A Motion-Tracking DMX512 Controller

Miren Bamforth | 6.111 Project Final Report | Fall 2014

Table of Contents

1 Introduction

Some modern theatrical lighting instruments are able to rotate in two dimensions; they are referred to as 'moving lights.' Integrating motion tracking capabilities with a moving light creates a real-time spotlight which automatically follows a person or an object onstage. This final project aimed to make a device which takes video input from a bird's eye view of a performance space, processes calibration and motion tracking data, and outputs serial data to control a set of moving lights. Currently, following a person or object with a light is usually done by hand and could be improved with automation.

The motion tracking system was the most important module of the project. It found objects by analyzing the video input for pixels of a certain color; a very saturated red sheet of paper was used as the object to track for proof of concept. At first, the system tracked the uniformly colored object and output the tracking data to a monitor and an LED display which showed internal system information. Ultimately, the project output data to a moving light, not a computer screen, for debugging and further proof of concept. All of the desired, non-optional modules specified in the proposal were implemented on time, reaching the goal of the project.

The motivation of the project was to take steps towards creating a device that could be used for future research, and the project successfully became a good jumping point for further investigation into novel performance technology.

2 Overview and Motivation

2.1 Terms

The following are some terms which will be useful to understand the rest of this document. These are industry standard terms used by electrical engineers, theater designers, and technicians.

- **DMX** or **DMX512**: The serial protocol which is used to control theatrical elements such as lighting, fogging machines, and automation
- **Lighting board**: A piece of hardware which outputs DMX512. Typically used to control lighting elements
- **Moving light** or **intelligent lighting**: A theater light which is capable of panning and tilting in place. Usually also able to change color and pattern. Controlled with DMX512
- **Conventional light**: A light which is not capable of changing position, color, pattern, or any aspect other than its brightness. Controlled with DMX512

2.2 Project Overview

The project as a whole can be modeled as four major blocks as shown in Figure 1; these blocks are the motion tracking block, the DMX processing block, the calibration block, and the display and testing block.

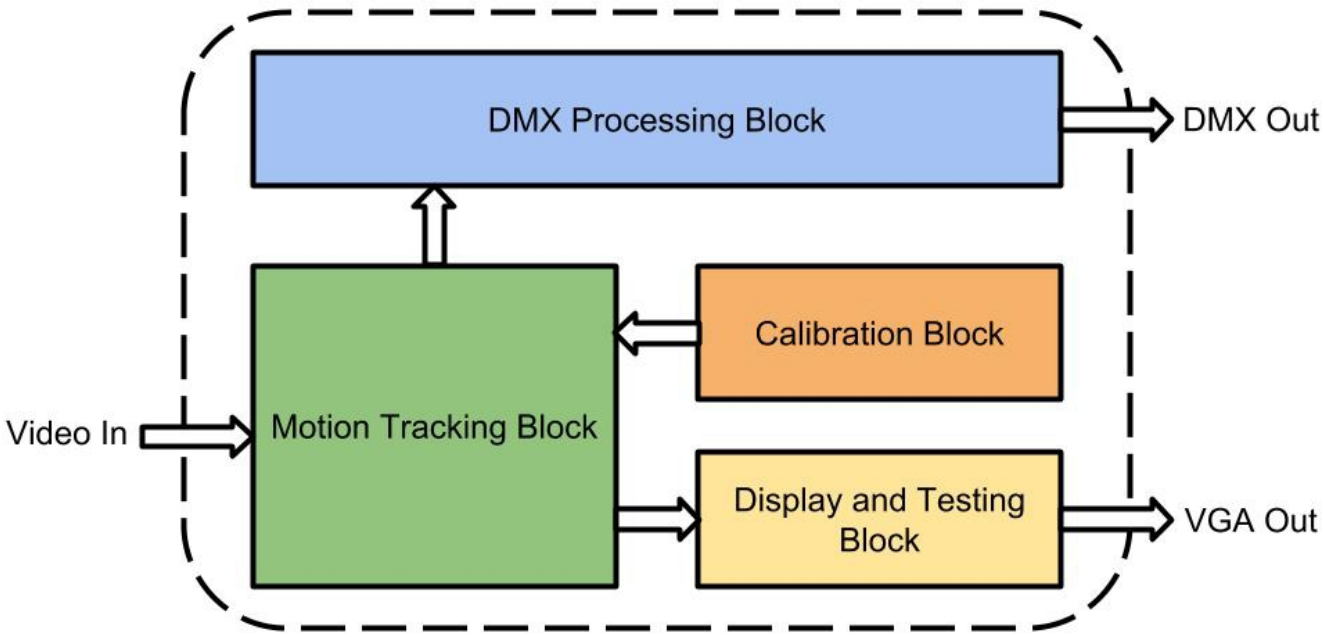


Figure 1: The high-level project design

2.3 Design Decisions and Motivation

The primary motivation for this project was to make a freestanding device which could be used as part of future projects, or at least to take steps towards that device. Completing this device in a way which would allow it to be freestanding required finishing every module, even the optional ones, which was a lofty goal given the time frame and resources of the project. The design decisions that were made in each module were guided by this motivation and the expected time squeeze. Ultimately, all of the non-optional modules were implemented, but very few of the optional modules were implemented. The design decisions also allowed for interesting, testable intermediate steps and for a satisfying final product, despite not reaching the ultimate goal of a freestanding device.

The various design decisions will be explained at appropriate steps throughout this paper. First, I will explain the high-level blocks from Figure 1, including the design decisions along the way.

Overall, the project was designed to take in video data ('video in' from Figure 1), process that video data in the **motion tracking block**, combine that processed data with calibration data from the **calibration block**, and output DMX ('DMX out') from the **DMX processing block** to some connected lighting instruments. Specifically, the motion tracking block calculated the pan and tilt data to be sent over DMX to the lights. See Appendix B for a more detailed description of the pan and tilt functionality of a moving light. The system also output video ('VGA out') from the **display and testing block** for debugging purposes and to allow the project to be functional without a DMX-controlled lighting instrument.

The **motion tracking block** stored the incoming video data in ZBT memory. The beginnings of the block were altered from the example ZBT memory code available on the course website. The motion

tracking module used data from the calibration block and from four consecutive video frames to determine the center of mass of the tracked object smoothly. It operated at 60 Hertz because it pulled its data from the outgoing VGA pixels, and the VGA protocol operates at 60 frames per second for the desired display size. The calculation of the pan and tilt bytes was done by a state machine that took approximately two or three microseconds to output one set of data.

The **calibration block** was the least important block, so it was mostly hard-coded. One of the few optional improvements was made to this block; live calibration alteration was controlled with button presses to correctly align the tilt of the light to the tracked object.

The **DMX processing block** took the pan and tilt bytes from the motion tracking block and output them using the DMX512 protocol. See Appendix A for more information on the DMX512 protocol. This block was less important than the motion tracking block but more important than the calibration, but the data flow of the project meant that it was implemented last. The block stores the most recent pan and tilt bytes as well as some hard-coded information that was necessary to make the moving light work for testing.

The **display and testing block** outputs motion tracking and calibration data to a computer monitor over VGA for easy debugging and testing without having a moving light in lab. It also makes use of the LED display on the labkit for outputting internal signals without having to use the logic analyzer.

2.4 Successes and Challenges

Specific successes and challenges will be explained in detail later in the paper, but a few key successes and challenges will first be explained here.

One of the most important intermediate successes was creating a smooth, reliable motion tracking module. Once I completed this step, I was more confident that the project would work overall. Approximately the first half of the project lead up to having a finished motion tracking module, so it was a significant milestone to reach.

Overall, the project worked, given the correct calibration settings. Attaining basic accuracy from the moving light was a significant challenge that was attained once the test rig was set up and functional.

