each of these signals in determining a trajectory will be outlined below. The offset and count are used to separate the first disc from the second. A counter is incremented in `vga_with_ram` as soon as the first region becomes active on the first disc. After the specified offset, the `trajectory` module begins to look for the second disc in the first region. The offset delay is necessary because it gives the first disc time to clear the first zone before the system begins looking in that zone for another sequence of white pixels. If no offset were used, the first disc would trigger the second zone as it moved out.

After each of the trajectory instances finishes computing the path of their respective disc, the `disc_detect` signal is asserted to indicate that the values are ready for processing. When `vga_with_ram` receives this signal, it performs a fixed interpolation according to the following equations:

$$x_{\text{interp}} = x_{\text{zone4}} + (x_{\text{zone4}} - x_{\text{zone3}})$$
$$y_{\text{interp}} = y_{\text{zone4}} + (y_{\text{zone4}} - y_{\text{zone3}})$$

The zone values correspond to different regions at different y-values on the screen. The distance between the zones can be adjusted in realtime (calculated in `detection_toplevel` and passed down), thus adjusting the distance used for the interpolation. At first, a more complicated interpolation model involving calculated slopes with multiplication and division IPCores; however, tests with this method proved to be equally accurate at the expense of $s$ome flexibility. Since multiplication and division of sets of 10-bit numbers is very computationally expensive and difficult in a real-time system such as this, sacrificing flexibility for simplicity is a very desirable tradeoff. This proved to be a good decision – the fixed interpolation works very well, and the system is much easier to maintain since the timing issue are lessened.

The final role played by `vga_with_ram` is displaying the region bars, zone indictors and interpolation indicators on the VGA screen. Since *Duckhunter* uses a TV for the actual gameplay, the VGA monitor is free for diagnostics. Five colored bars, repositionable via the 'up' and 'down' buttons on the labkit, indicate the regions where the points of trajectory will be calculated. Immediately after coordinates are registered in each of the four zones, colored boxes are displayed to indicate the trajectory determined for the disc. After a brief delay controlled by the `trajectory` module, a different-colored box appears to coincide with the disc passing through the region covered by that box. The delay 'waits' for the disc to move from the edge of region 4 to the area that the interpolation predicts. When the delay has been satified, the `delay_interp` signal is asserted, causing the `vga_with_ram` module to display the interpolation box and the gun assembly to pull the trigger.

Interpolation Issues While the interpolation scheme works perfectly for typically 9 out of every 10 discs, there are two specific cases that will cause diminished accuracy. The first condition, a disc speed significantly slower than average, will cause the disc to undershoot the mark. If the speed is only slighly under average, the tolerance of the gun will still likely consider the shot a 'hit', even though it was not precisely on target. However, if the disc is fired significantly below speed, it will not reach the point necessary for the interpolation to be successful.

The second issue is the opposite: too much speed. A disc traveling significantly faster than average will be past the target point by the time the gun is aimed and ready to fire. Again, depending on the actual speed of the disc, a hit may still be registered if the offset is within the tolerance of the gun.

Of these two problems, underspeed is more troublesome because of the linear nature of the interpolation. When a disc is traveling faster than expected, it will remain very linear up to the point where the interpolation is made. Therefore, the only concern is the timing issue. On the other hand, a disc moving too slow will

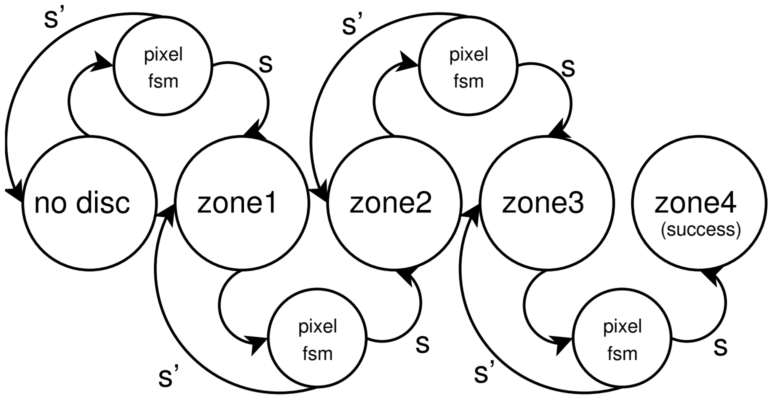


Figure 3: Zones FSM

The zones FSM acts as a filter in determining the path of the disc. Since the system is only looking at a small zone at a time, it is less susceptible to noise and can provide more robust coordinates. The final two zones, zone3 and zone4 are used for the interpolation.

begin to exhibit arc in its path. The deviation from linearity, combined with the timing problems of a slow disc, make this particular situation very difficult to effectively track.

2.1.6 trajectory

The `trajectory` module is the engine of the disc-tracking system, taking in only a single pixel and outputting a set of trajectory coordinates. The system itself works as a series of FSMs (see Figure 4 and Figure 3) which allow for a robust and accurate solution. The detection of a disc is based on a series of consecutive pixels. Once a disc is identified, it is matched up with the current "zone" in which it is located, and a coordinate is asserted based on that zone.

The central FSM to the system is the zones FSM. As Figure 3 indicates, the default 'reset' state of the system is hte 'no disc' state. This occurs when the game is waiting for a disc to be "thrown" across the screen. As soon as the disc is fired, the `trajectory` module immediately recognizes the the first white pixel in the sequence. This causes the FSM to move into the pixel sub-FSM. As Figure 4 shows, the pixel FSM handles the sorting of random white pixels (often caused by the analog-digital conversion) from the series of white pixels indicative of an actual disc. As the the pixel FSM figure shows, the `trajectory` module needs to "see" three consective white pixels before it will reach the final (success) stage. At any point, if the pixel that is expected to be white is not white (indicated by the wp'), then the FSM will reset and wait for the next white pixel to appear.

The pixel FSM was used to ensure that the system was robust enough to guard against random noise in the signal. Initially, it was designed to use only one pixel to detect a disc; however, a large false-positive rate inspired the current system. Although no number of pixel "look-ahead" can **guarantee** that the system will not have false positives (as there is always a small but nonzero probability of getting n pixels in a row), the current scheme with 3 pixels brings this risk down to an acceptable level.

Once the pixel FSM registers a success (indicated on Figure 3 by `s`), the zones FSM will assert the `region` signal to correspond to the last region to "hit". Since this design uses four zones, the `region`

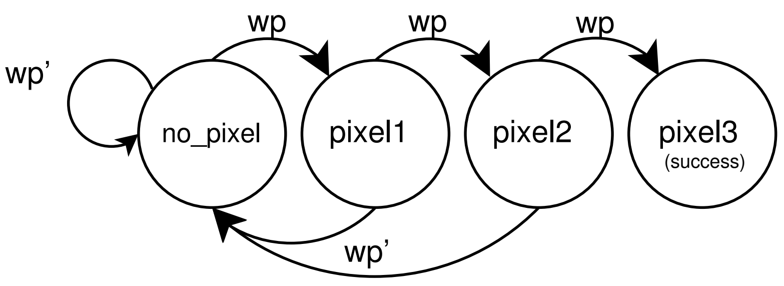


Figure 4: Pixel FSM

The pixel FSM provides noise tolerance in the system. Only 3 consecutive white pixels (wp) will trigger a success. Otherwise, the system resets and waits for the next white pixel.

signal is 2-bits. For example, when the system first recognizes a disc, the lowest zone on the screen will be the first zone to become activated, which will cause a `01` to be placed on the `region` line. The current value on this line indicates to the zones FSM which state it is in, thus activating the correct outputs for the given position of the disc. When the fourth and final region is activated, the `region` signal is reset, and the `disc_detect` signal is asserted. This signal indicates to the parent module, `vga_with_ram`, that the trajectory calculations are complete, and the data lines corresponding to the disc positions in each of the four zones are valid. When the `disc_valid` signal is received, the `vga_with_ram` module calculates the interpolation, adjusts the coordinates to translate the displayed VGA resolution to an appropriate TV screen resolution, and displays the expected position of the gun with a green box. At the same time the `disc_detect` signal goes high, a counter is started. After a delay appropriate to the time needed for a disc to fly from the edge of the fourth zone to the interpolated box passes, the `display_interp` signal is asserted. This indicates to the `vga_with_ram` module that the trigger should be "fired". When this action occurs, two things are done: first, the actual `trigger` is sent up to the `detection_toplevel` to go to the gun system, and secondly, a blue box is put on the VGA screen to indicate that the shot was "fired". The twofold scheme allows for more robust and simpler diagnostics, since there are now two ways to check if the coordinates and signal are correct.

Since the `trajectory` component is modular and pertains to only one disc, it is logical to attempt to generalize it so that two disc functionality can be implemented with little more than another instance of the module. To allow this, two additional signals are present in the module which are set differently for the two-disc case than for the one-disc case. The `count` and `offset` inputs, each 26-bits wide, allows for a timed offset to be specified before "looking" for the second disc. When the two `trajectory` instances are generated in the `vga_with_ram` module, the first one is specified with a null count and a zero offset, meaning that the first disc is detected immediately. The instance for the second disc, however, includes a count signal and an experimentally-determined offset. The count is detemined and incremented in the calling module. Within `trajectory`, the offset causes the system to wait for a specified number of clock ticks after the first disc is detected in zone 1 before registering a "hit" in the first zone for the second disc. The is necessary because without the offset, the first disc would trigger the first zone for the second disc on its way out of the first zone. The offset period allows the first disc to move out of the zone before once again looking for the white pixels in that zone, thus preventing a false trigger. In theory, this scheme should work as planned; however, since most of the design and time was spent ensuring that the trajectory for the first disc was correct and robust, the second disc system is not fully implemented in this version of the code.

The final consideration in the trajectory module is the zones themselves. In designing a noise-resistant system, it is desirable to block out all potential sources of noise that aren't absolutely needed. In *Duckhunter* , the zone scheme allows the `trajectory` module to focus on a small part of the screen at a time. Without zones, the module would be looking at the entire screen when only the lowest part was needed, hence dramatically increasing the probability of picking up an errant set of white pixels and causing a false positive. The zones are set up with both horizontal (designated by vertical bars outside of which the signal is not read) and vertical (designated by stacked horizontal bars) cutoffs. The horizontal cutoffs are fixed, since there is a certain point on each side of the screen from which discs are never fired; therefore, it is safe to ignore past these points. The value for this point was determined experimentally and hard-coded as a parameter in the system.

The vertical zone scheme, on the other hand, is more complex. Since it is advantageous in testing and implementation to have an adjustable "squelch" in the system, each of the 4 vertically-stacked zones can be adjusted. The actual adjustment is done in the `detection_toplevel` parent module, and signals for each of the 5 bars are passed down to `trajectory`. The adjustment itself is done by setting the switches[3:1] to values of 1 through 5 corresponding to the desired bar, and operating the up and down buttons. To make adjustment even faster, the speed of the adjustment can be changed to one of 4 speeds by adjusting switches[7:6].

Different placement of the zones causes different tradeoffs. Since the interpolation is calculated based on the register disc position on the last two zones, these cause the biggest difference in the system when adjusted. To obtain the largest possible time between the edge of zone 4 and the trigger signal (for instance, when it is desirable for the gun to have as much time to "lock on" the new coordinates as possible), the 3rd and 4th zone should be as wide as possible. This causes the interpolation region to be the farthest from the edge of zone 4. However, since the discs follow an arched trajectory, taking coordinates early in the run will result in diminished accuracy. Consequently, if accuracy is the primary concern and the gun can track very rapidly, a higher "hit" yield can be reached by positioning zones 3 and 4 close together near the peak of the disc arc. Just before the arc bends is the area in which the path is most linear, so this method will be more accurate at the expense of prediction time.

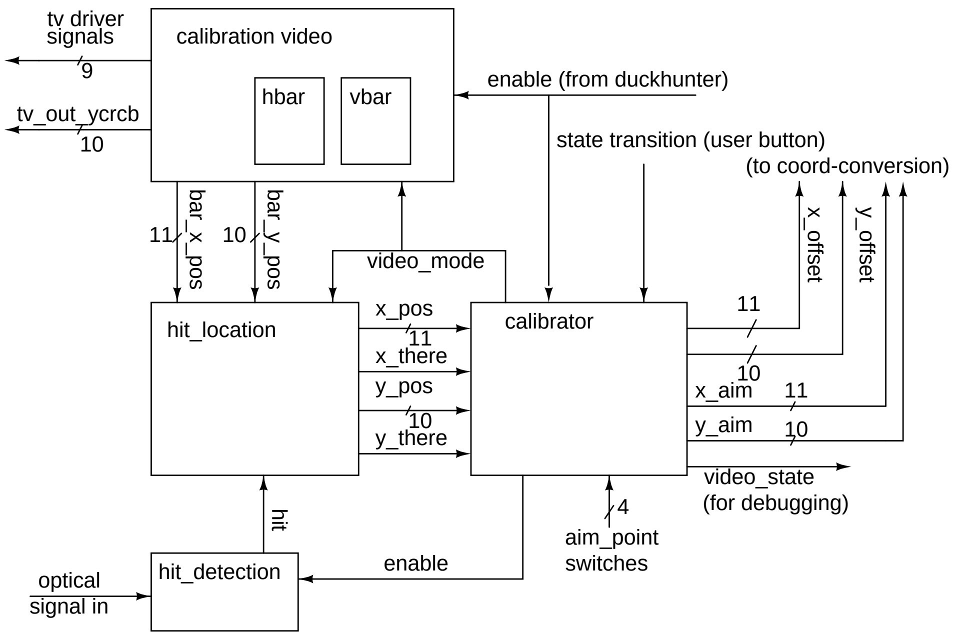
2.2 Calibration

The block diagram for the calibration part of *Duckhunter* is shown in Figure 2.2.

In calibration mode, *Duckhunter* outputs its own video signal to the television. This signal is handled by the `video` module, which has `enable` and `mode` (vertical bar, horizontal bar, both, or colorbars) inputs, and outputs a NTSC video signal and the various clocks and signals required to drive the ADV7194 video encoder.

2.2.1 Calibration Video

This module instantiates three submodules: `adv7194init`, `hbar`, and `vbar`. The module `adv7194init`, written by Nathan Ickes, initiates the video encoder and generates the necessary clocks. The `hbar` and `vbar` modules generate the YCrCb signal for a horizontal and vertical bar, respectively, at an input coordinate. These coordinates are selected by the `video` module, which simply increments the positions by 1 every 27Mhz clock cycle, returning to zero when the bars reach the far edge of the screen. The bar widths are 64 pixels, roughly the size of a *Duck Hunt* target. The `video` module outputs either one or both of the bars or displays colorbars, depending on the `mode` input. In addition, it has the horizontal and vertical positions of the bars as outputs, which are used by the `hit_location` module.



2.2.2 Hit Detection

The `hit_detect` module is also used by the `hit_location` module. This module's inputs are an `enable` signal and the `optical` signal from the light gun. If enabled, when the module detects a positive edge on the `optical` signal, it begins counting up from zero for as long as the `optical` signal remains high. If the count reaches some (hardwired) threshold value, the `hit` output signal is brought high for as long as `optical` is high.

2.2.3 Hit Location

The `hit_location` module uses the bar position outputs from `video` and the `hit` signal from `hit_location` to determine the position of a hit. In addition, it receives the `video_mode` input used by `video` as an input. Based on the video mode, when `hit` is high this module saves the position of the bar to one of two registers, which are also the module's outputs. For example, if `video_mode` is 1, meaning that the `video` module is displaying a horizontally moving vertical bar, and hit is high, then the module stores the bar's x-position to a register. The `hit location` also has two outputs `x_there` and `y_there` that indicate when the x and y positions have been found.

2.2.4 Calibrator FSM

Finally, the `calibrator` module combines all of the various outputs from these modules, and controls the `video_mode` and enable signals.