The biggest challenge of the project was being unable to test one of the core modules due to incompatibilities between Xilinx's Coregen modules and the version of ModelSim on the lab computers. Eventually, I was able to find a Windows 7 machine (my personal computer runs Windows 8 which is not supported by ISE) on which to run and simulate this module. However, there was a lot of time spent attempting to fix ModelSim on the lab computer and ISE on my personal computer beforehand. More details about overcoming this challenge are in section 3.3.

Finally, there is an unsolved bug in the project. I never overcame this challenge. The bug could not be found in simulation, so the next step would be to output some internal signals to the logic analyzer. I ran out of time before I could take this step. Ultimately, I changed some of the test rig to eliminate this bug somewhat for the demo video. More details about the bug are available in section 3.3.

3 Design and Implementation

This section explains more of the details of the system and verbalizes the block diagram shown in Figure 2.

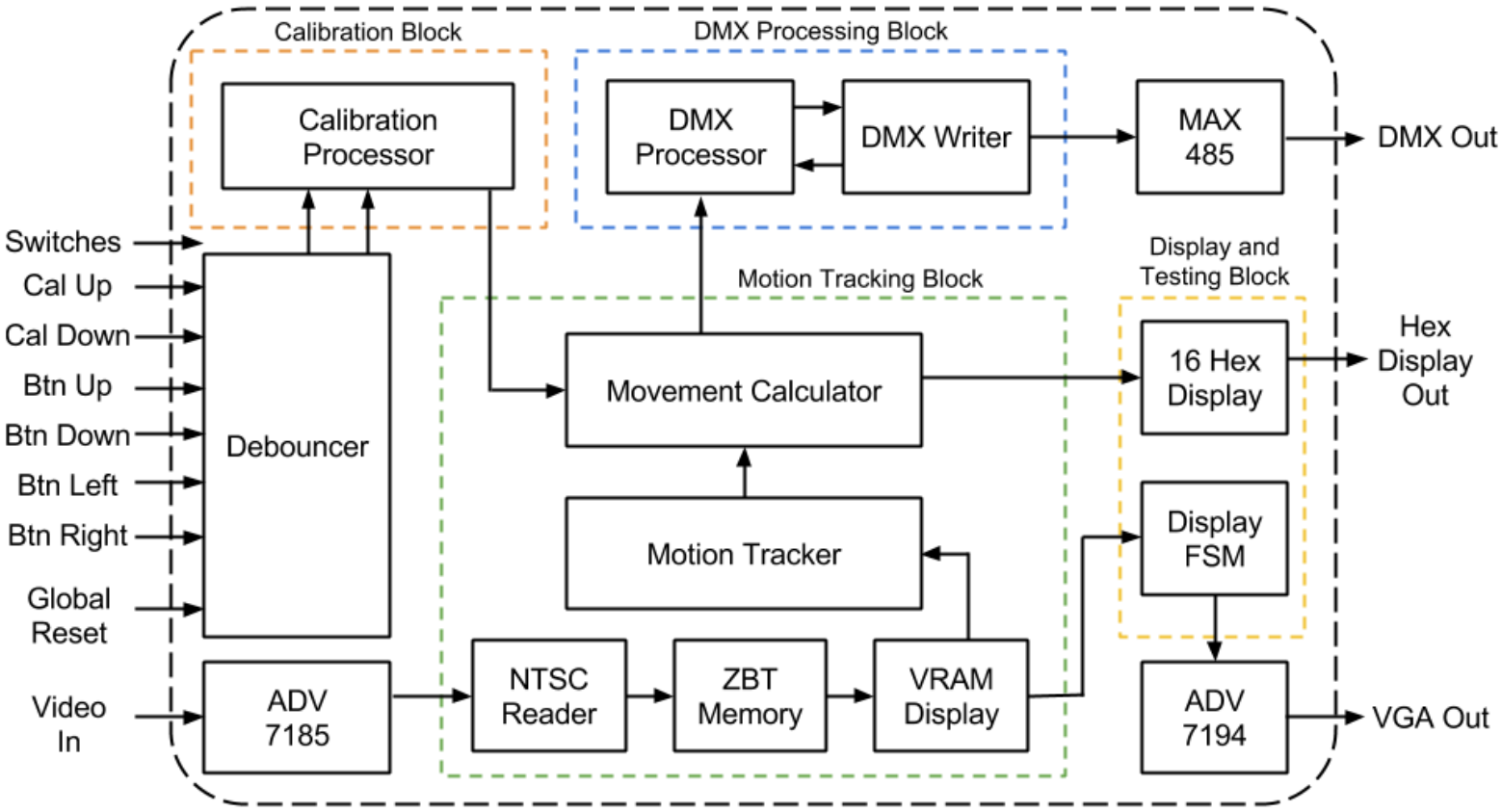


Figure 2: A detailed block diagram showing most modules and the simplified links between modules

3.1 Camera and Other Hardware Setup

This section will explain how the inputs and outputs interface with the project to give some context.

The camera used was an NTSC camera which sends composite video data to the ADV7185 chip in the labkit. There was demo code on the course website that showed the interfacing from the video to ADV7185, to the NTSC decoder, storing values in ZBT memory, and reading these pixels with the VRAM display module. Setting up the camera interface was a much simpler task than anticipated because of the example code.

The switches and the left, right, up, and down buttons were used to control the display FSM which will be explained in detail in section 3.5. Two additional buttons were used to control the calibration height as described in section 3.2. The enter button was used as the global reset.

The hex display output was created by a module available online, written by the course staff.

The VGA output was created by sending data to the ADV7194 chip inside of the labkit which translated digital data into the analog VGA signal.

The DMX output was created by sending a signal to one of the user outputs on the labkit and wiring it to a MAX485 chip on a breadboard. This chip created the differential signal necessary for the DMX protocol as explained in Appendix A. The output of the MAX485 chip was then sent to the moving light with a 3-pin XLR cable as explained in section 4. Figure 3 shows the wiring of the MAX485 to the labkit.

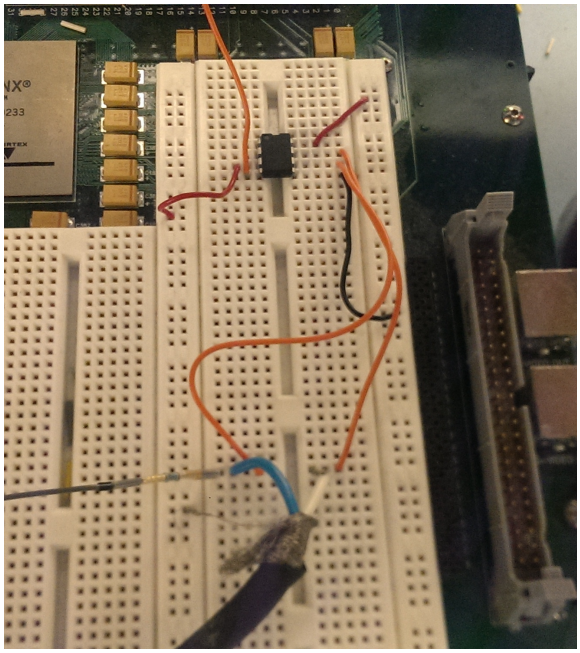


Figure 3: The wiring of the MAX485 chip. The two orange wires in the middle of the picture are the wires that connected the differential output to the blue and white 3-pin XLR cable. The orange wire at the top connected the user output of the labkit to the input of the chip

3.2 Calibration Block

The calibration block consisted of only one module in the end. The **calibration processor** contained mostly hard-coded information about the 3D arrangement of the test rig, the camera, and the tracked object. The following are the calibration data that the calibration block outputs, as shown in Figure 4:

- **x_real_world**: The horizontal size of the camera's view, in pixels
- **y_real_world**: The vertical size of the camera's view, in pixels
- **z_real_world**: The height of the camera, in pixels. This value could be changed by the height_cal_up and height_cal_down button inputs
- **x_light**: The horizontal coordinate of the camera's view above which the light was placed
- **y_light**: The vertical coordinate of the camera's view above which the light was placed
- **light_pan_addr**: The address of the pan byte for the light. See Appendix B for more information on addressing
- **light_tilt_addr**: The address of the tilt byte for the light

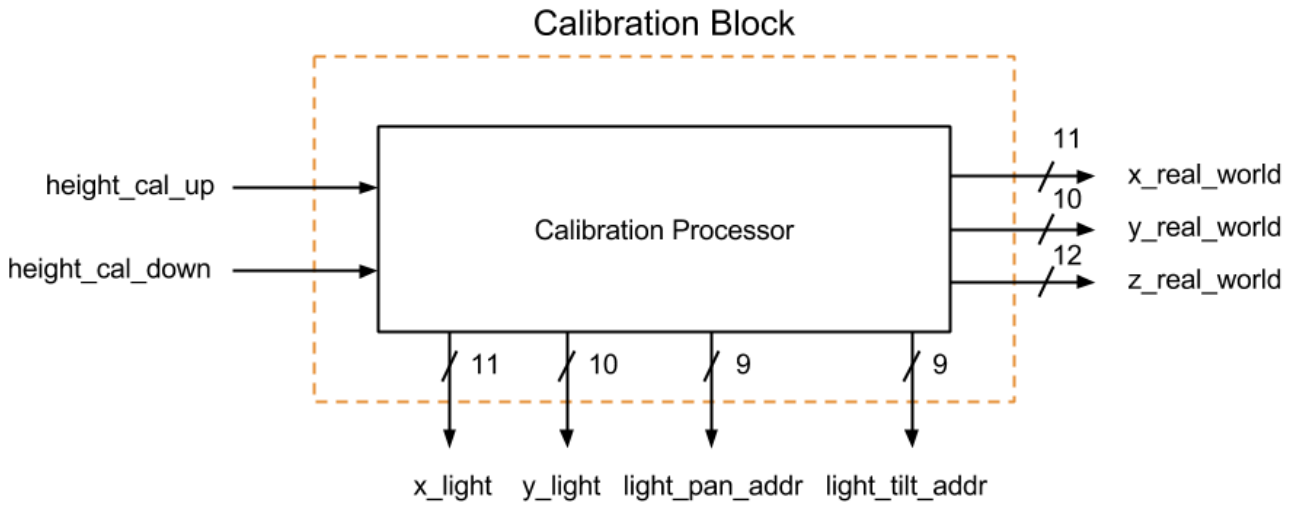


Figure 4: The calibration block with detailed inputs and outputs, including bit widths

The hard-coded values were chosen as follows. The x_real_world and y_real_world values were set to 1024 and 768 respectively, which was a mistake because the monitor had a resolution of 1024 by 768, whereas the camera had a resolution of approximately 704 by 480. The x_light and y_light values were chosen to be in the middle of the camera's input to allow for a good range of motion for the light. The pan and tilt addresses were chosen based on the light's addressing specifications.

The z_real_world value, or the height in 3D space, was not hard-coded. It was calculated as shown in the equation below, where x is x_real_world and z is a ratio of the height of the system in feet over the width of the camera view in feet, multiplied by 512. The ratio was multiplied by 512 to allow for extra granularity when changing the ratio. The ratio acted as a conversion between the real world height which cannot be easily measured in pixels and a virtual height, measured in pixels, which could be passed through the system conveniently with the x and y values, also measured in pixels.

$$height = \frac{x \times z}{512}$$

The numerator was divided by 512 because a division by a multiple of two is simply a shift right. The resulting height was output as z_real_world.

Pressing the height_cal_up button increased the ratio z, and pressing the height_cal_down button decreased the ratio z. In this way, the height calibration of the system was adjustable as necessary.

This module was simple to create, did not cause any design issues such as timing or memory constraints, and fulfilled the design requirements of the calibration block completely.

3.3 Motion Tracking Block

The motion tracking block was the most complex block in the system. This section will go over each module shown in Figure 5 as well as some design and implementation challenges.

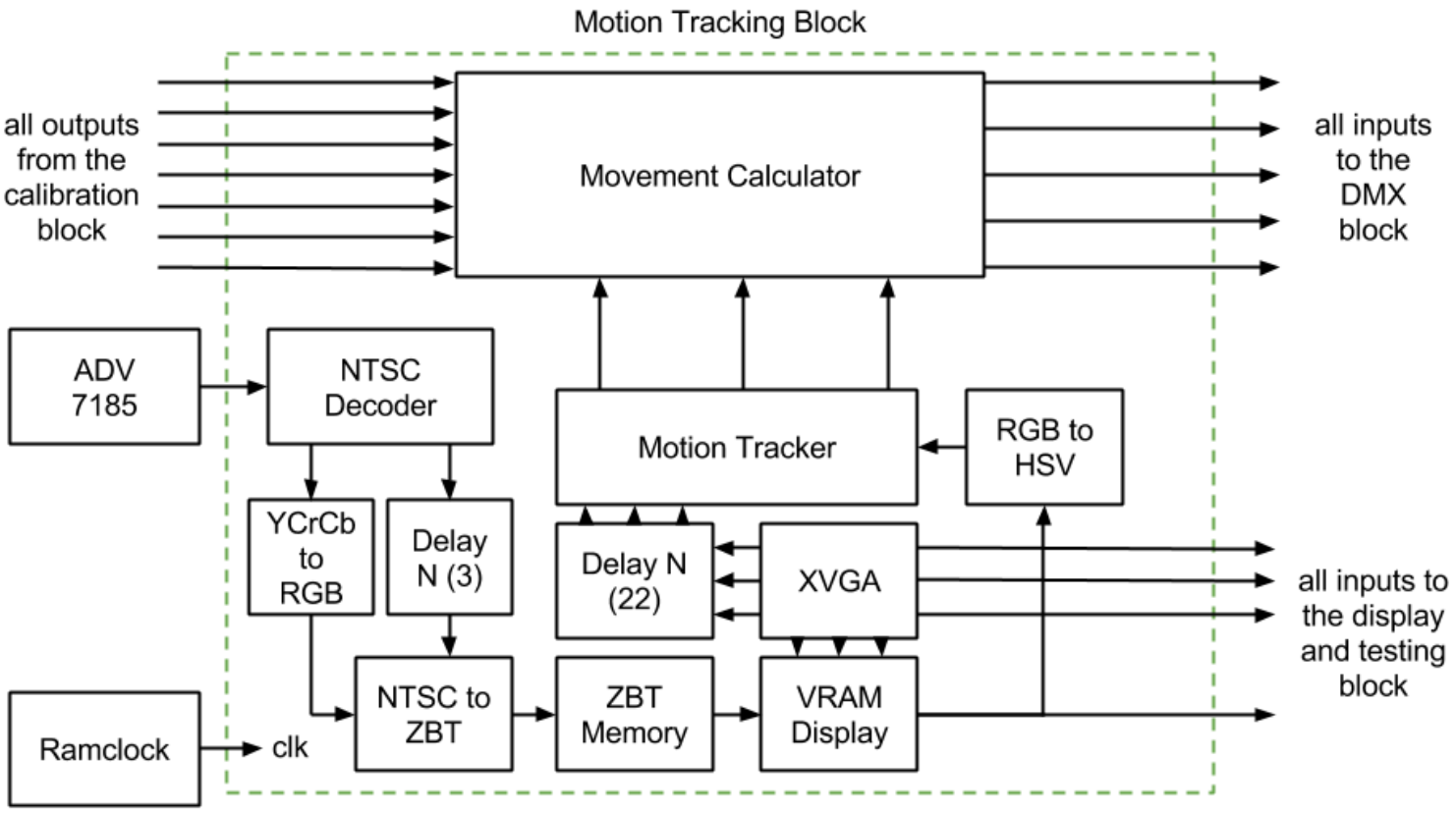


Figure 5: The motion tracking block with inputs, outputs, and intermediary values. The names and bit widths of the wires have been left out for simplicity. They will be explained in writing

The descriptions of the modules will be in the approximate order of the data flow, from the video input to the calculator outputs.