Based on 4 `aim_point` signals, which are set by the user using the lab kit switches, the `calibrator` module selects a point of aim and points the gun at that point. It sets the `enable` signal for `hit_detection` high and the `video_mode` to 1 (vertical bar), and waits until it receives `x_there` and `x_pos` signals from `hit_location`. As soon as the hit is received, `enable` is brought low and `x_offset` is then calculated by comparing `x_aim` to `x_hit`. Upon receiving a `state transition` signal from the user, the calibrator repeats

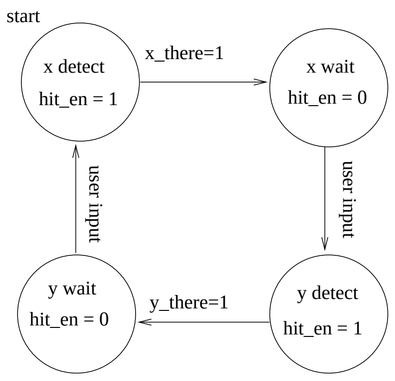


Figure 5: State Transition Diagram for Calibration Module

The calibration module cycles through the states, waiting for user input, until the user is satisfied with the calibration

this process with y-values. The entire process is iterated until the user is satisfied with the calibration. In the demo mode (see Section 4.1.9), the point of aim and hit position are displayed on the XVGA monitor, so that the user can observe the state of the calibration. A state transition diagram describing the operation of this module is shown in Figure 5

2.3 Aiming the Light Gun

The block diagram for the modules that aim and move the servomechanisms is shown in Figure 6. As is described in detail below, the `coord_conversion` module converts from (x, y) coordinates to the signals required for the servomechanisms, and the `rc_motor` modules drive the servomechanisms to their final positions.

2.3.1 Driving the Servomechanisms

The servomechanisms are HS-422 Standard Deluxe Servos made by Hitec. They are pulse-width controlled, with the neutral position set by at pulse width of 1500μs. *Duckhunter*'s servos are driven using the `rc_motor_move` module, which in turn uses `rc_motor` to create the pulses. The input to `rc_motor_move` is an 8 bit signal corresponding to the desired motor angle. For these servos, this value should be between 0x38 and 0xF8 to cover the range of 180 degrees. As its output, the module creates a series of 16 pulses which drives the motor to its final position. The `rc_motor_move` and `rc_motor` modules were written by Issac Chuang.

2.3.2 Coordinate Conversion

The `coord_conversion.v` module converts cartesian coordinates corresponding to a position on the screen into angle coordinates that are used to drive the motors. Its inputs are unsigned x and y target coordinates from Andrew Lisy's interpolation system, signed x and y offsets from the calibration module, and the approximate distance of the light gun from the television, indicated by the user using the switches on the lab kit. Based on these various dimensions, the module derives the appropriate horizontal and vertical angle coordinates for the servomechanism driver.

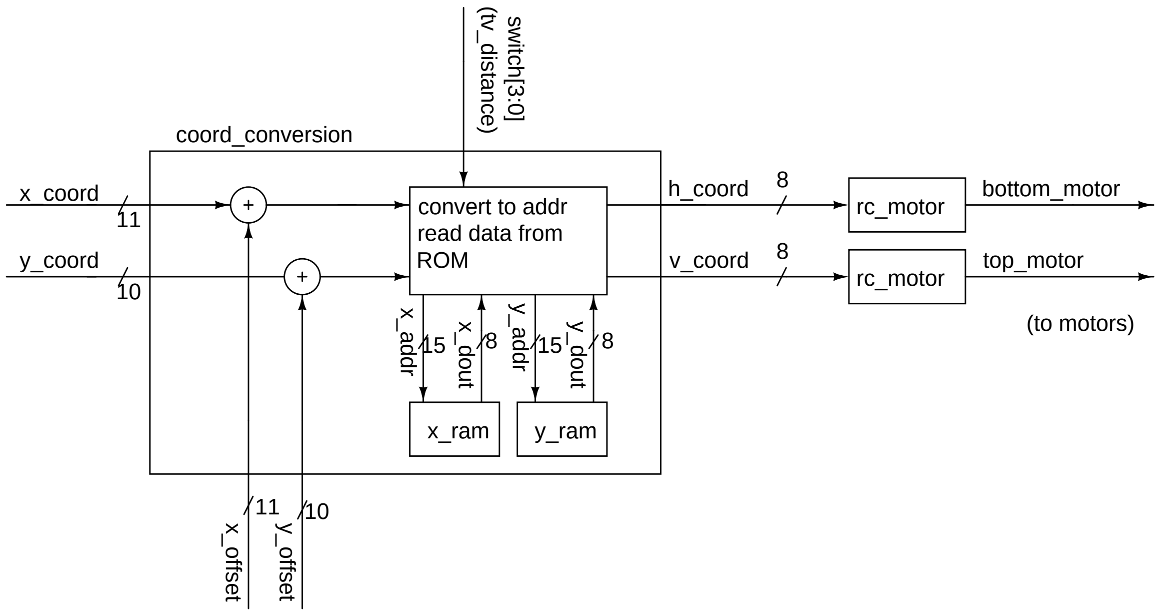


Figure 6: Block Diagram for Aiming and Driving the Servomechanisms

The coord_conversion module converts from cartesian coordinates to angles for the servomechanisms, and rc_motor drives each motor to this angle.

Ultimately, this module uses a lookup table to determine the necessary motor angles. The lookup table, generated by the perl script appended in section 6.2, takes as its "inputs" the x or y position and the distance to the television. That is, the values in the ROM are ordered such that the address corresponds to $\{tv\_distance, position\}$. Figure 7 gives an overview of this addressing scheme.

The *position* value is based on a division of each half-screen into a set of blocks of pixel in order to control the size of the lookup table. *Duckhunter* uses a horizontal division of 16 blocks in the horizontal half-screen. Because of the dimensions of the television, this corresponds to 12 blocks per vertical half-screen to get square blocks. The `coord_conversion` modules first adds the target coordinate and offset values to find the correct x and y coordinates, compares the coordinates to the center of the screen to determine which half-screens they are in, and then uses modular arithmetic to determine which x and y blocks these coordinates fall into. The angle values are then read from the two lookup tables (one for each dimension, so that the x and y conversions can be done simultaneously). Finally, based on which half-planes the original coordinates were in, these values are either added to or subtracted from the neutral position angle values.

An exemplary sketch is shown in Figure 8. This figure shows two example target points. Cartesian coordinate values are shown on the left and top sides, and angle coordinates on the right and bottom.

2.3.3 TV Coordinate Conversion

The TV coordinate conversion module is identical to the conversion module described above, except it is adapted for use with the resolution of the television (720 × 486). Since Verilog does not allow modular arithmetic with numbers that are not powers of two, the only change in the module is that it uses a series of `if-else` statments to determine which block the coordinates lie in.

$$
\begin{array}{l}
\text{distance} = 1 \begin{cases} 0, & \text{block } 0 \\ 11, & \\ 22, & \\ \vdots & \\ 75, & \text{block } 15 \end{cases} \\
\text{distance} = 2 \begin{cases} 0, & \text{block } 0 \\ 5, & \\ 11, & \\ \vdots & \\ 58, & \text{block } 15 \end{cases} \\
\qquad\qquad\quad \vdots
\end{array}
$$

Figure 7: Lookup Table Organization

The coordinate conversion lookup table is organanized so that the address is of the form {tv_distance, position}. There are two of these lookup tables, one for horizontal and one for vertical coordinates.

3 The Light Gun

The controller used to "shoot" the ducks and clay pigeons in *Duck Hunt*is the Nintendo Zapper light gun. In order to control the light gun for electric triggering and calibration, it was first necessary to understand how the gun works.

The most important feature of the light gun is its phototransistor, which creates a current based on the amount of light striking its surface (the base of the transistor). In the Zapper, there is a circuit that converts this current into a voltage signal sent to the Nintendo. The circuit is shown in Figure 9[2]. Since there is no documentation available on the integrated circuit (marked as IR3TO7A), the exact details of this circuit are unknown, but examination of the circuits outputs led to the conclusion that this IC is a demodulator. Resistor $R1$ and $C1$ set the frequency of demodulation; since the Zapper is designed to be uesd with a television, this frequency is set to equal the refresh rate of NTSC video. Essentially, the light gun circuit looks at the light level coming from the TV at each frame, and outputs a pulse width corresponding to that light level. Measured values of the pulse width ranged from 0ms (black screen) to 1.36ms (white). A lens at the end of its barrel gives the Zapper a fairly accurate aim.

In order to determine if the shot hit the target, the Nintendo system draws a black screen with a white square at the target location during the frame after the trigger is pulled. If the light gun outputs a pulse during this frame – presumably above some threshold pulse width, athough the exact operation of the Nintendo is unkown – the shot is registered as a hit.

The triggering machanism for the light gun is a spring-loaded switch, indicated in the circuit diagram. Ordinarily, the `trigger` signal voltage is high (5V), but is shorted to ground when the trigger is pulled. Experimentation indicates that the Nintendo is triggered off the rising edge of the `trigger` voltage. That is, it registers a shot when the trigger is released.

[2]http://www.zero-soft.com/HW/USB_ZAPPER/

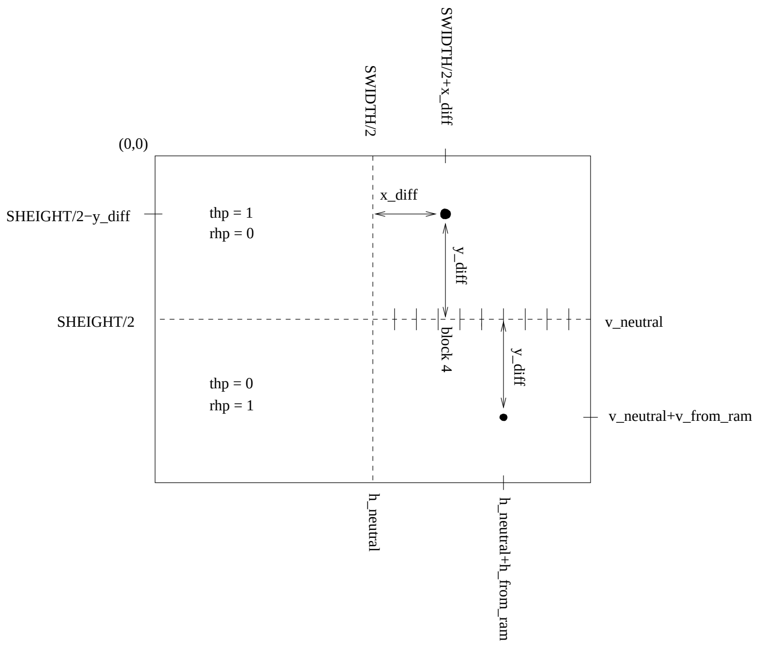


Figure 8: Coordinate Conversion Example

The coordinate conversion module uses x and y coordinates to find the required motor angles. By dividing the screen into two symmetric half-screens and each half-screen into blocks, the size of the lookup table can be dramatically reduced

3.1 Hacking the Light Gun

In order to trigger the light gun from the lab kit and to use the `optical` signal for the calibration routine, it was necessary to take apart the light gun. The `optical` and `trigger` signals described above are transmitted to the Nintendo through a cable with the connector shown in Figure 13. Rather than trying to interface with this custom connector, it seemed more practical to disassemble the light gun and solder additional wires to the individual wires where they enter the circuit board. In addition, the trigger was removed and the two wires previously connected to the trigger switch were extended. All of these wires were then brought out from the gun through the space that originally held the trigger. These modifications proved to be more difficult than expected due to the tiny wires used in the Zapper, which broke easily when removed from their original insulation. However, with some patience, the `power`, `neutral`, `optical`, and `trigger` signal wires as well as the trigger wires were extended, and the light gun reassembled.

To properly use these signals, it is important to note that the lab kit and Nintendo use different electrical plugs to connect to line power, which indicates that they are operating with different voltage references. The Nintendo system uses a NEMA 1-15 connector – an ungrounded two-pin plug – which indicates that its voltages are referenced to line neutral rather than earth ground. The lab kit's plug is a NEMA 5-15 plug – similar to the NEMA 1-15 but with an additional pin to earth ground.[3] From the lab kit schematic (http://www-mtl.mit.edu/Courses/6.111/labkit/labkit_schematic.pdf, page 28), it is clear that the lab kit voltages are referenced to earth ground rather than to line neutral.

[3]http://en.wikipedia.org/wiki/Outlet

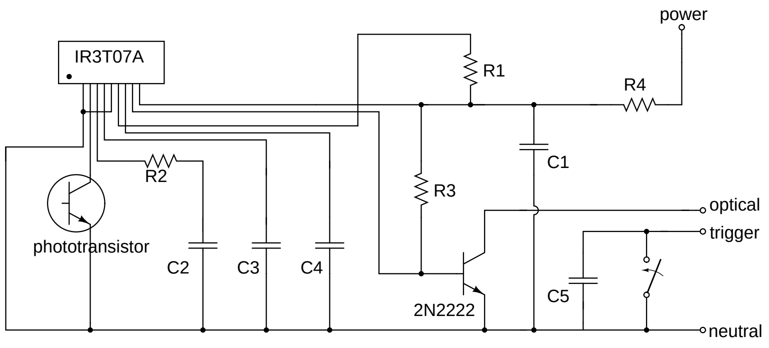


Figure 9: Internal Light Gun Circuit

This circuit triggers off the refresh rate of NTSC video to output a pulse signal voltage between “optical” and “neutral,” with the pulse width corresponding to the light level from the television

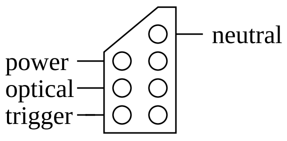


Figure 10: Configuration of Light Gun Connector

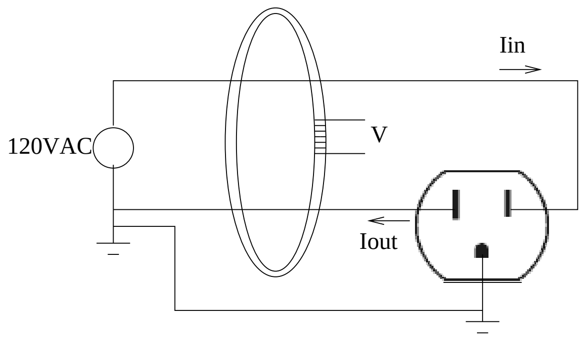


Figure 11: Ground Fault Circuit Interruptor

If the neutral and earth ground terminals are connected, current flows through bad paths and trips the GFCI

Since the two systems have different reference voltages, the signals from the light gun cannot be used directly as inputs to the lab kit. For example, connecting `optical` to one of the lab kit user inputs would not work, because the two voltage references could potentially be very different, making the voltage signal useless. Furthermore, the two references cannot simply be tied together, because this would draw current from the power line and, among other things, trip the ground fault circuit interruptor (GFCI).

The GFCI is a safety mechanism present in many buldings (including the 6.111 laboratory) that monitors current levels in the hot and neutral power lines to determine if there is a fault in the electrical wiring. It is essentially a toroid through which both the hot and neutral lines pass (Figure 11). By Faraday's law, the voltage across the windings on the toroid is proportional to the total flux linked, so that if I_{in} and I_{out} are equal, the net flux and hence V is zero. If, however, there is a fault somewhere in the circuit, some of the current will be drawn away from the neutral line, and so there will be a net flux linkage in the toroid. The toroid voltage is monitored by a system that shuts off power to the outlet if this case occurs. The GFCI was designed primarily in case of a short from the hot line to earth ground, e.g. through a person. However, since the neutral line voltage is generally somewhat higher than ground ($I_{\text{out}}R$, where R is the resistance of the (generally long) wire connecting the outlet to ground), if the neutral line is connected directly to ground before the toroid then the GFCI will also be tripped.

The problem of different reference voltages was circumvented using optical isolators to communicate signals between the lab kit and Nintendo. These were built using LED and phototransistor (Lite-on LTR-301) pairs. The following sections describe the circuits used in greater detail.