Quite a few of these modules came from the 6.111 example NTSC code and will not be explained in great detail. The starter code for these modules is available on the course website. The following modules came from the example code and were not modified or modified very little:

- **NTSC Decoder**: This module took in the digital NTSC data from the ADV7185 and output a 30-bit wide YCrCb signal and some other NTSC signals like field, vertical sync, horizontal sync, and data valid as described in the NTSC documentation. These signals will be referred to as FVH and DV from now on
- **YCrCb to RGB**: This module took in the YCrCb data from the decoder and transformed it to the RGB color space. It output a 24-bit wide RGB signal
- **Delay N (3)**: YCrCb to RGB took three clock cycles, so this module delayed FVH and DV by three cycles, too
- **NTSC to ZBT**: This module took in the RGB values along with the delayed values and prepared them for storage in memory. This module was slightly altered from the starter code because the project stores two 18-bit RGB values in one memory location instead of 4 bytes of Y values. It output the memory address, the 36-bit data for that address, and the write enable line for the memory

- **ZBT Memory**: This unchanged module controlled the writes and reads to the ZBT memory. As previously mentioned, two 18-bit RGB values are stored in each memory location. This is an attempt to use memory as efficiently as possibly. The two least significant bits are dropped from the R, G, and B values to create an 18-bit RGB to store in pairs
- **XVGA**: This module created hcount (11 bits), vcount (10 bits), hsync (1 bit), vsync (1 bit), and blank (1 bit) values for the outgoing video data as described by the VGA specifications. It was unchanged from the example code
- **VRAM Display**: This module took in the hcount and vcount timing parameters from the XVGA module and the read data from the ZBT Memory. It then output the 18-bit pixels that corresponded to the current hcount and vcount. It is mostly unchanged except for adjusting to the 36-bit RGB values in memory instead of the 4 bytes of Y information
- **RGB to HSV**: This module took in 18-bit RGB values and output 18-bit HSV values. It was unchanged from the example code
- **Delay N (22)**: This module is similar to the Delay N above. Specifically, it delayed hcount and vcount for use in the tracker module and hsync, vsync, and blank for use in the display and testing block. These delays were because the RGB to HSV conversion takes 22 cycles
- **Ramclock**: This unchanged module created various clocks such as the 65MHz clock that was necessary for the XVGA signals; this clock was used for most of the modules in the project

Modifying the above modules that needed adjustment was the first step of the project. At first, I neglected to change the frequency of the write enable signal, so I was only writing half of the data to memory. This challenge was the first hurdle that I faced, and it took hours of debugging to figure out the issue due to the complexity of this block.

The **motion tracker** was the first module that I wrote completely from scratch. At a high level, this module was supposed to computer the center of mass of an object within the view of the camera. For simplicity, I used color tracking to find the object. Ideally, color tracking would not be used because this technique limits the system to tracking one object at once. Also, the object must be a very uniform color, so changes in lighting conditions affected the tracking module.

I used a very saturated red piece of paper as the tracked object. I experimentally found the hue, saturation, and value of the paper, so I was able to set narrow bounds on the HSV range. When the tracker found a pixel that was in the correct HSV range, it added its horizontal and vertical location to a sum of other horizontal and vertical locations, as well as incrementing a counter. The equations for this process are as follows.

$$x_{total} = \sum x_i \text{ where } HSV_i \text{ is within the desired range}$$
$$y_{total} = \sum y_i \text{ where } HSV_i \text{ is within the desired range}$$
$$count = n \text{ where } n \text{ is the number of pixels within the desired range}$$

The values calculated from these equations were then used to calculate the coordinates of the center of mass in terms of the center of mass's location on the screen. The widths of x_com and y_com were 11-bit and 10-bit respectively because they represented locations on the 1024 by 768 pixel screen.

$$x_{com} = \frac{x_{total}}{count}$$
$$y_{com} = \frac{x_{total}}{count}$$

The simplicity of the algorithm meant that x_com and y_com would be successfully calculated during each frame of the outgoing video since the XVGA resolution operates at 60 frames per second. The x_com and y_com values were sent to the motion calculator and to the display and testing block. Figure 6 shows how the project looked when the display and testing block overlaid the center of mass data on top of the regular video output.

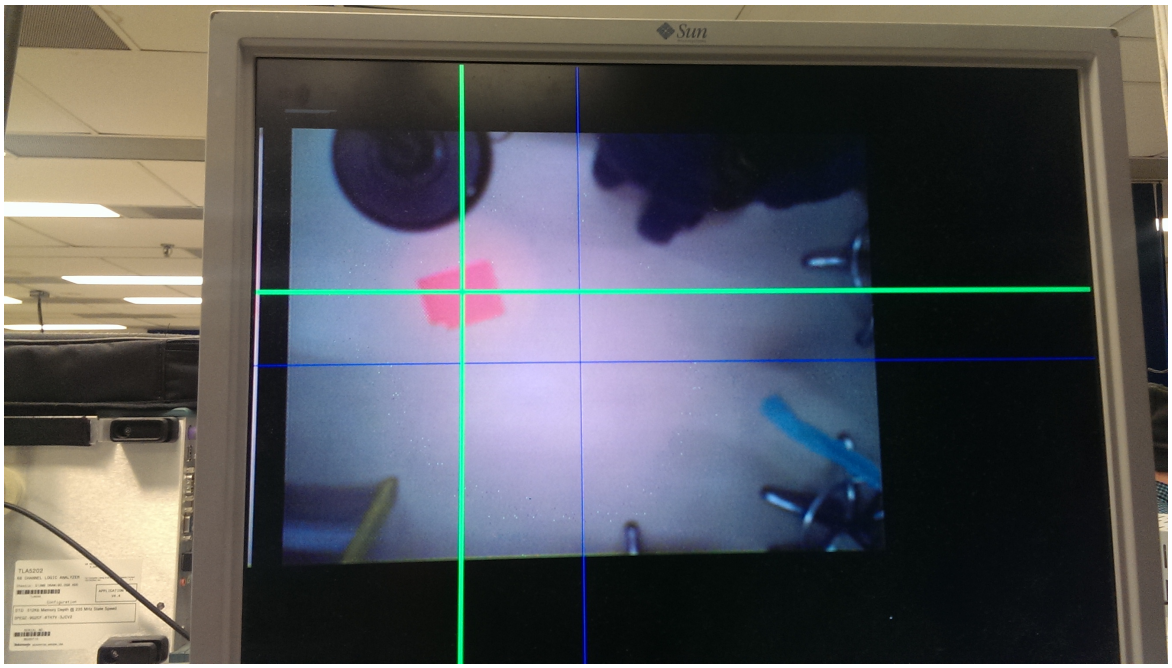


Figure 6: The center of mass data was represented as the intersection of the green cursors (and the intersection of the blue cursors was the light's position). Here, the cursors were centered on the red piece of paper which was the tracked object

After implementing the green center of mass cursors on the screen, it was obvious that the tracking was a little jumpy. I used a low pass filter to smooth out the movements. I did this by averaging the current x_com and y_com with the previous three x_com and y_com values. The x_com and y_com equations became as follows:

$$x_{com} = \frac{(x_{total} + x_1 + x_2 + x_3)}{(count + count_1 + count_2 + count_3)}$$

$$y_{com} = \frac{(y_{total} + y_1 + y_2 + y_3)}{(count + count_1 + count_2 + count_3)}$$

After each new value was calculated, the new values were stored as x_1 and y_1, then x_1 and y_1 were stored as x_2 and y_2, and so on. This technique greatly decreased the jumpiness of the center of mass. There was also very little perceivable delay when moving the object very quickly. The 60

Hertz frame rate and the use of only four frames for averaging allowed the center of mass to move perceptively as quickly as before the low pass filter was added. More frames could have been used, but averaging too many frames would have caused noticeable lag in the center of mass location.