3.2 Triggering

Triggering the light gun electrically is a matter of replacing the original mechanical switch with a transistor, and being able to control this transistor from the lab kit. As described in Section 3.1, this goal is somewhat complicated because the lab kit must remain electrically isolated from the light gun. Using an opto-isolator, however, the solution is simple. The circuit is shown in Figure 12. *Duckhunter* uses one of the connectors on the user I/O banks to light an LED, with the 1kΩ resistor limiting the current drawn. The light shining on the base of the phototransistor causes current to flow from its collector to emitter, shorting `trigger+` and `trigger-` together.

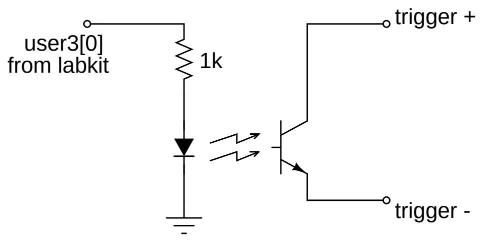


Figure 12: Light Gun Trigger Circuit

This circuit triggers the light gun electrically from the lab kit while keeping the lab kit electrically isolated from the NES.

3.3 Optical Sensor Output

As described in Section 3.1, the light gun outputs a pulse train signal based on the brightness of the part of the television it is aimed at. When converting this signal to a useable input for the lab kit, it is especially important to avoid loading the light gun at all, since doing so would alter the signal received by the Nintendo and thus the operation of the game. The `optical` signal is therefore used as the inverting input to an operational amplifier, since ideally an opamp has an infinite input impedance so there is no difference between it and an open circuit. The opamp is connected as a comparator with hysteresis to keep the signal as close to a square wave as possible. It is run off of the power and neutral voltages from the light gun (5 volts). Similarly, the corresponding signal is passed through another comparator on the lab kit input side. Since each comparator inverts the signal, the voltage going into the labkit corresponds directly with the signal received by the Nintendo. The complete circuit is shown in Figure 13, and a typical waveform is shown in Figure 3.3. This figure shows the signal at the input to the lab kit when the light gun is pointed at an intermediate brightness.

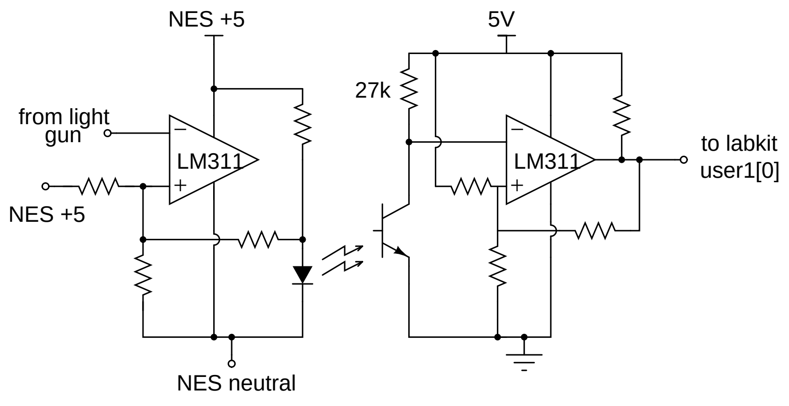


Figure 13: Optical Conversion Circuit

This circuit converts the optical output from the light gun to a useable signal for the lab kit. The input comparator ensures that this circuit draws only a negligeable amount of current from the light gun, and the output comparator cleans up the signal from the LED. Except where otherwise noted, all resistor values are 1kΩ.

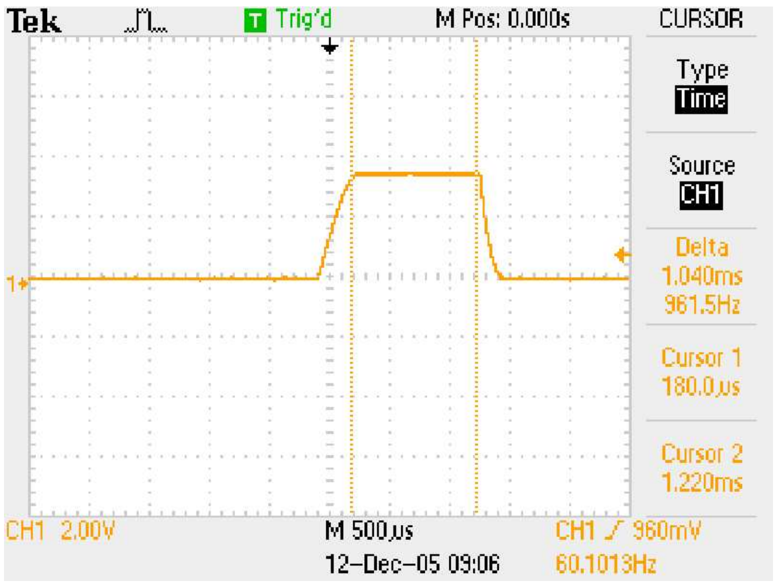


Figure 14: Optical Signal Input to Lab Kit

When this snapshot was taken, the light gun was pointed at a television screen displaying a moderately bright signal. The pulse width corresponds to the brightness, with the maximum observed pulse width at 1.36ms.

4 Debugging

Debugging was performed in different ways for the two different systems. For the light gun, a series of demos provided feedback as to the correct (or incorrect) workings of the gun and allowed for focused debugging. The disc tracking system followed a more traditional debugging approach due to the nature of the system. Since video output was available that, in most cases, gave a good idea of the failure, a more traditional method of code, observe, debug, reobserve was followed.

4.1 Demo Modes

In order to debug *Duckhunter*, a series of tests were developed that isolated a single module or aspect of the design. Later, these debugging modes were used as demonstration modes. A description of these tests follow. The corresponding Verilog code is appended in Section 6.1.8.

Many of the various modules that make up *Duckhunter* have individual demo modes, so that they can be demonstrated independently. The demo modes are reached by setting the switches on the labkit to the binary representation of the demo mode, and then pressing the enter button and button 0 simultaneously. Table 1 provides a summary of the different demo mode numbers and the inputs used in each mode.

4.1.1 Trigger

This mode demonstrates the ability to "shoot" the light gun using an output from the lab kit. It tests both the light gun trigger circuit (Figure 12) and the ability to use an output from the user I/O banks. In this

Table 1: Demo Modes: number, name, and inputs used

#	name	inputs
1	Trigger Demo	`button0`
2	Coordinate Conversion	`switch[0]`, `switch[7:4]`
3	Pong	`button_enter`, `switch[7:4]`, `button_up`, `button_down`
4	Servomechanism Demo	`button_up`, `button_down`, `button_right`, `button_left`
5	Laser Demo	`button1`
6	Video Output Demo	`switch[1:0]`
7	Light Detection Demo	`switch[1:0]`
8	Hit Location Demo	`switch[1:0]`
255	Calibration Mode	`button0`

mode, pressing button0 causes the light gun to fire (when the trigger signal is connected to `user3[0]`).

4.1.2 Coordinate Conversion

The coordinate conversion demo mode demonstrates the the `coord_conversion` and `rc_motor_move` modules. The test module, (`test`), generates a XVGA signal showing a white square on a black background. The inputs to `test` are the `vsync`, `hcount`, and `vcount` signals required to generate the XVGA signal, as well as `clock_27mhz` and `enable`. When `test`'s `enable` signal is high, this white square scans across the screen beginning in the upper left corner. At the same time, `test` outputs the coordinates of the upper left corner of this target to the `coord_conversion` module, which ultimately drives the motors to point at the target. `switch[7:4]` are used to input the television distance to the conversion module.

4.1.3 Pong

The pong demo mode is similar to the coordinate conversion demo, except the target is generated by Andrew Lisy's pong game from Lab 4. `switch[7:4]` sets the speed of the puck, `button_up` and `button_down` control the paddle, and `button_enter` resets the game. This mode is a good demonstration of how the gun moves when driven at different speeds.

4.1.4 Servomechanisms

This demo mode uses the test module `driver_test` to demonstrate the servo driver directly. Its inputs are `h_speed` and `v_speed`, which are set with the lab kit switches `switch[7:4]` and `switch[3:0]` respectively, and the `up`, `down`, `left`, and `right` buttons on the lab kit. The module changes the pulse width sent to `rc_motor_move` based on the user button inputs. The pulse width is also displayed on the LED display on the lab kit.

4.1.5 Laser Pointer

This demo mode demonstrates the ability to turn the laser pointer on and off using a lab kit output. It is essentially the same as the trigger demo, but with the input `button1` and the output `user3[1]`. The other difference is that the laser pointer is an active low (it turns on when its switch signal is grounded), so the button input must be inverted.

4.1.6 NTSC Video Output

The video output demonstration simply outputs the calibration video patterns. This video signal is described in depth in Section 2.2.1. The lab kit switches select between horizontal bar and vertical bar mode, or colorbars.

4.1.7 Light Detection

The light detection demonstration uses the `hit` output signal of the `hit_detection` module (Section 2.2.2) to light the lab kit LEDs. Observers can see the LEDs light when one of the bars created in the calibration video module passes through the light gun's sights.

4.1.8 Hit Location

The hit location demonstration outputs the coordinates `x_hit` and `y_hit` created by the `hit_location` module (Section 2.2.3) onto the hex display. Using `switch[1:0]`, the user can choose between observing x-coordinate and y-coordinate hits.

4.1.9 Calibration Mode

The calibration mode is used to calibrate the system. It runs through exactly the process described in Section 2.2.4. In order to aid debuggung, this module displays a XVGA signal showing the x and y positions of the intended point of aim and the hit position.

The calibration mode was the most difficult to debug, because it exposed errors in the modules it used which otherwise had seemed to be working. At first, for example, there was a sign error in the `offset` inputs to the coordinate conversion module, which caused the targets displayed on the computer monitor to move away from, instead of towards, each other. Once this was fixed, the calibration module continued to display an interesting charactaristic that remains unsolved. Although it would initially move incrementally towards the desired coordinates, the hit position would generally begin to move away again after a certian number of iterations.

To a certain extent, this behavior is to be expected, since the gun can only aim to a certain accuracy due to the division of the screen into blocks (for ease of coordinate conversion and to reduce the required memory size).

The current solution to this behavior is to keep a certain level of human input to the system, so that when the calibration looks "good enough," the user halts the process and switches over to *Duckhunter* mode. A more elegant solution, which would have been persued had time allowed, is to have the calibration routine halt when the new offsets required to adjust the aim fall below a certain threshold value.

4.2 Disc Tracking

The issues in the disc tracking system came from two sources. The first problems to arise was getting the NTSC resolution video into a module that only supported 256x192 video. Once this problem was resolved, the next issues to arise were with the trajectory system. Originally, the trajectory system was integrated into the `vga_with_ram` module. However, when they were split and modularized individually, the problems were lessened.

4.2.1 Video Debugging

Since the original module provided by Javier Castro only supported a black and white resolution of 256x192, substantial modifications needed to be made in order to capture and display the 640x480 signal provided by the Nintendo NTSC signal. The first step in boosting the resolution was increasing the size of the dual-port ram address space from 16 bits to 18 bits. Since 16-bits only allows for a maximum of 256x256 (using the [yyyyyyyyxxxxxxxxx] scheme discussed earlier in the `NTSC_decoder` section, I clearly needed more space. The BRAMs at 8 bits wide can support only 262144 (2^18, or 18-bits), so I decided to use that. However, this arrangement only allowed for a 512x512 display resolution, which would be fine in the y direction, but come up short in the x-direction. The problem was, of course, that the final 128 bits at the right side of the video signal would not be included in the RAM, so no calculations could be done with the values.

Since I was not very concerned with image quality on the VGA display (since we were also piping the original input to a TV), I decided to save only every other pixel in the x-direction. This modification allowed me to receive all of the x-resolution in memory, at the expense of some signal distortion. Since there would still be more than enough pixels of "disc", and I could compensate for the distortion by shifting the eventual output coordinates, I opted to make this modification, which involved changing the dual-port RAM and updating the address lines and counter registers throughout the original system. Once this was done, I had a working image of the NES display with which to work.

The next (and recurring) issue was with the trajectory module. Since the signal was noisy, there was an inherent difficulty in preventing "false positives" caused by the stray white pixels over the display. My first attempt at disc detection was to look for one white pixel anywhere on the screen above the menubar. The worked a small percentage of the time, but was fooled frequently be noise before the disc had an opportunity to be "shot". My next attempt to compensate for this was to divide the screen into two zones, and require a detection on both before either would become valid. This worked better, however, the first zone was still frequently tripped by stray pixels, thus causing an inaccurate reading on the first region.

Thinking about this problem, a revelation came to me one night as I lay awake in bed. I thought to implement a sub-FSM to look for two consecutive pixels in each zone. This would perfect match the disc flying through, since it was much less likely that two pixels would show up sequentially in each zone. When I loaded up the code for this, it worked better than the original system, but still not quite enough for production. I added another state to the pixel FSM, as well as a 'clear' state for when it detects non-sequential pixels, and my system worked much better.

I then extended the zones to four sequential regions with adjustable bars delimiting each. Now, I could display four red squares corresponding to the disc as it passed through the regions. However, there was still an occasional glitch when the disc would pass through, but nothing would be registered. I looked at my code, realized it was now far too complicated to easily debug, and decided that I needed to take apart the giant `vga_with_ram` module and make a designated `trajectory` module. Once this was completed, I had a much more stable system.

When I designed the `trajectory` module, I added special signals, `count` and `offset`, to enable easy integration of a second disc. The exact details are explained in the `trajectory` description. Although most of the logic was there, since I had made my primary concern the solid timing and prediction of the first disc, I was unable to fully implement the second disc functionality. However, due to the encapsulation of each disc's trajectory, the basis is there and could be added (relatively) easily in the future.

4.2.2 Integrating the Systems

Integration proved to be more difficult than expected. Initially, there were some communication issues between the two parts. For example, the coordinates generated by the video input system were for a 640x480 screen, while the coordinate conversion module and television screen expected a resolution of 720 by 486 pixels. Most of these small difficulties were ironed out quickly.

The one remaining issue that *Duckhunter* had was using the video tracking system to trigger the light gun. Although it was proven early on in the project that it was possible to trigger the gun by pressing a button on the labkit, we did not initially consider the importance of the duration of this button press. The original Nintendo light gun trigger is spring loaded so that a single pull back both closes and opens the switch contacts. This sends the Nintendo a pulse of approximately the same width every time. In order to emulate this behavior so that the Nintendo would register our trigger signal, we should have measured that pulse width and sent one of the same length. We didn't encounter this problem originally because in testing with a pushbutton, the natural pulse width of pushing the button coincided roughly with what the Nintendo expected. Unfortunately, this simple solution did not occur to us until after the checkoff.

5 Conclusions

Our project required both new design and integration with old design. As a result, we gained invaluable experience dealing with not only our own code and Verilog, but also already-integrated systems. This is extremely important and will serve us well in our future academic and professional lives, since much of engineering is about designing systems that work alongside and interface with systems designed by others.

Both subsystems required interfacing with the Nintendo signals. The gun system needed to use the same signals provided and expected by the Nintendo. Furthermore, the system needed to be robust enough to withstand perturbations in angle, distance, and tracking speed. To compensate for the angle and distance variables, Taylor constructed a calibration system that would determine run-time offsets for the motors to ensure that coordinates pointed to the same physical location from run to run. To adjust for motor speed, Andrew's system is able to change the time allowed for the gun to retrack in realtime. These systems keep the gun accurate across demonstrations, and ensure that we don't need to continuously adjust its calibration manually.

The tracking subsystem also needed a measure of robustness to compensate for noise in the video signal. Since errant white pixels can throw off the disc trajectory utility, Andrew built a two-FSM system which filtered out the noise inherent in the signal. Although the probability of a false positive is not quite 0, these improvements bring the chance down to an acceptable level.