I anticipated the tracking module to be the most difficult module of the project because I originally wanted to use techniques other than color tracking such as taking the difference between successive frames to find movement or attempting corner tracking. However, given the time allotted and the fact that I was working alone, I chose to use color tracking for proof of concept. No challenges other than the jumpiness of the center of mass arose during the implementation.

The final module in this block was the **movement calculator**. The calculator took in center of mass data (x_com and y_com) from the tracker and calibration data from the calibration block. Specifically, it took in every single piece of output data from the calibration block as described in section 3.2. It then combined these pieces of data to output a pan byte and a tilt byte to be sent to the moving light by the DMX block. Figure 7 illustrates the data that the movement calculator has to work with.

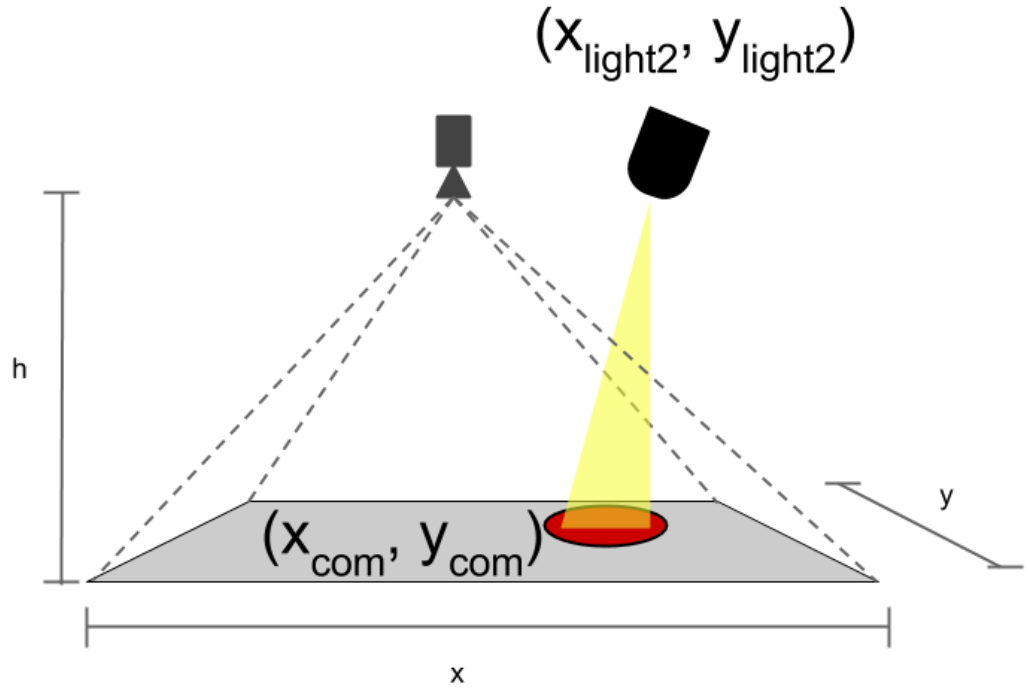


Figure 7: An illustration of the calculator's understanding of 3D space. The calculator uses primarily the two sets of coordinates and the light's height to calculate pan and tilt bytes

The calculator was a state machine which took about two microseconds to complete one calculation. Calculating the pan and tilt bytes can be separated into two different processes; these processes were done in parallel in the state machine as follows. Figure 8 also visually shows the calculator's data flow.

Pan:

1. Calculate Δx and Δy, the differences between the coordinates of the light and the center of mass
2. Normalize the values such that Δx and Δy were equal to or less than one with a divider module. This was necessary because the arctan module only took input between -1 and 1

3. Input Δy' divided by Δx' into the arctan module like so:

$$\arctan\left(\frac{(deltay')}{(deltax')}\right)$$

4. Scale the arctan output. Originally the output is negative pi to pi. The final output range should be 0 to 255 so that the values use the full range of the 8-bit data output

Tilt:

1. Calculate Δx and Δy, the differences between the coordinates of the light and the center of mass
2. Calculate D, the distance between the center of mass and the light by using this equation:

$$D=\sqrt{(deltax^2+deltay^2)}$$

3. Normalize D and H, the height, such that they are equal to or less than one with a divider module. This was necessary because the arctan module only took input between -1 and 1
4. Input the normalized D' and H' into the arctan module like so:

$$\arctan\left(\frac{(D')}{(H')}\right)$$

5. Scale the arctan output. Originally the output is negative pi to pi. The final output range should be 0 to 255 so that the values use the full range of the 8-bit data output

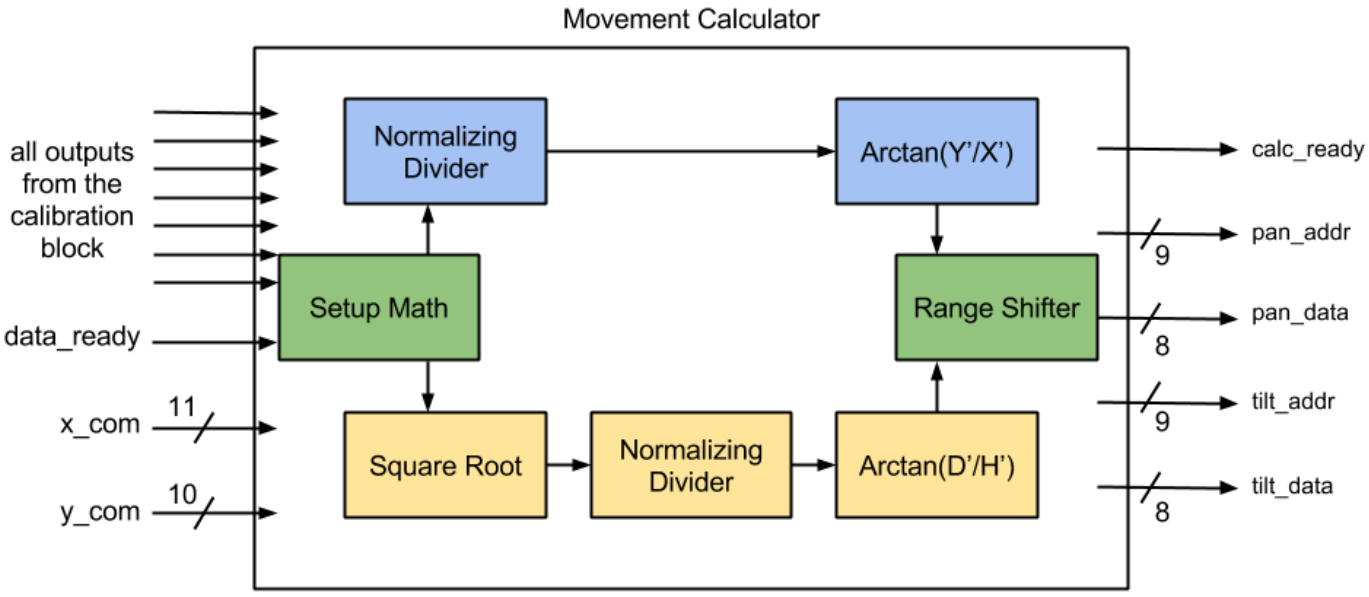


Figure 8: The inside of the calculator with data flow from left to right. Blue elements were used for the pan calculation, yellow elements were used for the tilt calculation, and green elements were for both.

The processes for calculating pan and tilt were quite similar. Pan used one divider and one arctan module. Tilt used one square root, one divider, and one arctan module. Designing the module was relatively simple; it consisted of very basic math. However, getting this module to function was arguably the hardest part of the entire project. I used Xilinx's Coregen modules to implement the square root, pipelined divider, and arctan modules. Unfortunately, ModelSim, at least on the lab computers, is incapable of running simulations on modules with include Coregen modules as submodules. The calculator is such a large module that debugging it without a simulation would have been very difficult and would have taken very long.

I spent many hours in lab trying to get the simulation to run, at home trying to get ISE to run on my Windows 8 machine, and at home after borrowing a Windows 7 machine which could run ISE and ISim correctly. Ultimately, I was able to debug my module after rewriting it to use blocking assignments and setting up the Windows 7 machine correctly. The delay caused by this module directly impacted the final result of the project. I was not under too much pressure during the last week of the class, but I certainly spent more time in lab than I would have had to if I had not had the Coregen simulation issues.

Additionally, once I ported the module to my labkit and my lab computer, I could no longer touch that module at all. This restriction meant that when I found a bug in the tilting near the end of the project, I could not easily do anything about it. The bug caused the tilt to reset itself to a tilt value of approximately 0x7F when it reached a value of 0x9C. It looked like an overflow error, but I was unable to find an overflow anywhere. I managed to isolate the issue to the calculator module before I ran out of time, but I could not locate the bug before the end of the class due to the complications with simulation. If I had had more time, my next step would have been to try to use the logic analyzer like a simulator; I would have output internal signals and connected the logic analyzer to these signals. This approach would have presumably taken a long time to find the bug, and it would have been quite tedious due to the slow compile time of the labkit.

In the process of creating these modules, I learned a lot about how ISE, ModelSim, and ISim work. I learned about Coregen modules and choosing appropriate bit widths to maximize precision and minimize space and time usage. Certainly, designing and writing the tracker and the calculator modules was a large part of my work and effort on the project.

3.4 DMX Processing Block

The DMX block consisted of two modules, one of which handled the incoming pan and tilt bits from the calculator and one of which created the serial output stream that went to the MAX485 chip and then on to the moving light, as seen in Figure 9. Both of these modules use a 27MHz clock because I inadvertently wrote the writer to work with a 27MHz without thinking about how the rest of the project uses a different clock from the ramclock module. These modules could easily be changed to use the same clock as the rest of the project, but I ran out of time.

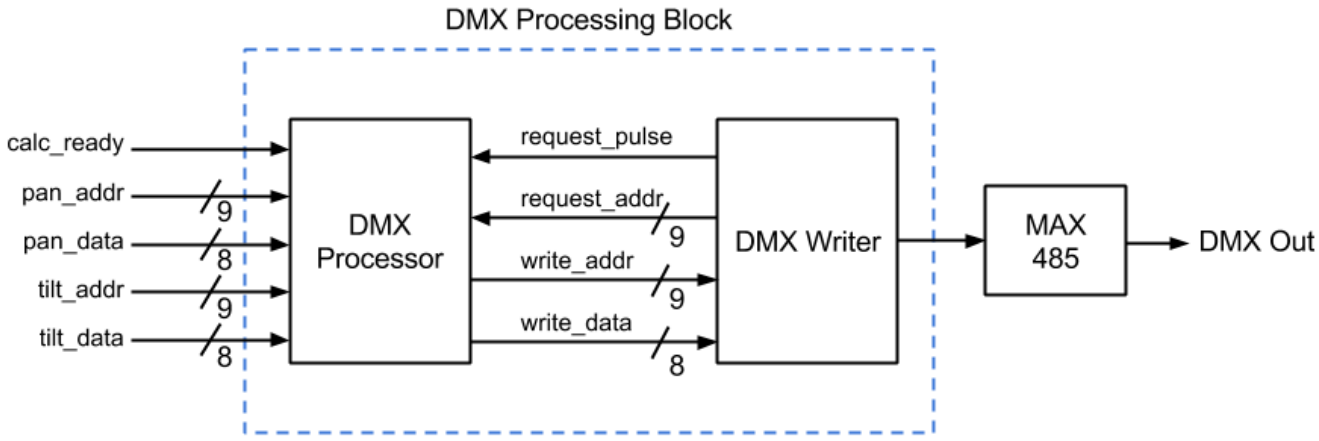


Figure 9: The DMX processing block with detailed input, output, and intermediary signals

The **DMX processor** module waited for the ready pulse from the calculator. It stored the most recent pan, tilt, pan address, and tilt address values from the calculator. When requested, the DMX processor passed these values to the DMX writer; it returned zero if the requested address was not one of the pan or tilt addresses. This module was very simple to write. However, I tried to hard-code some data into this module, but doing so inexplicable broke the hex display without fail. To counter this, I hard-coded the same data into the DMX writer instead.

The **DMX writer** output a single serial stream of data. It followed the DMX protocol as explained in Appendix A by counting each clock cycle to hold the signal low or high for the appropriate number of clock cycles. When writing the frames that contained actual DMX data (as opposed to the starting or stopping frames), the writer requested data from the DMX processor. It then output the returned value, which could possibly be pan, tilt, or zero.

Additionally, to counter the hard-coding issues in the processor, the writer output hard-coded data to tell the moving light to be on at a certain intensity, to be red, and to have its shutter open when it was time to write the corresponding addresses for those characteristics. Writing data in this way overcame the challenge of not breaking the hex display which was the only challenge to arise for this block.

The DMX block was far easier to write than I expected. Once I found a reliable description of the protocol, it took only a few hours to implement from start to finish. The hard-coding issues were the only reason that this block took any real amount of time to create.

3.5 Display and Testing Block

The display and testing block shown in Figure 10 consisted of a few different modules to output two types of debugging data. The VGA output was manipulated in various ways to help with debugging, such as adding the green cursors that followed the center of mass in Figure 6. The hex display was used to show internal values in a more cohesive way than the logic analyzer or the oscilloscope could.

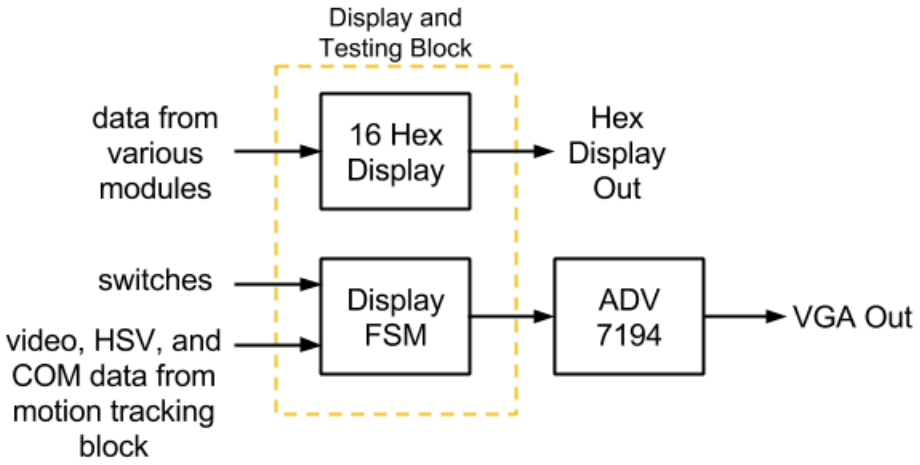


Figure 10: The display and testing block with simplified inputs and outputs

The **16 hex display** module output values to the 16 LED display locations on the labkit which could display hex characters. This module was not modified from the module given on the course website.

At the end of the project, the display would sometimes not work correctly. As far as I could tell, this was due to different arrangements of signals during compiling of the project because the breaking of the display was moderately random and could be fixed by simple changes like changing a constant. Figure 11 shows the hex display with various data.