Overall, the project was very helpful in giving us first-hand experience at systems engineering. In a non-ideal world, its is important to have mechanisms to control the independent variables, and this system does just that.

5.1 Extensions

Although our system performs relatively well, there are three areas in which improvements could be made: general integration, second disc tracking, and tracking precision. The general integration is perhaps the closest goal. At this point, although the gun does move to track the interpolated disc, the trigger signal is not asserted correctly. The reason behind this is the differences in a computer asserting a signal and a human finger. To properly cause the trigger signal to be interpreted by the Nintendo, we need to attach the oscilloscope to the trigger and measure the signal duration that comes out. Our first attempt used a single clock high signal, which did not properly fire the gun. With some empirical data, we could determine the exact duration that the signal needs to be high, and then use that effectively.

6 Appendices

6.1 Verilog Code

6.1.1 Interface Module (detection_toplevel.v)

detection toplevel:

```
module detection_toplevel (reset, clock_27mhz, upbutton, downbutton, ledbank,
vga_out_red, vga_out_green, vga_out_blue,
vga_out_sync_b, vga_out_blank_b, vga_out_pixel_clock,
vga_out_hsync, vga_out_vsync,
tv_in_reset_b, tv_in_i2c_clock, tv_in_i2c_data,
tv_in_ycrcb, switch, tv_in_line_clock1, x_coord, y_coord, trigger);

/*General I/O*/
input reset, clock_27mhz;
input upbutton, downbutton;  //synchronized/debounced versions of button_up and button_down
input [7:0] switch;
input       tv_in_line_clock1;


output [7:0] ledbank;
output   trigger;

wire   trigger;


/*Video outputs*/
output [7:0] vga_out_red, vga_out_green, vga_out_blue; // Outputs to DAC
output vga_out_sync_b, vga_out_blank_b; // Composite sync/blank outputs to DAC
output vga_out_pixel_clock; // Pixel clock for DAC
output vga_out_hsync, vga_out_vsync; // Sync outputs to VGA connector

/*NTSC */
output tv_in_reset_b; // Reset signal to ADV7185
output tv_in_i2c_clock; // I2C clock output to ADV7185
output tv_in_i2c_data; // I2C data line to ADV7185
input  [19:0] tv_in_ycrcb;

output [10:0] x_coord;
output [9:0]  y_coord;



/*Set bars - this determines the zones where the discs will be detected*/
reg [9:0] bar1, bar2, bar3, bar4, bar5;
initial bar1 = 10'h093;
initial bar2 = 10'h0F4;
initial bar3 = 10'h16F;
initial bar4 = 10'h17F;
initial bar5 = 10'h18D;

wire [63:0] hexbank;
wire [7:0]  region_leds;
wire [63:0] hex_out;
wire [7:0]  vram_data_out;
reg [17:0]  vram_addr;
reg [7:0]   vram_data_in;
reg         vram_clk;
reg         vram_we;

wire [9:0] x_coord_tmp;
assign x_coord = {0, x_coord_tmp};

vga_with_ram vr(reset, clock_27mhz, vga_out_red, vga_out_green,
vga_out_blue,
vga_out_sync_b, vga_out_blank_b, vga_out_pixel_clock,
vga_out_hsync, vga_out_vsync,
vram_addr, vram_data_in, vram_data_out, vram_clk, vram_we, bar1, bar2, bar3,
bar4, bar5,region_leds, x_coord_tmp, y_coord, trigger);

adv7185init adv7185(.reset(reset), .clock_27mhz(clock_27mhz),
    .source(1'b0), .tv_in_reset_b(tv_in_reset_b),
    .tv_in_i2c_clock(tv_in_i2c_clock),
    .tv_in_i2c_data(tv_in_i2c_data));

wire [29:0] ycrcb; // video data (luminance, chrominance)
wire [2:0]  fvh; // sync for field, vertical, horizontal
wire       dv; // data valid

ntsc_decode decode (.clk(tv_in_line_clock1), .reset(reset),
    .tv_in_ycrcb(tv_in_ycrcb[19:10]),
    .ycrcb(ycrcb), .f(fvh[2]),
    .v(fvh[1]), .h(fvh[0]), .data_valid(dv));


/* Display memory: test pattern or NTSC video */
//Put test pattern on vdata3
wire [17:0] vaddr3;
wire [7:0]  vdata3;
wire        vclk3;
wire        vwe3;
vid_test_pat vp3 (clock_27mhz,vaddr3,vdata3,vclk3,vwe3);


//Convert YCrCb data to RGB data
wire [7:0] R, G, B;
YCrCb2RGB color_conv(R,G,B,clock_27mhz,reset,ycrcb[29:20],ycrcb[19:10],ycrcb[9:0]);

//Put NTSC-in signal on vdata4
wire [17:0] vaddr4;
wire [7:0]  vdata4;
wire        vwe4;
```

```
   wire        vclk4;
   ntsc_to_ram vp4 (.clk(tv_in_line_clock1),
    .fvh(fvh), .dv(dv), .din({R[7:5],G[7:5],B[7:6]}),//put RGB data directly into RAM
    .vaddr(vaddr4), .vwe(vwe4), .vdata(vdata4));

 // select video source
    always @(switch[0])
     case (switch[0])
       1'b0: begin // fill video RAM with NTSC video data
  vram_addr = vaddr4;
  vram_data_in = vdata4; // luminance data
  vram_clk = tv_in_line_clock1; //vclk4;
  vram_we = vwe4;
       end

       1'b1: begin  // fill video RAM with b/w bars
  vram_addr = vaddr3;
  vram_data_in = vdata3;
  vram_clk = vclk3;
  vram_we = vwe3;
       end
     endcase // case(switch[0])

   reg bu_old, bd_old, br_old, bl_old;
   reg [3:0] line_speed;

   //diagnostic mode
   always @ (posedge clock_27mhz)
   begin
      bu_old <= ~upbutton;
      bd_old <= ~downbutton;

      case (switch[7:6])
2'd0: line_speed <= 4'd2;
2'd1: line_speed <= 4'd5;
2'd2: line_speed <= 4'd10;
2'd3: line_speed <= 4'd15;
      endcase

      if (switch[3:1] == 3'd1)
begin
   if (~upbutton && ~bu_old) bar1 <= bar1 - line_speed;
   if (~downbutton && ~bd_old) bar1 <= bar1 + line_speed;
end

      if (switch[3:1] == 3'd2)
begin
   if (~upbutton && ~bu_old) bar2 <= bar2 - line_speed;
   if (~downbutton && ~bd_old) bar2 <= bar2 + line_speed;
end

      if (switch[3:1] == 3'd3)
begin
   if (~upbutton && ~bu_old) bar3 <= bar3 - line_speed;
   if (~downbutton && ~bd_old) bar3 <= bar3 + line_speed;
end

      if (switch[3:1] == 3'd4)
begin
   if (~upbutton && ~bu_old) bar4 <= bar4 - line_speed;
   if (~downbutton && ~bd_old) bar4 <= bar4 + line_speed;
end

      if (switch[3:1] == 3'd5)
begin
   if (~upbutton && ~bu_old) bar5 <= bar5 - line_speed;
   if (~downbutton && ~bd_old) bar5 <= bar5 + line_speed;
end
   end // always @ (posedge clock_27mhz)

   assign ledbank = ~region_leds;
   assign hexbank = 64'd0; //= hex_out;
/*
   display_16hex my_display(reset, clock_27mhz, hexbank,
        disp_blank, disp_clock, disp_rs,
    disp_ce_b, disp_reset_b, disp_data_out);
*/
endmodule // detection_toplevel
```

6.1.2 VGA RAM module (vga_with_ram.v)

```
// SVGA video module with dual-port video ram
// the vram_* lines are used to read/write to the video RAM
//

module vga_with_ram (reset, clock_27mhz, vga_out_red, vga_out_green,
     vga_out_blue,
     vga_out_sync_b, vga_out_blank_b, vga_out_pixel_clock,
     vga_out_hsync, vga_out_vsync,
     vram_addr, vram_data_in, vram_data_out, vram_clk,
     vram_we, bar1, bar2, bar3, bar4, bar5, region_leds,
     coord_x, coord_y, trigger);


   input reset; // Active high reset, synchronous with 27MHz clock
   input clock_27mhz; // 27MHz input clock
   output [7:0] vga_out_red, vga_out_green, vga_out_blue; // Outputs to DAC
   output vga_out_sync_b, vga_out_blank_b; // Composite sync/blank outputs to DAC
   output vga_out_pixel_clock; // Pixel clock for DAC
   output vga_out_hsync, vga_out_vsync; // Sync outputs to VGA connector


   input [9:0] bar1, bar2, bar3, bar4, bar5; //zone bars
   output [7:0] region_leds;

   input [17:0] vram_addr; // video ram address
   input [7:0] vram_data_in; // video ram data input
   input  vram_clk, vram_we; // video ram clock and write enable
   output [7:0] vram_data_out; // video ram data output

   output [9:0] coord_x, coord_y; //aiming coords
   output  trigger;


   //assign  discd_x_test = 10'b0000011010;
  // assign  discd_y_test = 10'b0000011010;



   ///////////////////////////////////////////////////////////////////////////
   //
   // Timing values
   //
   ///////////////////////////////////////////////////////////////////////////

   // 1024 X 768 @ 75Hz with a 78.750MHz pixel clock

`define H_ACTIVE 1024 // pixels
`define H_FRONT_PORCH   16 // pixels
`define H_SYNCH   96 // pixels
`define H_BACK_PORCH  176 // pixels
`define H_TOTAL 1312 // pixels

`define V_ACTIVE  768 // lines
`define V_FRONT_PORCH    1 // lines
`define V_SYNCH    3 // lines
`define V_BACK_PORCH   28 // lines
`define V_TOTAL  800 // lines

   ///////////////////////////////////////////////////////////////////////////
   //
   // Internal signals
   //
   ////////////////////////////////////////////////////////////////////////////

   wire pixel_clock;
   reg prst, pixel_reset; // Active high reset, synchronous with pixel clock

   reg [7:0] vga_out_red, vga_out_blue, vga_out_green;
   wire vga_out_sync_b, vga_out_blank_b;
   reg hsync1, hsync2, vga_out_hsync, vsync1, vsync2, vga_out_vsync;

   reg [10:0] pixel_count; // Counts pixels in each line
   reg [10:0] line_count; // Counts lines in each frame

   reg [9:0] xpos; // horizontal image pixel count
   reg [9:0] ypos; // vertical image pixel count


   ////////////////////////////////////////////////////////////////////////////
   //
   // Generate the pixel clock (78.750MHz)
   //
   ////////////////////////////////////////////////////////////////////////////

   // synthesis attribute period of clock_27mhz is 37ns;

   DCM vga_dcm (.CLKIN(clock_27mhz),
.RST(1'b0),
.CLKFX(pixel_clock));
   // synthesis attribute DLL_FREQUENCY_MODE of vga_dcm is "LOW"
   // synthesis attribute DUTY_CYCLE_CORRECTION of vga_dcm is "TRUE"
   // synthesis attribute STARTUP_WAIT of vga_dcm is "TRUE"
   // synthesis attribute DFS_FREQUENCY_MODE of vga_dcm is "LOW"
   // synthesis attribute CLKFX_DIVIDE of vga_dcm is 9
   // synthesis attribute CLKFX_MULTIPLY of vga_dcm is 26
   // synthesis attribute CLK_FEEDBACK of vga_dcm  is "NONE"
   // synthesis attribute CLKOUT_PHASE_SHIFT of vga_dcm is "NONE"
   // synthesis attribute PHASE_SHIFT of vga_dcm is 0
   // synthesis attribute clkin_period of vga_dcm is 37

   assign    vga_out_pixel_clock = ~pixel_clock;

   always @(posedge pixel_clock)
     begin
prst <= reset;
pixel_reset <= prst;
     end

   ////////////////////////////////////////////////////////////////////////////
   //
   // Pixel and Line Counters
   //
   ////////////////////////////////////////////////////////////////////////////
 always @(posedge pixel_clock)
     if (pixel_reset)
       begin
  pixel_count <= 0;
  line_count <= 0;
       end
     else if (pixel_count == (`H_TOTAL-1)) // last pixel in the line
       begin
  pixel_count <= 0;
  if (line_count == (`V_TOTAL-1)) // last line of the frame
    line_count <= 0;
  else
    line_count <= line_count + 1;
       end
     else
       pixel_count <= pixel_count +1;

   always @(pixel_reset or pixel_count or line_count)
     if (pixel_reset)
       begin
  xpos <= 0;
  ypos <= 0;
       end
     else begin
        if (pixel_count > `H_FRONT_PORCH)
  xpos <= pixel_count - `H_FRONT_PORCH;
        if (line_count > `V_FRONT_PORCH)
  ypos <= line_count - `V_FRONT_PORCH;
     end


   ////////////////////////////////////////////////////////////////////////////
   //
   // Sync and Blank Signals
   //
   ////////////////////////////////////////////////////////////////////////////

   always @ (posedge pixel_clock)
     begin
if (pixel_reset)
  begin
     hsync1 <= 1;
     hsync2 <= 1;
     vga_out_hsync <= 1;
     vsync1 <= 1;
     vsync2 <= 1;
     vga_out_vsync <= 1;
  end
else
  begin

     // Horizontal sync
     if (pixel_count == (`H_ACTIVE+`H_FRONT_PORCH))
       hsync1 <= 0; // start of h_sync
     else if (pixel_count == (`H_ACTIVE+`H_FRONT_PORCH+`H_SYNCH))
       hsync1 <= 1; // end of h_sync

     // Vertical sync
     if (pixel_count == (`H_TOTAL-1))
       begin
  if (line_count == (`V_ACTIVE+`V_FRONT_PORCH))
    vsync1 <= 0; // start of v_sync
  else if (line_count == (`V_ACTIVE+`V_FRONT_PORCH+`V_SYNCH))
    vsync1 <= 1; // end of v_sync
       end

  end

// Delay hsync and vsync by two cycles to compensate for 2 cycles of
// pipeline delay in the DAC.
hsync2 <= hsync1;
vga_out_hsync <= hsync2;
vsync2 <= vsync1;
vga_out_vsync <= vsync2;

     end

   // Blanking
   assign vga_out_blank_b = ((pixel_count<`H_ACTIVE) & (line_count<`V_ACTIVE));

   // Composite sync
   assign vga_out_sync_b = hsync1 ^ vsync1;


   ////////////////////////////////////////////////////////////////////////////
   //
   // Display an image from dual-port RAM
   //
   ////////////////////////////////////////////////////////////////////////////

   reg [7:0] pixdata; // for the memory read pipeline
   reg [17:0] memaddr; // for memory read address
   wire [7:0] memdata;  // for memory output data 8 bits: BBGGGRRR
   // read data from memory at pixel clock, with one pipeline stage
   reg [9:0] interp1_x = 10'd0;
   reg [9:0] interp1_y = 10'd0;
   reg interp1_detect = 1'b0;

   reg [26:0] count;
   reg        start_count;

   initial count = 0;
   initial start_count = 0;

   reg        trigger, trigger_temp;
   reg [9:0]  coord_x,coord_y;

   initial coord_x = 10'h17f;
   initial coord_y = 10'h93;

   //assign    hexbank = {4'hF,34'd0,count};

   wire [9:0] disca_x, disca_y, discb_x, discb_y, discc_x, discc_y, discd_x, discd_y;
   wire       disc_detect;
   wire [7:0] region_leds;

   wire [7:0] region1_leds;
   wire [1:0] region1, region2;

   reg        trigger_count_start = 0;
   reg [9:0]  trigger_count;

   //trajectory instance for disc1
   trajectory traj1(pixel_clock, reset, xpos, ypos, 26'd0, 26'd0, memdata, bar1, bar2, bar3, bar4, b
    discb_y, discc_x, discc_y, discd_x, discd_y, disc_detect, region1_leds, interp_display, region1)

//trajectory instance for disc2 -- not used
   //trajectory traj2(pixel_clock, reset, xpos, ypos, 26'd9000000, count, memdata, bar1, bar2, bar3,
//     discb2_y, discc2_x, discc2_y, discd2_x, discd2_y, disc2_detect, region_leds, interp2_display,


//Diagnostic display
 //assign     hexbank = {2'd0,disca2_x,2'd0,disca2_y,14'd0,count};

   always @(posedge pixel_clock)
     begin

if (reset) count <= 26'd1;
if (disc_detect)
  begin
     interp1_x <= discd_x + (discd_x - discc_x);
     interp1_y <= discd_y + (discd_y - discc_y);
     interp1_detect <= 1'b1;
  end

//Create trigger signal
trigger_temp <= interp_display;
if (~trigger_temp && interp_display)
  begin
     trigger <= 1'b1;
     trigger_count_start <= 1'b1;
  end
else if (trigger_count == 10'd1000)
  begin
     trigger <= 1'b0;
     trigger_count_start <= 1'b0;
  end
else if (trigger_count_start)
  begin
     trigger <= 1'b1;
     trigger_count <= trigger_count + 1;
  end
else
  trigger <= 0;


//Output coordinates
if (interp1_detect == 1)
  begin
     coord_x <= interp1_x - 17;
     coord_y <= interp1_y + 4;
  end
else  //position near center of screen
  begin
     coord_x <= 10'h17F;
     coord_y <= 10'h93;
  end

if (disca_x != 10'd0) count <= count + 1;

memaddr <= {xpos[9:1] + ypos[9:1]*512}; // oversample


/*Display red boxes for first disc*/
/*Draw calibration lines*/
if (ypos == bar1)
```

```
  pixdata <= 8'b11100000;
else if (ypos == bar2)
  pixdata <= 8'b00000011;
else if (ypos == bar3)
  pixdata <= 8'b00011100;
else if (ypos == bar4)
  pixdata <= 8'b11111100;
else if (ypos == bar5)
  pixdata <= 8'b11100011;

//Display horizontal threshold lines
// else if (xpos == horiz_scope1)
//   pixdata <= 8'b11100000;
// else if (xpos == horiz_scope2)
//   pixdata <= 8'b11100000;

//Display trajectory boxes
        else if (disc_detect == 1 && xpos>disca_x && xpos<disca_x+32 && ypos>disca_y && ypos<disca_y
  pixdata <= 8'b11100000;

else if (disc_detect == 1 && xpos>discb_x && xpos<discb_x+32 && ypos>discb_y && ypos<discb_y+32)
  pixdata <= 8'b11100000;

else if (disc_detect == 1 && xpos>discc_x && xpos<discc_x+32 && ypos>discc_y && ypos<discc_y+32)
  pixdata <= 8'b11100000;

else if (disc_detect == 1 && xpos>discd_x && xpos<discd_x+32 && ypos>discd_y && ypos<discd_y+32)
  pixdata <= 8'b11100000;

/*Display blue box for interpolation*/
else if (interp1_detect == 1 && interp_display == 1 && xpos>interp1_x && xpos<interp1_x+32 && ypos>i
  pixdata <= 8'b00000011;

//Display green box corresponding to gun aiming
else if (xpos>coord_x && xpos<coord_x+32 && ypos>coord_y && ypos<coord_y+32)
 pixdata <= 8'b00011100;

//Paint rest of picture
else
  pixdata <= memdata;

     end

   // RAM with a 512 x 512 b&w image, 8-bits per pixel
   // the vga output is 1024x768, so we skip 2 bits of xpos and ypos, each

   videoram vram( .addra(memaddr), .clka(pixel_clock), .douta(memdata),
  .addrb(vram_addr),
  .doutb(vram_data_out), .dinb(vram_data_in),
  .clkb(vram_clk), .web(vram_we), .wea(1'b0), .dina(8'b0));

   // feed pixel values to VGA machine

   always @ (posedge pixel_clock)
     begin

vga_out_red   <= {pixdata[7:5], 5'b00000};
vga_out_green <= {pixdata[4:2], 5'b00000};
vga_out_blue  <= {pixdata[1:0], 6'b000000};


/*
vga_out_red <= pixdata[7:5]; // for black and white display
vga_out_green <= pixdata[4:2];
vga_out_blue <= pixdata[1:0];
 */
     end

endmodule
```