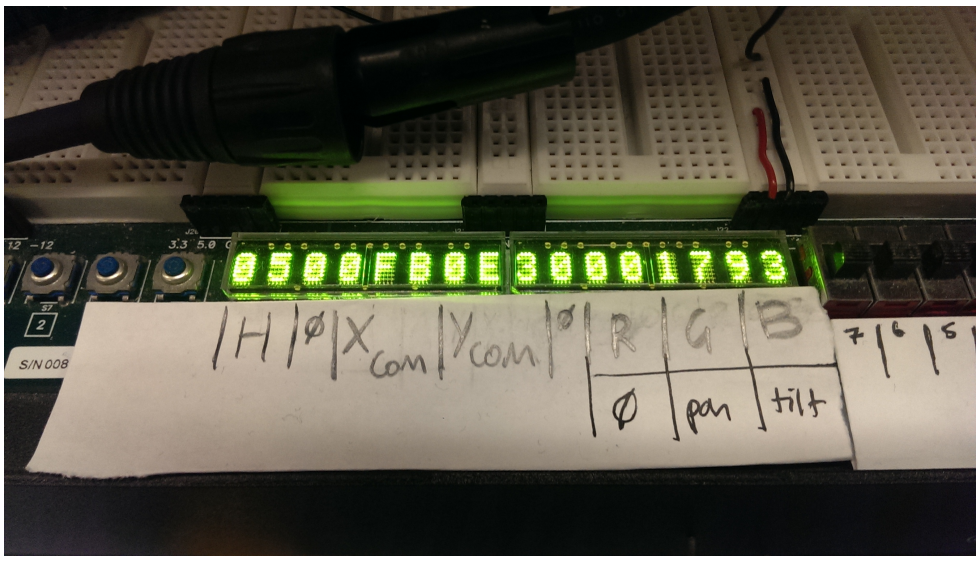


Figure 11: The 16 hex displays working. The labels describe what each hex display location represents. From left to right, it displays the hue, then nothing, then x_com and y_com, then nothing, then the pan and tilt bytes. Previously it displayed RGB values as shown by the label

The **display FSM** controlled the mode of the output video image based on the switch inputs. There were four modes:

- **Plain video mode**: The video stream was outputted without any added features
- **Motion tracking mode**: The video stream was outputted with thick, moving green cursors that would follow the center of mass of the tracked object. Blue cursors would display the location of the light as described by the calibration values. Figure 6 shows this mode
- **Moveable cursor mode**: The video stream was outputted with thin green cursors that would move when the up, down, left, or right buttons were pressed. The intersection of the cursors was treated as if it was the current center of mass, so the calculator and DMX block also reflected the use of the cursor placement as the fake center of mass. Blue cursors would display the location of the light as described by the calibration values. Figure 12 shows this mode
- **HSV mode**: The video stream was completely overwritten based on the HSV value of each pixel. If the pixel was within the tracker's hue range, it was red. If it was within the saturation and value ranges, it was blue. If it was within no ranges, it was black. If it was within all ranges, it was white. This mode showed the tracker object as showing up white, whereas everything else showed up as red, blue, or (most commonly) black. Figure 13 shows this mode

Figure 12: A photos of the display in cursor mode. The red light can be seen faintly at the location of the green cursors' intersection.

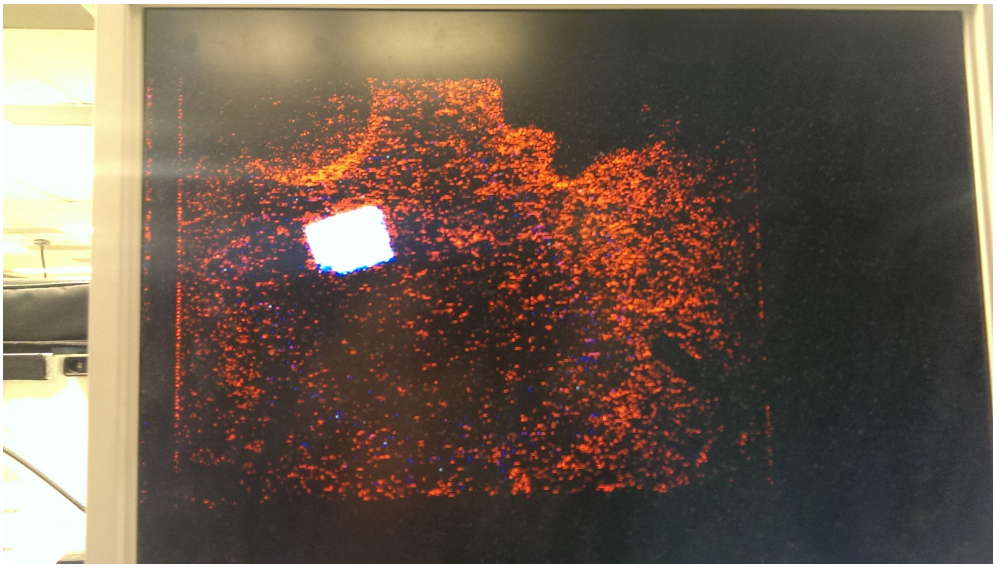


Figure 13: HSV mode. The white rectangle was the piece of red paper that was tracked. This view shows how noisy the NTSC camera can be

To implement these modes, the display FSM was passed a large amount of data. Therefore, I placed this module within the top-level labkit file instead of making it into a separate module. This module uses the following signals from other modules: hcount, vcount, hsync, vsync, and blank to display the basic video stream; delayed versions of those five signals for use in HSV mode which has a delay of 22 clock cycles; switches and button inputs to handle the states and the cursor movement; center of mass data to control the cursor in motion tracking mode; and, finally, RGB data for each pixel.
There were also two other features that the display FSM controlled, based on the switches. One switch could overwrite all of the memory to black so that the noise from the memory was removed. Another switch could show a small, compressed version of the video screen. That feature was leftover from the example code. I would not count either of these features as modes due to their simplicity and lack of debugging uses.

I altered the hex and video display features throughout the project to meet my current debugging needs. In that way, there were not any specific challenges or successes for this block in particular because the block was assembled piece by piece alongside the other blocks. For example, getting the center of mass to display as moving cursors was a key success, but that was more a success of the motion tracking block than of the display and testing block.

3.6 Optional Features

I specified a number of optional features in my original proposal. Due to time constraints, only one of these features was implemented or even designed. The unimplemented optional modules will be discussed in section 5 along with other suggested next steps.

The sole optional feature is the ability to change the calibration in real time with the buttons as explained in section 3.2.

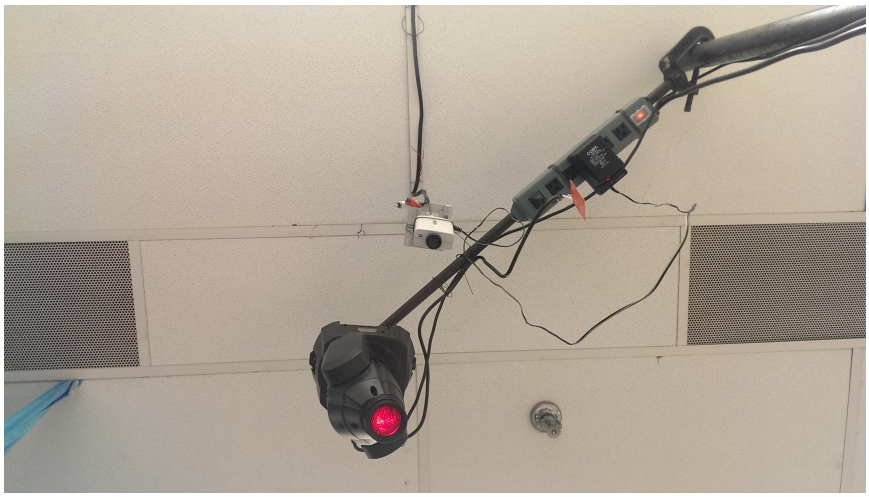
4 Testing

There were multiple ways in which I tested the project. First, ModelSim test benches were used to test many of the modules. Alternatively, some of the modules were tested with sight; for example, the modules which changed the video output from black and white to colored were simply tested by observing the video on the screen.