6.1.3 Trajectory Module (trajectory.v)

```
module trajectory(clock, reset, xpos, ypos, offset, count, memdata, bar1, bar2, bar3, bar4, bar5, di
    discb_y, discc_x, discc_y, discd_x, discd_y, disc_detect, region_leds, display_interp, region_de

   input clock, reset;
   input [9:0] xpos, ypos;
   input [25:0] offset, count;
   input [7:0]  memdata;
   input [9:0]  bar1, bar2, bar3, bar4, bar5;
   output [9:0] disca_x, disca_y, discb_x, discb_y, discc_x, discc_y, discd_x, discd_y;
   output [7:0] region_leds;
   output  disc_detect;
   output   display_interp;
   output [1:0]  region_detect;



   reg [9:0]  disca_x, disca_y, discb_x, discb_y, discc_x, discc_y, discd_x, discd_y;
   reg [1:0]  pix_detect;
   reg [1:0]  region_detect;
   reg        disc_detect;
   reg [7:0]  region_leds;
   reg        display_interp;

   reg        start_interval_count, start_wait_count;
   initial velocity = 26'd0;
   initial wait_count = 26'd0;
   initial start_wait_count = 1'b0;

   initial region_leds  = 8'd0;

   initial disca_x = 10'd0;
   initial disca_y = 10'd0;
   initial discb_x = 10'd0;
   initial discb_y = 10'd0;
   initial discc_x = 10'd0;
   initial discc_y = 10'd0;
   initial discd_x = 10'd0;
   initial discd_y = 10'd0;

   parameter  horiz_scope1 = 150;
   parameter  horiz_scope2 = 600;


   always @ (posedge clock)
     begin
//Wait for trigger signal
if (wait_count == 26'd18000000) display_interp <= 1'b1;

if (start_wait_count) wait_count <= wait_count + 1;

if (reset)   //reset all hit counters
  begin
     disc_detect <= 1'b0;

     disca_x <= 10'd0;
     disca_y <= 10'd0;
     discb_x <= 10'd0;
     discb_y <= 10'd0;
     discc_x <= 10'd0;
     discc_y <= 10'd0;
     discd_x <= 10'd0;
     discd_y <= 10'd0;

     region_detect <= 2'b0;
     region_leds <= 8'd0;

     pix_detect <= 2'd0;

     interval_count <= 26'd0;
     velocity <= 26'd0;
     start_interval_count <= 1'b0;
     start_wait_count <= 1'b0;
     wait_count <= 26'd0;

     display_interp <= 1'b0;


  end

/*Stage 4*/
//Final part
if (xpos==discd_x+2 && ypos==discd_y && memdata == 8'b11111111 && disc_detect == 0 &&
    region_detect == 2'd3)
  begin
     velocity <= interval_count;
     start_wait_count <= 1'b1;
     start_interval_count <= 1'b0;
     //interval_count <= 26'd0;
     wait_count <= 26'd0;
     disc_detect <= 1'b1;
     pix_detect <= 2'd0;
     region_detect <= 2'd4;
     region_leds <= 8'b00001111;
  end

//*Second part*//
else if (xpos==discd_x+1 && ypos==discd_y && memdata == 8'b11111111 && disc_detect == 0
 && region_detect == 2'd3 && pix_detect == 2'd1)
  begin
     pix_detect <= 2'd2;
    end

//*First part*//
else if (ypos>bar1 && ypos<bar2 && memdata == 8'b11111111 && disc_detect == 0
 && xpos>horiz_scope1 && xpos<horiz_scope2 && region_detect == 2'd3)
  begin
     pix_detect <= 2'd1;
     discd_x <= xpos;
     discd_y <= ypos;
     //region_leds <= 8'b00000100;
  end

//*Clear part - reset if next pixel isn't white*//
else if ((xpos==discd_x+1 || xpos==discd_x+2) && ypos==discd_y && memdata != 8'b11111111 && disc_det
 && region_detect == 2'd3 && (pix_detect == 2'd1 || pix_detect == 2'd2))
  begin
     pix_detect <= 2'd0;
     discd_x <= 10'd0;
     discd_y <= 10'd0;
     //region_leds <= 8'd0;
  end


/*Stage 3 --------------------------------*/
//*Final part*//
        if (xpos==discc_x+2 && ypos==discc_y && memdata == 8'b11111111 && disc_detect == 0
 && region_detect == 2'd2) //&& pix_detect == 2'd2)
  begin
     start_interval_count <= 1'b1;
     interval_count <= 26'd0;
     start_wait_count <= 1'b0;
     wait_count <= 26'd0;
     velocity <= 26'd0;
     pix_detect <= 2'd0;
     region_detect <= 2'd3;
     region_leds <= 8'b00000111;
  end

//*Second part*//
else if (xpos==discc_x+1 && ypos==discc_y && memdata == 8'b11111111 && disc_detect == 0
 && region_detect == 2'd2 && pix_detect == 2'd1)
  begin
     //disc_detect <= 1'b1;
     pix_detect <= 2'd2;
     //region_detect <= 2'd1;
     //disca_x <= xpos;
     //disca_y <= ypos;
     //region_leds <= 8'b0000001;
  end

//*First part*//
else if (ypos>bar2 && ypos<bar3 && memdata == 8'b11111111 && disc_detect == 0
 && xpos>horiz_scope1 && xpos<horiz_scope2 && region_detect == 2'd2)
  begin
     pix_detect <= 2'd1;
     discc_x <= xpos;
     discc_y <= ypos;
     //region_leds <= 8'b00000100;
  end

//*Clear part - reset if next pixel isn't white*//
else if ((xpos==discc_x+1 || xpos==discc_x+2) && ypos==discc_y && memdata != 8'b11111111 && disc_det
 && region_detect == 2'd2 && (pix_detect == 2'd1 || pix_detect == 2'd2))
  begin
     pix_detect <= 2'd0;
     discc_x <= 10'd0;
     discc_y <= 10'd0;
     //region_leds <= 8'd0;
  end



/*Stage 2 ---------------------------*/
//*Final part*//
if (xpos==discb_x+2 && ypos==discb_y && memdata == 8'b11111111 && disc_detect == 0
 && region_detect == 2'd1) //&& pix_detect == 2'd2)
  begin
     //disc_detect <= 1'b1;
     pix_detect <= 2'd0;
     region_detect <= 2'd2;
     //disca_x <= xpos;
     //disca_y <= ypos;
     region_leds <= 8'b00000011;
  end

//*Second part*//
else if (xpos==discb_x+1 && ypos==discb_y && memdata == 8'b11111111 && disc_detect == 0
 && region_detect == 2'd1 && pix_detect == 2'd1)
  begin
     //disc_detect <= 1'b1;
     pix_detect <= 2'd2;
     //region_detect <= 2'd1;
     //disca_x <= xpos;
     //disca_y <= ypos;
     //region_leds <= 8'b0000001;
  end

//*First part*//
else if (ypos>bar3 && ypos<bar4 && memdata == 8'b11111111 && disc_detect == 0
 && xpos>horiz_scope1 && xpos<horiz_scope2 && region_detect == 2'd1)
  begin
     pix_detect <= 2'd1;
     discb_x <= xpos;
     discb_y <= ypos;
     //region_leds <= 8'b00000100;
  end

//*Clear part - reset if next pixel isn't white*//
else if ((xpos==discb_x+1 || xpos==discb_x+2) && ypos==discb_y && memdata != 8'b11111111 && disc_det
 && region_detect == 2'd1 && (pix_detect == 2'd1 || pix_detect == 2'd2))
  begin
     pix_detect <= 2'd0;
     discb_x <= 10'd0;
     discb_y <= 10'd0;
     //region_leds <= 8'd0;
  end



/*Stage 1 --------------------------*/
//*Final part*//
if (xpos==disca_x+2 && ypos==disca_y && memdata == 8'b11111111 && disc_detect == 0
 && region_detect == 2'd0) //&& pix_detect == 2'd2)
  begin
     //disc_detect <= 1'b1;
     pix_detect <= 2'd0;
     region_detect <= 2'd1;
     //disca_x <= xpos;
     //disca_y <= ypos;
     region_leds <= 8'b00000001;
  end

//*Second part*//
else if (xpos==disca_x+1 && ypos==disca_y && memdata == 8'b11111111 && disc_detect == 0
 && region_detect == 2'd0 && pix_detect == 2'd1)
  begin
     //disc_detect <= 1'b1;
     pix_detect <= 2'd2;
     //region_detect <= 2'd1;
     //disca_x <= xpos;
     //disca_y <= ypos;
     region_leds <= 8'b11000000;

  end

//*First part*//
else if (ypos>bar4 && ypos<bar5 && memdata == 8'b11111111 && disc_detect == 0
 && xpos>horiz_scope1 && xpos<horiz_scope2 && region_detect == 2'd0)
 //&& count >= offset)
  begin
     pix_detect <= 2'd1;
     disca_x <= xpos;
     disca_y <= ypos;
     region_leds <= 8'd1;
    // start_count <= 1'b1;
   //  count <= 26'd0;
    // region_leds <= 8'b10000000;

  end

//*Clear part - reset if next pixel isn't white*//
else if ((xpos==disca_x+1 || xpos==disca_x+2) && ypos==disca_y && memdata != 8'b11111111 && disc_det
 && region_detect == 2'd0 && (pix_detect == 2'd1 || pix_detect == 2'd2))
  begin
     pix_detect <= 2'd0;
     disca_x <= 10'd0;
     disca_y <= 10'd0;
     //region_leds <= 8'b00100000;

  end

     end // always @ (posedge clock)
endmodule // trajectory
```

6.1.4 Calibration Modules

video generation:

```
// generates NTSC encoded video output used for calibration


module video (reset, clock_27mhz, tv_out_ycrcb, tv_out_reset_b, tv_out_clock,
      tv_out_i2c_clock, tv_out_i2c_data, tv_out_pal_ntsc,
      tv_out_hsync_b, tv_out_vsync_b, tv_out_blank_b,
      tv_out_subcar_reset,  mode, up, down, left, right, enable, x_pos, y_pos);

   input reset;
   input clock_27mhz;
   input enable;

   input [2:0] mode;
   input       up, down, left, right;


   output [9:0] tv_out_ycrcb;
   output tv_out_reset_b, tv_out_clock, tv_out_i2c_clock, tv_out_i2c_data,
  tv_out_pal_ntsc, tv_out_hsync_b, tv_out_vsync_b, tv_out_blank_b,
  tv_out_subcar_reset;
   output [10:0] x_pos;
   output [9:0]  y_pos;

   wire colorbars, vbar, hbar;
   assign colorbars = (mode == 0 | mode == 5);
   assign vbar = (mode == 1 | mode == 3);
   assign hbar = (mode == 2 | mode == 4);

   parameter SHEIGHT = 486;
   parameter SWIDTH = 720;

   //
   // Bar generator
   //

   wire [9:0]   y_pos;
   wire [10:0]  x_pos;
   reg [9:0]   ry_pos;
   reg [10:0]  rx_pos;

//   initial ry_pos = 200;
//   initial rx_pos = 300;


   wire [9:0]  hbar_ycrcb, vbar_ycrcb;

   always @(posedge clock_27mhz)
     begin
if(reset)
  begin
     ry_pos <= 20;
     rx_pos <= 20;
  end
else if (enable)
  begin
     if (rx_pos < SWIDTH)
  rx_pos <= rx_pos + 1;
     else
  rx_pos <= 0;
     if (rx_pos < SHEIGHT)
  ry_pos <= ry_pos + 1;
     else
  ry_pos <= 0;
  end
     end

   assign x_pos = rx_pos;
   assign y_pos = ry_pos;


   hbar bar1 (reset, clock_27mhz, hbar_ycrcb,  y_pos);
   defparam bar1.dim = 32;
   vbar bar2 (reset, clock_27mhz, vbar_ycrcb,  x_pos);
   defparam bar2.dim = 32;

   //
   // ADV7194 (Output)
   //

   assign tv_out_clock = clock_27mhz;
   assign tv_out_pal_ntsc = 1'b0;
   assign tv_out_vsync_b = 0;
   assign tv_out_hsync_b = 0;
   assign tv_out_subcar_reset = 0;
   assign tv_out_blank_b = 0;

   adv7194init init7194 (reset, clock_27mhz, colorbars,
 tv_out_reset_b, tv_out_i2c_clock, tv_out_i2c_data);

   assign tv_out_ycrcb = vbar ? vbar_ycrcb :
  (hbar ? hbar_ycrcb : (vbar_ycrcb | hbar_ycrcb));


endmodule
```

---

generate vertical bar:

Note: the module to generate a horizontal bar is not included because it is so similar to this one.

```
module vbar (reset, clock_27mhz, ycrcb, h_pos);

   input reset, clock_27mhz;
   input [10:0] h_pos;
   output [9:0] ycrcb;

   reg [9:0] y, cr, cb;

   reg [10:0] sample_count;
   reg [9:0] line_count;
   reg f, v, h;
   reg [7:0] xy;
   reg [9:0] ycrcb;
   reg [9:0] h_position, v_position;



   always @(negedge clock_27mhz)
     if (reset)
       begin
  sample_count <= 0;
  line_count <= 0;
       end
     else
       begin
  if (sample_count < 1440)
    // Output active video
    // 720 pixels per line,  two samples (Y and C) per sposition
    // Sample order is Cb0, Y0, Cr1, Y1, Cb2, Y2, . . .
    case (sample_count[1:0])
```

```
      2'b00: ycrcb <= cb;
      2'b01: ycrcb <= y;
      2'b10: ycrcb <= cr;
      2'b11: ycrcb <= y;
    endcase
  else if (sample_count == 1440)
    // Start header for the EAV timecode
    ycrcb <= 10'h3FC;
  else if (sample_count == 1441)
    ycrcb <= 10'h000;
  else if (sample_count == 1442)
    ycrcb <= 10'h000;
  else if (sample_count == 1443)
    // EAV timecode
    begin
       ycrcb <= {xy, 2'b00};
       $display("EAV at line %d, sample %d is 0b%B",
line_count, sample_count, xy);
    end
  else if (sample_count < 1712)
    ycrcb <= sample_count[0] ? 10'h200 : 10'h040;
  else if (sample_count == 1712)
    // Begin header for SAV timecode
    ycrcb <= 10'h3fC;
  else if (sample_count == 1713)
    ycrcb <= 10'h000;
  else if (sample_count == 1714)
    ycrcb <= 10'h000;
  else if (sample_count == 1715)
    // SAV timecode
    begin
       ycrcb <= {xy, 2'b00};
       $display("SAV at line %d, sample %d is 0b%B",
line_count, sample_count, xy);
        end

  if (sample_count == 1715)
    begin
       sample_count <= 0;
       if (line_count == 524)
 line_count <= 0;
       else
 line_count <= line_count+1;
    end
  else
    sample_count <= sample_count+1;
       end
```

```
   // Compute the F, V, H, X, and Y bits for the timecodes
   always @(sample_count or line_count)
     begin
f <= (line_count < 9) || (line_count > 271);
v <= (line_count < 19) || ((line_count > 262) && (line_count < 282));
h <= (sample_count > 1439) & (sample_count < 1715);
xy <= {1'b1, f, v, h, v^h, f^h, f^v, f^v^h};
     end

   always @(sample_count or line_count)
     if (line_count < 19)
       begin
  h_position <= 0;
  v_position <= 0;
       end
     else if (line_count < 262)
       begin
  h_position <= (sample_count[10:1] < 719) ?
       sample_count[10:1]+sample_count[0] : 0;
  v_position <= {line_count-19, 1'b1};
       end
     else if (line_count < 282)
       begin
  h_position <= 0;
  v_position <= 0;
       end
     else
       begin
  h_position <= (sample_count[10:1] < 719) ?
sample_count[10:1]+sample_count[0] : 0;
  v_position <= {line_count-281, 1'b0};
       end

   parameter SHEIGHT = 486;
   parameter SWIDTH = 720;

   parameter dim = 64;

   always @(h_position or v_position or h_pos)
     if ((h_position > h_pos) && (h_position < h_pos+dim)
 && (v_position > 0) && (v_position < SHEIGHT))
       begin
  y <= 512;
  cr <= 512;
  cb <= 512;

       end
     else
```

```
       begin
  y <= 0;
  cr <= 512;
  cb <= 512;
       end

endmodule
```

---

hit detection:

```
// hit detection module

// inputs: optical signal from light gun
//         reset
//         27mhz clock
//         threshold value
// outputs: hit detect signal

module hit_detect(reset, clock_27mhz, hit, optical, enable);

   input clock_27mhz, reset, enable;
   input optical;

   output      hit;

   parameter   thresh = 20000;


   reg [15:0]  count;
   reg         hit;


   always @(posedge clock_27mhz)
     begin
if (reset)
  begin
     hit <= 0;
     count <= 0;
  end
else if (optical & enable)
  begin
     count <= count + 1;
     if (count == thresh)
       hit <= 1;
     // hit is high for one clock pulse only
  end
else
```

```
  begin
     count <= 0;
     hit <= 0;
  end
  end // always @ (posedge clock_27mhz)

endmodule
```

---

hit location:

```
// detect hit location using hit output from hit_detect
// and video generator outputs

module hit_location(clock_27mhz, reset, video_mode,
      hit, bar_x_pos, bar_y_pos, x_pos, y_pos, x_there, y_there);

   input reset;
   input clock_27mhz;
   input [2:0] video_mode;
   input [10:0] bar_x_pos;
   input [9:0]  bar_y_pos;
   input  hit;

   output [10:0] x_pos;
   output [9:0]  y_pos;
   output   x_there;
   output   y_there;


   wire   vbar = (video_mode == 1 | video_mode == 3) ? 1 : 0;
   wire   hbar = (video_mode == 2 | video_mode == 4) ? 1 : 0;

   reg [10:0]   x_pos;
   reg [9:0]   y_pos;

   reg   x_there;
   reg   y_there;

   // save hit positions
   always @(posedge clock_27mhz)
     begin
if (reset)
  begin
     x_pos <= 0;
     y_pos <= 0;
  end
else if(hit & vbar) // detected hit at horizontal position
```

```
  begin
     x_pos <= bar_x_pos;
     x_there <= 1;
  end
else if(hit & hbar) // detected hit at vertical position
  begin
     y_pos <= bar_y_pos;
     y_there <= 1;
  end
else
  begin
     x_there <= 0;
     y_there <= 0;
  end
     end // always @ (posedge clock_27mhz)


endmodule
```

---

calibrator:

```
// calibrator FSM
// outputs: x_offset[10:0], y_offset[9:0] (SIGNED offsets)
//          x_aim[10:0], y_aim[9:0]
//          detect_mode, detect_reset
//          done
// inputs:  x_hit[10:0], y_hit[9:0]
//          enable

module calibrator (clock_27mhz, reset, enable, x_offset, y_offset,
   x_aim, y_aim, video_mode, hit_enable, done,
   x_hit, y_hit, x_there, y_there, video_state,
                   b0, aim_point_x, aim_point_y);

   input clock_27mhz, reset, enable;
   input [10:0] x_hit;
   input [9:0]  y_hit;
   input  x_there, y_there;
   input  b0;
   input [1:0]  aim_point_x;
   input [1:0]  aim_point_y;



   output [10:0] x_offset, x_aim;
   output [9:0]  y_offset, y_aim;
   output   hit_enable;
   output [2:0]  video_mode;
   output   done;
   output [2:0]  video_state;



   reg [1:0]   detect_mode;
   reg [10:0] x_aim;
   reg [9:0]  y_aim;

   reg [2:0]  pass;

   reg [2:0]  video_state;
   reg [2:0]  video_mode;
   reg        hit_enable;

   // FSM
   always @(posedge clock_27mhz)
     begin
if (reset) //reset
  begin
     video_state <= 0;
     video_mode <= 3'b001;
     hit_enable <= 1;
     pass <= 0;
     x_aim <= 0;
     y_aim <= 0;
  end
else
  case (video_state)
    3'b000 : video_state <=  3'b001;
    3'b001: begin // x detect mode
       video_mode <= 1;
       video_state <= x_there ? 3'b010 : 3'b001;
       hit_enable <= 1;
    end
    3'b010: begin // x wait mode
       video_state <= b0 ? 3'b011 : 3'b010; // wait some.
       hit_enable <= 0;
    end
    3'b011: begin // y detect mode
       video_mode <= 2;
       video_state <= y_there ? 3'b100 : 3'b011;
       hit_enable <= 1;
    end
    3'b100: begin // y wait mode
       pass <= pass + 1; // end of this pass
       video_state <= b0 ? 3'b101 : 3'b100;
       hit_enable <= 0;
    end
    3'b101: begin // start over
       video_state <= 3'b001;
    end
  endcase // case(video_state)
x_aim <= 180*aim_point_x;
y_aim <= 114*aim_point_y;

     end



   // indicate when done, enabling with high pulse
   // this didn't actually get used.
   assign done = (video_state == 5) ? 1 : 0;

   reg signed [10:0] x_offset;
   reg signed [9:0]  y_offset;


   // calculate offsets:

   always @(x_there)
      x_offset = x_aim - x_hit;
   always @(y_there)
      y_offset = y_aim - y_hit;

endmodule
```

6.1.5 Servomechanism Modules

coordinate conversion:

```
// coordinate conversion module
// inputs: x_coord and y_coord (from Main FSM)
//         x_offset and y_offset (from calibration FSM, already SIGNED)
//         tv_distance (switches on labkit)
// outputs: h_coord (horizontal, ie bottom motor, angle)
//          v_coord (vertical, ie top mottor, angle)

// offsets will be signed, so need to sign x,y coord
// position = coord + offset
// compare position to center of screen, set right/left and up/down bits
// take absolute value of difference
// convert to larger blocks, resolution = 10 blocks / half screen.
// look up address is {distance, pos_diff}
// output coord  is (output of ROM + neutral coord value)
```

```
module coord_conversion(x_coord, y_coord, x_offset, y_offset, tv_distance,
h_coord, v_coord, clock, reset);
   input clock, reset;
   input [10:0] x_coord;      // unsigned target coordinates from main FSM
   input [9:0]  y_coord;
   input [10:0] x_offset;     // signed offset values from calibration FSM
   input [9:0]  y_offset;
   input [3:0]  tv_distance;  // (debounced) switch inputs from labkit

   output [7:0] h_coord, v_coord;  // signals for the motor drivers

   parameter  signed  SWIDTH = 1024;
   parameter  signed  SHEIGHT = 768;
//   parameter  signed SWIDTH = 768;
//   parameter  signed SHEIGHT = 384;

   parameter  XRES = 16; // each half-screen is divided into XRES by YRES blocks
   parameter  YRES = 12;


   // sign extend the (always positive) x and y coords.
   // do i need to extend the offsets too??
   wire signed [11:0]  sx_coord, sx_offset;
   wire signed [10:0]  sy_coord, sy_offset;
   assign      sx_coord = {x_coord[10], x_coord};
   assign      sy_coord = {y_coord[9], y_coord};
   assign      sx_offset = {x_offset[10], x_offset};
   assign      sy_offset = {y_offset[9], y_offset};


   // calculate position to aim at:
   wire signed [11:0] x_pos;
   wire signed [10:0] y_pos;


   assign y_pos = sy_coord+sy_offset;
   assign x_pos = sy_coord+sx_offset;

   reg        rhp, thp; // right-half-plane and top-half-plane indicators.
                        // when rhp = 1, it the target is in the right half
                        // of the screen.

   reg signed [10:0]  x_diff;  // these are signed too now, but values should always be positive.
   reg signed [9:0]  y_diff; // the maximum values are half of the max for x_coord and y_coord
   wire [3:0] x_block_tmp, y_block_tmp; // indicate which of the "chunks" the target is in.
   wire [3:0] x_block, y_block;// ultimately part of the lookup address
   always @(x_pos)
     begin
if (x_pos >= SWIDTH/2) // target is in right half of screen
  begin
     rhp = 1;
     x_diff = x_pos - SWIDTH/2;
  end
else
  begin
     rhp = 0;
     x_diff = SWIDTH/2 - x_pos;
  end
     end // always @ (x_pos)

   always @(y_pos)
     begin
if (y_pos >= SHEIGHT/2) // target is in bottom half of screen.
  // y_pos INCREASES as it moves DOWN!
  begin
     thp = 0;
     y_diff = y_pos - SHEIGHT/2;
  end
else
  begin
     thp = 1;
     y_diff = SHEIGHT/2 - y_pos;
  end
     end // always @ (y_pos)


   assign x_block_tmp = (x_diff  - x_diff%(SWIDTH/(2*XRES)))/(SWIDTH/(2*XRES))+rhp;
   assign y_block_tmp = (y_diff - y_diff%(SHEIGHT/(2*YRES)))/(SHEIGHT/(2*YRES))+thp;


   //make sure blocks are within bounds
   assign x_block = (x_block_tmp < XRES) ? x_block_tmp : XRES - 1;
   assign y_block = (y_block_tmp < YRES) ? y_block_tmp : YRES - 1;


   wire [7:0] x_addr, y_addr;
   assign     x_addr = {tv_distance,x_block};
   assign     y_addr = {tv_distance,y_block};

   wire [7:0] h_ram, v_ram;

   x_ram xram(x_addr, clock, h_ram);
   y_ram yram(y_addr, clock, v_ram);
   parameter  h_neutral = 152;
   parameter  v_neutral = 100;

   assign     h_coord = rhp ? (h_neutral + h_ram) : (h_neutral - h_ram);
   assign     v_coord = thp ? (v_neutral + v_ram) : (v_neutral - v_ram);



endmodule
```

TV coordinate conversion:

```
// coordinate conversion module for TV
// inputs: x_coord and y_coord (from Main FSM)
//         x_offset and y_offset (from calibration FSM, already SIGNED)
//         tv_distance (switches on labkit)
// outputs: h_coord (horizontal, ie bottom motor, angle)
//          v_coord (vertical, ie top mottor, angle)

// offsets will be signed, so need to sign x,y coord
// position = coord + offset
// compare position to center of screen, set right/left and up/down bits
// take absolute value of difference
// convert to larger blocks, resolution = 10 blocks / half screen.
// look up address is {distance, pos_diff}
// output coord  is (output of ROM + neutral coord value)

module coord_conversion_tv(x_coord, y_coord, x_offset, y_offset, tv_distance,
h_coord, v_coord, clock, reset, x_block, y_block);
   input clock, reset;
   input [10:0] x_coord;      // unsigned target coordinates from main FSM
   input [9:0]  y_coord;
   input [10:0] x_offset;     // signed offset values from calibration FSM
   input [9:0]  y_offset;
   input [3:0]  tv_distance;  // (debounced) switch inputs from labkit

   output [7:0] h_coord, v_coord;  // signals for the motor drivers
   output [3:0] x_block, y_block;

   parameter  signed  SWIDTH = 720;
   parameter  signed  SHEIGHT = 486;

   parameter  XRES = 16; // each half-screen is divided into XRES by YRES blocks
   parameter  YRES = 12;
   // sign extend the (always positive) x and y coords.
   // do i need to extend the offsets too??
   wire signed [11:0]  sx_coord, sx_offset;
   wire signed [10:0]  sy_coord, sy_offset;
   assign      sx_coord = {x_coord[10], x_coord};
   assign      sy_coord = {y_coord[9], y_coord};
   assign      sx_offset = {x_offset[10], x_offset};
   assign      sy_offset = {y_offset[9], y_offset};


   // calculate position to aim at:
   wire signed [11:0] x_pos;
   wire signed [10:0] y_pos;


   assign y_pos = sy_coord+sy_offset;
   assign x_pos = sy_coord+sx_offset;

   reg        rhp, thp; // right-half-plane and top-half-plane indicators.
                        // when rhp = 1, it the target is in the right half
                        // of the screen.

   reg signed [10:0]  x_diff;  // these are signed too now, but values should always be positive.
   reg signed [9:0]  y_diff; // the maximum values are half of the max for x_coord and y_coord
   // x_diff and y_diff refer to the distance of the position from the center of the screen
   // they are used to determine which block the coords are in
   reg [3:0] x_block, y_block;// ultimately part of the lookup address


   always @(x_pos)
     begin
if (x_pos >= SWIDTH/2) // target is in right half of screen
  begin
     rhp = 1;
     x_diff = x_pos - SWIDTH/2;
  end
else
  begin
     rhp = 0;
     x_diff = SWIDTH/2 - x_pos;
  end
     end // always @ (x_pos)

   always @(y_pos)
     begin
if (y_pos >= SHEIGHT/2) // target is in bottom half of screen.
  // y_pos INCREASES as it moves DOWN!
  begin
```

```
     thp = 0;
     y_diff = y_pos - SHEIGHT/2;
  end
else
  begin
     thp = 1;
     y_diff = SHEIGHT/2 - y_pos;
  end
     end // always @ (y_pos)


   // determine x block:
   // (add rhp or thp so that there aren't two zero blocks at the center)
   always @(x_diff or rhp)
     begin
if (x_diff < 45)
  x_block = 0 + rhp;
else if (x_diff < 2*45)
  x_block = 1 + rhp;
else if (x_diff < 3*45)
  x_block = 2 + rhp;
else if (x_diff < 4*45)
  x_block = 3 + rhp;
else if (x_diff < 5*45)
  x_block = 4 + rhp;
else if (x_diff < 6*45)
  x_block = 5 + rhp;
else if (x_diff < 7*45)
  x_block = 6 + rhp;
else if (x_diff < 8*45)
  x_block = 7 + rhp;
else if (x_diff < 9*45)
  x_block = 8 + rhp;
else if (x_diff < 10*45)
  x_block = 9 + rhp;
else if (x_diff < 11*45)
  x_block = 10 + rhp;
else if (x_diff < 12*45)
  x_block = 11 + rhp;
else if (x_diff < 13*45)
  x_block = 12 + rhp;
else if (x_diff < 14*45)
  x_block = 13 + rhp;
else if (x_diff < 15*45)
  x_block = 14 + rhp;
else if (x_diff < 16*45)
  x_block = 15;
     end
   always @(y_diff or thp)
     begin
if (y_diff < 40)
  y_block = 0 + thp;
else if (y_diff < 2*40)
  y_block = 1 + thp;
else if (y_diff < 3*40)
  y_block = 2 + thp;
else if (y_diff < 4*40)
  y_block = 3 + thp;
else if (y_diff < 5*40)
  y_block = 4 + thp;
else if (y_diff < 6*40)
  y_block = 5 + thp;
else if (y_diff < 7*40)
  y_block = 6 + thp;
else if (y_diff < 8*40)
  y_block = 7 + thp;
else if (y_diff < 9*40)
  y_block = 8 + thp;
else if (y_diff < 10*40)
  y_block = 9 + thp;
else if (y_diff < 11*40)
  y_block = 10 + thp;
else
  y_block = 11;
     end


   wire [7:0] x_addr, y_addr;
   assign     x_addr = {tv_distance,x_block};
   assign     y_addr = {tv_distance,y_block};

   wire [7:0] h_ram, v_ram;

   x_ram xram(x_addr, clock, h_ram);
   y_ram yram(y_addr, clock, v_ram);

   parameter  h_neutral = 152;
   parameter  v_neutral = 100;

   assign     h_coord = rhp ? (h_neutral + h_ram) : (h_neutral - h_ram);
   assign     v_coord = thp ? (v_neutral + v_ram) : (v_neutral - v_ram);



endmodule
```

6.1.6 *Duckhunter* and Integration

This is a truncated version of the `duckhunter` module. It includes only the additions made to the standard `labkit.v` file that can be found at `http://www-mtl.mit.edu/Courses/6.111/labkit/verilog/labkit.v`

```
////////////////////////////////////////////////////////////////////////////
//
// DUCKHUNTER
//
////////////////////////////////////////////////////////////////////////////

// use FPGA's digital clock manager to produce a
// 65MHz clock (actually 64.8MHz)
wire clock_65mhz_unbuf,clock_65mhz;
DCM vclk1(.CLKIN(clock_27mhz),.CLKFX(clock_65mhz_unbuf));
// synthesis attribute CLKFX_DIVIDE of vclk1 is 10
// synthesis attribute CLKFX_MULTIPLY of vclk1 is 24
// synthesis attribute CLK_FEEDBACK of vclk1 is NONE
// synthesis attribute CLKIN_PERIOD of vclk1 is 37
BUFG vclk2(.O(clock_65mhz),.I(clock_65mhz_unbuf));

// power-on reset generation
wire power_on_reset;    // remain high for first 16 clocks
SRL16 reset_sr (.D(1'b0), .CLK(clock_65mhz), .Q(power_on_reset),
.A0(1'b1), .A1(1'b1), .A2(1'b1), .A3(1'b1));
defparam reset_sr.INIT = 16'hFFFF;

// ENTER button is user reset
wire reset,user_reset;
wire [7:0] switches;
debounce db0(power_on_reset, clock_65mhz, ~button_enter, b_enter);
debounce db1(power_on_reset, clock_65mhz, ~button_up, up);
debounce db2(power_on_reset, clock_65mhz, ~button_down, down);
debounce db3(power_on_reset, clock_65mhz, ~button_right, right);
debounce db4(power_on_reset, clock_65mhz, ~button_left, left);
debounce db5(power_on_reset, clock_65mhz, switch[0], switches[0]);
debounce db6(power_on_reset, clock_65mhz, switch[1], switches[1]);
debounce db7(power_on_reset, clock_65mhz, switch[2], switches[2]);
debounce db8(power_on_reset, clock_65mhz, switch[3], switches[3]);
debounce db9(power_on_reset, clock_65mhz, switch[4], switches[4]);
debounce db10(power_on_reset, clock_65mhz, switch[5], switches[5]);
debounce db11(power_on_reset, clock_65mhz, switch[6], switches[6]);
debounce db12(power_on_reset, clock_65mhz, switch[7], switches[7]);
debounce db13(power_on_reset, clock_65mhz, ~button0, b0);
debounce db14(power_on_reset, clock_65mhz, ~button1, b1);
debounce db15(power_on_reset, clock_65mhz, ~button2, b2);
debounce db16(power_on_reset, clock_65mhz, ~button3, b3);
```

```
   assign reset =  power_on_reset | b_enter;

   divider n_hz(clock_27mhz, n_hz_enable, reset_sync);
   defparam n_hz.DELAY=84375; // =27000000/320

   // Set Mode using labkit switches:
   reg [7:0] mode;

   always @(posedge clock_27mhz)
     if (b_enter & b0)
       mode <= switches;

   wire trigger_demo, cc_demo, cc_pong_demo, servo_demo, laser_demo,
video_output_demo, light_detect_demo, hit_location, aj_mode,
calibration_mode, c_mode_0, c_mode_1, c_mode_2, c_mode_3,
duckhunter;

   assign duckhunter = (mode == 0) ? 1 : 0;
   // DUCKHUNTER mode!
   assign    trigger_demo = (mode == 1) ? 1 : 0;
   // Mode 1: Demo trigger functionality. This demo uses button_0 on
   // labkit to "pull" light gun trigger.
   assign cc_demo = (mode == 2) ? 1 : 0;
   // Mode 2: Coordinate Conversion/Motor Drive Demo.
   // Uses switch[0] to scan target across screen (VGA)
   // watch servos track the same path.
   assign cc_pong_demo = (mode == 3) ? 1 : 0;
   // Mode 3: Coordinate Conversion Demo.
   // Servos track the puck from Andrew Lisy's Pong Game (6.111 Lab 4)
   // Use switch[7:4] to set puck speed, button_up and button_down
   // to move paddle.
   assign servo_demo = (mode == 4) ? 1 : 0;
   // Mode 4: Servomechanism Demo
   // use up/down and left/right buttons to move servos
   // can also shoot with button0 to play duckhunt
   assign laser_demo = (mode == 5) ? 1 : 0;
   // Mode 5: Laser Demo
   // Uses button0 to turn on laser
   // Laser pointer is currently broken.
   assign video_output_demo = (mode == 6) ? 1 : 0;
   // Mode 6: Outputs moving bars for calibration
   // Uses switches[1:0]:
   //    00: color bars
   //    01: vertical bar
   //    10: horizontal bar
   //    11: both bars
   assign light_detect_demo = (mode == 7) ? 1 : 0;
   // Mode 7: Light Detection
   // Uses button0 as enable. If enabled, outputs high pulse
   // to LEDs and logic analyzer 1 if hit detected.
   assign hit_location = (mode == 8) ? 1 : 0;
   assign aj_mode = (mode == 9) ? 1 : 0;
   // test AJ's stuff.
   assign calibration_mode = (mode == 255)? 1:0;
   // for display stuff:
   assign c_mode_0 = (calibration_mode & (switches[7:6] == 0)) ? 1 : 0; // "CALIBRATION MODE"
   assign c_mode_1 = (calibration_mode & (switches[7:6] == 1)) ? 1 : 0; // calculated offset values
   assign c_mode_2 = (calibration_mode & (switches[7:6] == 2)) ? 1 : 0; // hit position
   assign c_mode_3 = (calibration_mode & (switches[7:6] == 3)) ? 1 : 0; // input offset values


   // generate basic XVGA video signals:
   wire [10:0] hcount;
   wire [9:0]  vcount;
   wire hsync,vsync,blank;
   xvga xvga1(clock_65mhz,hcount,vcount,hsync,vsync,blank);


   // Test Hit Detection (Mode 7):
   reg        optical;
   reg        pos_edge, neg_edge;

   always @(posedge clock_27mhz)
     begin
optical <= user1[0];
pos_edge <= user1[0] & ~optical;
neg_edge <= ~user1[0] & optical;
     end

   wire [9:0] video_bars_ycrcb;


   wire [2:0] video_mode;

   wire [10:0] bar_x_pos;
   wire [9:0]  bar_y_pos;

   wire       hit;

   wire [10:0] hit_x_pos;
   wire [9:0]  hit_y_pos;

   wire [10:0] x_offset;
   wire [10:0] x_aim;
   wire [9:0]  y_offset;
```

```
  wire [9:0]  y_aim;
  wire [1:0]  detect_mode;
  wire        detect_reset;
  wire        done;
  wire        y_there, x_there;

  // use this video to detect hits:
  hit_detect hd(b0, clock_27mhz, hit, optical, hit_enable);
  // detect hit position:
  hit_location hl(clock_27mhz, reset, video_mode, hit,
  bar_x_pos, bar_y_pos, hit_x_pos, hit_y_pos, x_there, y_there);

  video bars(reset, clock_27mhz, video_bars_ycrcb, tv_out_reset_b, tv_out_clock,
     tv_out_i2c_clock, tv_out_i2c_data, tv_out_pal_ntsc,
     tv_out_hsync_b, tv_out_vsync_b, tv_out_blank_b,
     tv_out_subcar_reset,  video_mode, up, down, left, right, n_hz_enable,
     bar_x_pos, bar_y_pos);

  // calibrate based on hit position:

  wire [1:0]  video_mode_out;
  wire [2:0]  video_state;

  calibrator calibrate (clock_27mhz, reset, 1'b1, x_offset, y_offset,
x_aim, y_aim, video_mode, hit_enable, done,
hit_x_pos, hit_y_pos, x_there, y_there,
video_state, b0, switches[1:0], switches[3:2]);



  assign      tv_out_ycrcb =  video_bars_ycrcb;

  wire [2:0]  blob_pixel, blob1_pixel, blob2_pixel;

  blob blob1(x_aim, y_aim, hcount, vcount, blob1_pixel);
  blob blob2(hit_x_pos, hit_y_pos, hcount, vcount, blob2_pixel);
  defparam    blob2.COLOR = 3'b100;
  // draw outline of screen:
  wire [2:0]  blobs1_pixel, blobs2_pixel, blobs3_pixel, blobs4_pixel;

  blob blobs1(11'b0, 10'b0, hcount, vcount, blobs1_pixel);
  defparam    blobs1.WIDTH = 4;
  defparam    blobs1.HEIGHT = 486;
  blob blobs2(11'd716, 10'b0, hcount, vcount, blobs2_pixel);
  defparam    blobs2.WIDTH = 4;
  defparam    blobs2.HEIGHT = 486;
  blob blobs3(11'b0, 10'b0, hcount, vcount, blobs3_pixel);
  defparam    blobs3.WIDTH = 720;
```

```
   defparam    blobs3.HEIGHT = 4;
   blob blobs4(11'b0, 10'd482, hcount, vcount, blobs4_pixel);
   defparam    blobs4.WIDTH = 720;
   defparam    blobs4.HEIGHT = 4;


   assign      blob_pixel = blob1_pixel | blob2_pixel |
       blobs1_pixel | blobs2_pixel | blobs3_pixel | blobs4_pixel;


//////////// MOTOR STUFF /////////////

   wire [2:0] pixel_target, pixel, pixel_pong;
   wire [10:0] x_coord_target, x_coord_pong;
   wire [9:0]  y_coord_target, y_coord_pong;
   wire [7:0]   h_coord, v_coord;
   wire [7:0] h_coord_cc, v_coord_cc;
   wire [3:0]  x_block_tv, y_block_tv;
   wire [7:0]  h_coord_tv, v_coord_tv;


   // for motor:

   reg         signed [10:0] x_off;
   reg         signed [9:0] y_off;

   initial x_off = 0;
   initial y_off = 0;

// Generate target for standard coordinate conversion demo (mode 2):
   test target(b_enter, x_coord_target, y_coord_target, clock_65mhz,
            vsync, hcount, vcount, pixel_target, switches[0]);
// Generate target for pong game coordinate conversion demo (mode 3):
   pong_game  pg(clock_27mhz,b_enter,up,down,switches[7:4],
  hcount,vcount,hsync,vsync,blank,
  pixel_pong, x_coord_pong, y_coord_pong);


   wire [10:0] aj_x_pos;
   wire [9:0]  aj_y_pos;


   // AJ MODE (Mode 8):

   wire [7:0] aj_red, aj_green, aj_blue;
   wire       aj_vga_out_sync_b, aj_vga_out_blank_b, aj_vga_out_pixel_clock,
      aj_vga_out_hsync, aj_vga_out_vsync;
   wire [7:0] ledbank;
   wire [63:0] hexbank;
   wire        aj_trigger;


  detection_toplevel aj(reset, clock_27mhz, up, down, ledbank,
   aj_red, aj_green, aj_blue,
   aj_vga_out_sync_b, aj_vga_out_blank_b, aj_vga_out_pixel_clock,
   aj_vga_out_hsync, aj_vga_out_vsync,
   tv_in_reset_b, tv_in_i2c_clock, tv_in_i2c_data,
   tv_in_ycrcb, switches, tv_in_line_clock1, aj_x_pos, aj_y_pos, aj_trigger);


   reg [10:0]  x_coord;
   reg [9:0]   y_coord;


// manually generate offsets:
   always @(posedge clock_27mhz)
     if (b3) // reset offset values
       begin
  y_off <= 0;
  x_off <= 0;
       end
     else if(up)
       y_off <= y_off - 1;
     else if (down)
       y_off <= y_off + 1;
     else if (right)
       x_off <= x_off + 1;
     else if (left)
       x_off <= x_off - 1;

   wire   signed [10:0] cc_x_off;
   wire   signed [9:0] cc_y_off;

   assign cc_x_off = calibration_mode ? x_offset+x_off : 10'b0;
   assign cc_y_off = calibration_mode ? y_offset+y_off : 9'b0;


   // Set distance of gun from lab kit using switches[3:0]
   // and buttons 1&2
   // make default reasonable: 4'b0110

   reg [3:0] distance;
   initial distance = 4'b0110;
   always @(posedge clock_27mhz)
```

```
    if (b1 & b2)
      distance <= switches[3:0];







  // Test Motors separately (mode 4):
  wire [7:0] h_pw, v_pw;
  driver_test test_driver(clock_27mhz, reset, ~switches[7:4],
  ~switches[3:0], up, down, left, right, h_pw, v_pw);


 always @(mode)
    case (mode)
      2: begin
 x_coord = x_coord_target;
 y_coord = y_coord_target;
      end
      3: begin
 x_coord = x_coord_pong;
 y_coord = y_coord_pong;
      end
      9: begin
 x_coord = aj_x_pos;
 y_coord = aj_y_pos;
      end
      default: begin
 x_coord = x_aim;
 y_coord = y_aim;
      end
    endcase



  coord_conversion_tv converts_tv(x_coord, y_coord, cc_x_off, cc_y_off, distance,
  h_coord_tv, v_coord_tv, clock_27mhz, reset, x_block_tv, y_block_tv);
  coord_conversion converts(x_coord_target, y_coord_target, 11'b0, 10'b0, distance,
    h_coord_cc, v_coord_cc, clock_27mhz, reset);

  always @(mode)
    case(mode)
      2: begin
 h_coord = h_coord_cc;
 v_coord = v_coord_cc;
      end
      3: begin
  h_coord = h_coord_cc;
  v_coord = v_coord_cc;
       end
       4: begin
  h_coord = h_pw;
  v_coord = v_pw;
       end
       default: begin
  h_coord = h_coord_tv;
  v_coord = v_coord_tv;
       end
     endcase



   rc_motor_move top (clock_27mhz, v_coord, top_motor);
   rc_motor_move bottom (clock_27mhz, h_coord, bottom_motor);

   // LASER POINTER DEMO (Mode 5):
   reg       laser_green = laser_demo ? ~b1 : 1'bZ;

   // TRIGGER DEMO (Mode 1):
   wire       trigger = b0;

   // SET LEDS
   assign      led =  light_detect_demo ? ~{hit, hit, hit, hit, hit, hit, hit, hit} :
       (aj_mode ? ledbank :
(calibration_mode ? ~{5'b0, video_state} : ~mode));

   // SET LED DISPLAY - for coolness.
   wire [16*8-1:0] disp_string;

   wire     disp_blank_tmp, disp_clock_tmp;
   wire     disp_rs_tmp, disp_ce_b_tmp, disp_reset_b_tmp, disp_data_out_string;
   wire     disp_blank_tmp1, disp_clock_tmp1;
   wire     disp_rs_tmp1, disp_ce_b_tmp1, disp_reset_b_tmp1, disp_data_out_hex;


// TEXT Display
   assign     disp_string = trigger_demo ? "B0 = Trigger    " :
   (cc_demo ? "COORD CONVERSION" :
    (cc_pong_demo ? "PONG DEMO       " :
     (laser_demo ? "B1=LASER POINTER" :
      (video_output_demo ? "CALIB VIDEO OUT " :
       (light_detect_demo ? "LIGHT DETECTION " :
(aj_mode ? "INTERPOLATION   " :
 (calibration_mode ? "CALIBRATION     " :"   DUCKHUNTER   ")))))));
// HEX Display
```

```
   wire [63:0]     motor_hex = servo_demo ? {24'b0, h_pw, 24'b0, v_pw} :
   (aj_mode ? hexbank :
    ((hit_location | c_mode_3) ? {21'b0, hit_x_pos, 22'b0, hit_y_pos} :
     (c_mode_0 ? {21'b0, x_offset, 22'b0, y_offset} : 64'b0)));



   display_string hex_display(reset, clock_27mhz, disp_string,
disp_blank_tmp, disp_clock_tmp, disp_rs_tmp, disp_ce_b_tmp,
disp_reset_b_tmp, disp_data_out_string);

   display_16hex hex_display1(reset, clock_27mhz, motor_hex,
disp_blank_tmp1, disp_clock_tmp1, disp_rs_tmp1, disp_ce_b_tmp1,
disp_reset_b_tmp1, disp_data_out_hex);

   assign     disp_blank = servo_demo ? disp_blank_tmp1 : disp_blank_tmp;
   assign     disp_clock = servo_demo ? disp_clock_tmp1 : disp_clock_tmp;
   assign     disp_rs = servo_demo ? disp_rs_tmp1 : disp_rs_tmp;
   assign     disp_ce_b = servo_demo ? disp_ce_b_tmp1 : disp_ce_b_tmp;
   assign     disp_reset_b = servo_demo ? disp_reset_b_tmp1 : disp_reset_b_tmp;
   assign     disp_data_out = (servo_demo | hit_location | calibration_mode) ?
     disp_data_out_hex : disp_data_out_string;


   // VGA Output.
   // tbarton's xvga out:

   assign pixel = (cc_demo == 1) ? pixel_target :
  (cc_pong_demo ? pixel_pong :
   (calibration_mode ? blob_pixel : 0));

   reg [2:0] rgb;
   reg b,hs,vs;
   always @(posedge clock_65mhz)
     begin
        hs <= hsync;
vs <= vsync;
b <= blank;
rgb <= pixel;
     end


   // combine with alisy's:
   assign vga_out_red = aj_mode ? aj_red : {8{rgb[2]}};
   assign vga_out_green = aj_mode ? aj_green : {8{rgb[1]}};
   assign vga_out_blue = aj_mode ? aj_blue : {8{rgb[0]}};
   assign vga_out_sync_b = aj_mode ? aj_vga_out_sync_b : 1'b1;
   assign vga_out_blank_b = aj_mode ? aj_vga_out_blank_b : ~b;
   assign vga_out_pixel_clock = aj_mode ? aj_vga_out_pixel_clock : ~clock_65mhz;
   assign vga_out_hsync = aj_mode ? aj_vga_out_hsync : hs;
   assign vga_out_vsync = aj_mode ? aj_vga_out_vsync : vs;

   // USER INPUT and OUTPUT:

   assign user1 = {32'hZ};

   assign trigger_out =  aj_trigger | trigger;


   assign user3 = {top_motor, bottom_motor, 28'hZ, laser_green, trigger_out};

   assign analyzer1_clock = clock_27mhz;
   assign analyzer1_data = {15'b0, hit};

   assign analyzer2_clock = clock_27mhz;
   assign analyzer2_data = {13'b0, video_state};

endmodule
```

6.1.7 Demo Modules

6.1.8 Demo Modules

coordinate conversion demo (demo mode 2):

```
// test module for coordinate conversion

module test(reset, x_coord, y_coord, clock, vsync, hcount, vcount, pixel, enable);
   input reset, clock;
   input vsync;
   input [10:0] hcount;
   input [9:0]  vcount;
   input  enable;

   output [10:0] x_coord; // use these as inputs to converter and to blob
   output [9:0]  y_coord;
   output [2:0]  pixel;


   parameter   SWIDTH = 1024;
   parameter   SHEIGHT = 768;
   parameter   TDIM = 64; // target dimension


   reg [10:0]   x_coord;
   reg [9:0]   y_coord;
   //Look for frame transitions
   reg old_pvsync;
   wire new_frame;

   always @ (posedge clock) old_pvsync <= vsync;
   assign new_frame = ~old_pvsync & vsync;  //start of a new frame

   always @(posedge clock)
     begin
if (reset)
  begin
     x_coord <= 0;
     y_coord <=0;
  end
else if (new_frame & enable)
  if(x_coord < SWIDTH-TDIM)
    x_coord <= x_coord + 2;
  else
    begin
       x_coord <= 0;

       if (y_coord < SHEIGHT-TDIM)
 y_coord <= y_coord + 64;
       else
 y_coord <= 0;
    end // else: !if(x_coord < SWIDTH)
     end // always @ (posedge clock)

   blob target(x_coord,y_coord, hcount, vcount, pixel);
   defparam target.WIDTH = TDIM;
   defparam target.HEIGHT = TDIM;


endmodule // test
```

---

pong game (demo mode 3), by alisy:

```
////////////////////////////////////////////////////////////////////////////////
//
// Pong Game
//
// By: Andrew Lisy (alisy)
// 6.111 - Digital Electronics Lab
// October 24, 2005
//
////////////////////////////////////////////////////////////////////////////////
```

```
module pong_game (vclock,reset,up,down,pspeed,
  hcount,vcount,hsync,vsync,blank,
  pixel, puck_h, puck_v);
   input vclock; // 65MHz clock
   input reset; // 1 to initialize module
   input up; // 1 when paddle should move up
   input down; // 1 when paddle should move down
   input [3:0] pspeed;  // puck speed in pixels/tick
   input [10:0] hcount; // horizontal index of current pixel (0..1023)
   input [9:0]  vcount; // vertical index of current pixel (0..767)
   input hsync; // XVGA horizontal sync signal (active low)
   input vsync; // XVGA vertical sync signal (active low)
   input blank; // XVGA blanking (1 means output black pixel)

   output [2:0] pixel; // pong game's pixel
   output [10:0] puck_h;
   output [9:0]  puck_v;


   assign phsync = hsync;
   assign pvsync = vsync;
   assign pblank = blank;

   //////////////////////////
   /////Create Elements//////
   //////////////////////////

   ///Draw box:
   wire box_draw; //high when the border should be lit
   assign box_draw = (vcount == 0 || vcount == 767 || hcount == 0 || hcount == 1023);

   ///Draw puck
   reg [10:0] puck_h;
   reg [9:0] puck_v;
   initial puck_h = 400;    //start about 1/3 across and 1/7 down the screen
   initial puck_v = 100;
   wire [2:0] puck_pixel;
   blob puck_1(puck_h, puck_v, hcount, vcount, puck_pixel);
   defparam puck_1.WIDTH= 64;
   defparam puck_1.HEIGHT = 64;
   defparam puck_1.COLOR = 3'b111;  //white

   //Draw paddle
   reg [9:0] paddle_v;
   initial paddle_v = 400;  //start out just over halfway down the screen
   wire [2:0] paddle_pixel;
   blob paddle_1(11'd0, paddle_v, hcount, vcount, paddle_pixel);
```

```
 defparam paddle_1.WIDTH=16;
 defparam paddle_1.HEIGHT = 128;
 defparam paddle_1.COLOR = 3'b100;  //red

 //Look for frame transitions
 reg old_pvsync;
 wire new_frame;

 always @ (posedge vclock) old_pvsync <= pvsync;
 assign new_frame = ~old_pvsync & pvsync;  //start of a new frame


 //Initial puck velocities
 wire [3:0] puck_vel_h;
 assign puck_vel_h = pspeed;
 wire [3:0] puck_vel_v;
 assign puck_vel_v = pspeed;

 //Puck signs
 // Although these only represent 2 values (-1 and 1),
 //   I have defined them as 2 bits each so that they can take
 //   the actual signed value of 1 or -1. This simplifies the
 //   code in the representation. However, it could be optimized out
 //   if size was a concern.
 reg signed [1:0] puck_sign_h, puck_sign_v; // 1: right or down, -1: left or up
 initial puck_sign_h = 2'b01;     //start going right
 initial puck_sign_v = 2'b01;     //start going down



 ///////////////////
 ////Move elements///
 ///////////////////
 always @ (posedge vclock)  //overarching always block
  begin
  //Handle reset condition
   if (reset)
begin
 puck_h <= 220;
 puck_sign_v <= 1;
 puck_sign_h <= 1;
 puck_v <= 220;
     end

 //Trigger on new_frame
   if (new_frame)
    begin
//Handle paddle movements
```

```
  if (up) paddle_v <= paddle_v - 4;    //move up
  if (down) paddle_v <= paddle_v + 4;  //move down

  //Handle puck movements
       puck_h <= puck_h + puck_vel_h * puck_sign_h;  //horizontal
  puck_v <= puck_v + puck_vel_v * puck_sign_v;  //vertical
      end

      //Right wall: Bounce
 if (puck_h + 64 >= 1023) puck_sign_h <= -1 ;

      //Top and Bottom walls: Bounce
 ////Note: for the top wall detector, I use a 1-frame "look-ahead"
 //// to determine if the next frame will cause the ball to crash.
 //// This simplifies the code and allows me to avoid using a signed
 ////  register for puck_v.
      if (puck_v + puck_vel_v * puck_sign_v <= 0) puck_sign_v <= 1; //look-ahead
 if (puck_v + 64 >= 767) puck_sign_v <= -1;

 //Hit paddle: Bounce
 ////Make sure that the puck is at the right horizontal alignment AND
 //// is within the vertical "slot" where the paddle exists
     if (puck_h <= 16 && puck_v + 64 > paddle_v && puck_v < paddle_v + 128)
  begin
   puck_sign_h <= 1;
       end

 //Hit left wall: Lose
 ////Basically the opposite of the paddle hit checker. Test to see if the
 //// puck is within the horizontal alignment, but OUTSIDE the vertical alignment.
      if (puck_h <= 15 && (puck_v + 64 < paddle_v || puck_v > paddle_v + 128))
  begin
   puck_sign_v <= 0;  //Stop all motion and put puck near the center
   puck_sign_h <= 0;
   puck_h <= 300;
   puck_v <= 300;
       end
    end

   //Draw objects based on draw signals
   assign pixel = (
     (box_draw ? 3'b111 : 3'b000) |
puck_pixel |
paddle_pixel);


endmodule
```

driver demo (demo mode 4):

```
// module to demonstrate servo driver
// displays h_coord and y_coord on hex display

module driver_test(clock_27mhz, reset, h_speed, v_speed, up, down, left, right, h_pw, v_pw);
   input reset, clock_27mhz;
   input [3:0] h_speed, v_speed;
   input       up, down, left, right;

   output [7:0] h_pw, v_pw;

   // set pulse width with up and down buttons
   parameter BRT  = 30000;

   reg [31:0] v_pcount, h_pcount;
   reg [7:0]  h_pw, v_pw;
   wire       v_bflag = (v_pcount == BRT * v_speed) ? 1 : 0;
   wire       h_bflag = (h_pcount == BRT * h_speed) ? 1 : 0;

   always @(posedge clock_27mhz)
     begin
v_pcount <= v_bflag ? 0 : v_pcount + 1;
h_pcount <= h_bflag ? 0 : h_pcount + 1;
v_pw <= (down & v_bflag) ? (v_pw>8'h38 ? v_pw-1 : v_pw) :
((up & v_bflag) ? (v_pw<8'hf8 ? v_pw+1 : v_pw) :
 (v_pw < 8'h2f ? 8'h38 : v_pw));
h_pw <= (left & h_bflag) ? (h_pw>8'h38 ? h_pw-1 : h_pw) :
((right & h_bflag) ? (h_pw<8'hf8 ? h_pw+1 : h_pw) :
 (h_pw < 8'h2f ? 8'h38 : h_pw));
     end

endmodule
```

6.2 Perl Code

```
#!/usr/bin/perl -w

use POSIX;

$x_res = 16;
$distance_res = 16;   # for 1m, 6.25cm units

$width = 38; # in cm

# offsets for motor control:
```

```
$d_max = 100;         # maximum distance of gun from TV (cm)
$motor_max = 248;     # maximum value sent to motor (pulse width)
$motor_neutral = 152; # pulse width value corresponding to zero angle

$pi = 3.14159;

print "memory_initialization_radix = 10;\nmemory_initialization_vector=\n";

for (my $d = 1; $d <= $distance_res; $d += 1){
    for (my $x = 0; $x < $x_res; $x += 1){

        $x_pos = ($x/$x_res) * $width/2;
        $dist = ($d*$d_max/$distance_res);

        $theta = atan ( $x_pos / $dist );

        $theta_pw = int(($motor_max-$motor_neutral)*(2*$theta/$pi));

        print "$theta_pw,\n";
    }
}
```