The motion tracking capabilities of the project were tested by creating green cross hairs on the screen as shown earlier in Figure 6. The calculations done by the module were first tested with a test bench, but qualities like the smoothness of the tracking were tested by sight.

The LED display was also an integral part of the testing process. Throughout the project, I displayed different hex values on the LED display as seen earlier in Figure 11.

Finally, I set up a test rig with a moving light on a pipe as seen in Figures 12 and 13 below. The moving light was positioned overhead as it would usually be in a theater space. The position of the moving light was the reason for the camera to be mounted from the ceiling, pointing downwards; the camera had the same perspective as the light, allowing for the calculations to be simpler. The pan and tilt calculations were tested with a ISim test bench on a computer outside of lab and by visual inspection.



Figures 12 and 13: On the left, Figure 12 shows the test rig from far away. The light is attached to the sidearm which is clamped to the vertical pipe. There is a power strip attached to the sidearm which powered the light and the camera. On the right, Figure 13 shows the test rig from below. The relationship between the camera and the light's positions in the xy plane can be seen more easily

The test rig consisted of the following items. All equipment except for the light was borrowed from E33 Productions, a student lighting group.

- **Chauvet Intimidator 100 IRC**: This was the moving light used. This style of moving light is much smaller than most moving lights which was good for testing in the lab; this light is quieter and dissipates much less heat than a regularly sized light
- **Sidearm with a c-clamp**: This was the horizontal bar that the light was clamped to
- **Threaded pipe**: This was the vertical bar that the sidearm was clamped to
- **50-pound pipe base**: This pipe base was much larger and heavier than necessary, but it was the only available base that the threaded pipe could screw into
- **3-pin XLR cable**: The DMX data was transmitted over this cable. The Chauvet Intimidator used 3-pin connectors, not the more conventional 5-pin, so 3-pin XLR cable was used here
- **IEC power cable**: This cable powered the moving light. It plugged into a power strip provided by the lab which was mounted on the sidearm
- **NTSC camera and cables**: The lab provided an NTSC camera which transmitted composite video data to the labkit. It was also powered by the lab power strip on the sidearm

Testing the functionality of the entire project was as simple as calibrating the light, moving around the tracked object, and observing the effect. Figure 14 shows the light successfully shining on the tracked object. For testing, the light was chosen to be red. Other colors of light changed the hue of the tracked object, but the red color allowed the light to be seen without altering the motion tracking.

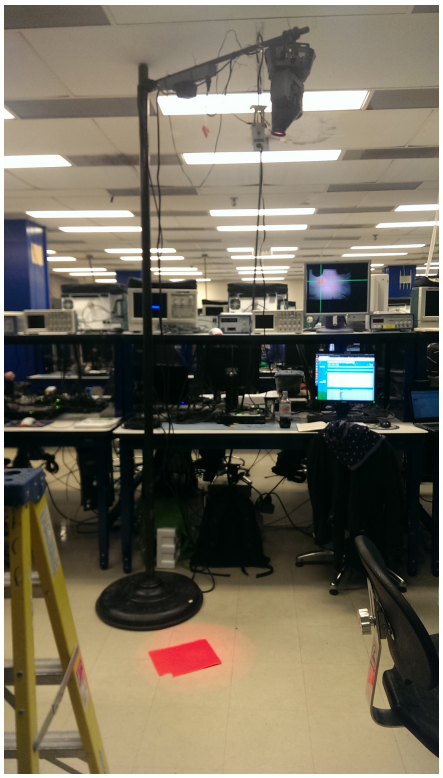


Figure 14: The red light shining on the tracked object on the floor. The whole test rig can be seen here

5 Review and Recommendations

This section will discuss improvements and next steps for the project as well as what I would do differently if I did this project again.

I more or less stuck to the schedule that I specified in my proposal. However, my time estimations were wrong for a number of modules. Namely, the tracker and DMX writer were much easier than expected, and the calculator was much, much more complicated than expected. In hindsight, I would have spent less time trying to get ModelSim to work with Coregen modules; I would have moved to the Windows 7 machine much earlier if I knew that that was a viable solution.

Because of time constraints, I ended up using two different clock speeds in my design: a clock for the DMX block and a clock for everything else. I would have refined the design for the DMX block to use the same clock as the rest of the project for consistency and to resolve the synchronization issues between the calculator and the DMX processor which arose as a result of the mismatched clock speeds.

I did plan my modules and the connections between modules in advance, but there were times when I had to alter my design, such as changing the calculator to use blocking assignments. I would think through the design just a bit more if I were to do this project again based on what I now know about Coregen, ModelSim, and Verilog.

I integrated everything as I went along so that I could do non-simulation testing at every step. I was very pleased with how this process went. I did not have any surprise integration issues because of this approach. I would not change the order or the way in which I designed and wrote the modules.

There are quite a few optional modules that I did not get to. Other than smoothing out any remaining bugs and timing issues, my next step for the project would be to implement the DMX reader optional module. Since the DMX writer was easy to write, I anticipate the DMX reader to be similarly simple. This would improve the system greatly without adding much complexity. A chip to decode the RS-485 signal would be necessary. A lighting board or other external DMX controller would have to be added to the test rig to test the functionality of the DMX reader.

After the DMX reader, I would implement a fancy calibration module. The project would have a DMX address which it listened to for calibration data. In this way, the calibration could be controlled through an external lighting board.

I would like to change the color tracking algorithm to something more complicated so that the project could track a person or multiple people onstage without them being of a certain color. Originally, I wanted to implement this project with IR LEDs and an IR camera, so perhaps adding that would be the next step.

Finally, the project is currently configured to control only one moving light. If this were a real device, presumably the user would want to automate more than one light at once, so changing the number of controllable lights would be necessary.

6 Conclusion

Overall, this project works as a motion tracker for one person or object on stage, given the correct calibration and settings. The motion tracking is reliable if simplistic. The pan and tilt calculation is not perfect but is functional enough for proof of concept. The DMX writing is easily done with an FPGA.

Personally, I am extraordinarily glad that I took this class and that I chose this project. I have gained such a large amount of knowledge and experience from designing and implementing this project by myself; I chose to work alone because I am substituting 6.111 for 6.UAP, so I tried to treat this project as if it were my thesis project. I am pleased with my results, and I had a lot of fun when I wasn't wrestling with ModelSim.

I am happy with the amount of this project that I completed, although I feel as if there is more to be done. I would really like to continue with it as a future project. I want to find the remaining bug, and I want to expand beyond color tracking and tracking one object. Potentially, this work could be the start of something exciting; hopefully there will be opportunities to continue down the path with this project in the future.

Appendix A – DMX512 Protocol

DMX512 is a widely used, industry standard serial protocol which is commonly used to control theatrical elements, primarily lighting fixtures. It is an extra layer on top of the RS-485 protocol which is designed for asynchronous digital communication between devices [3]. The RS-485 uses differential signals on two wires to send data.

One wire transmitting DMX is referred to as a *universe* and can send up to 512 *frames* of DMX in a single DMX *packet*. Each frame contains a start bit, eight bits of information, and two stop bits. Because data frames are 8-bit, there are 2^8 possible values for a piece of DMX data, ranging from 0 to 255. The addresses for each piece of data are 9-bit addresses because there are 0 to 512 or 2^9 frames.

More information is available from *DMX 101: A DMX 512 Handbook* which is in the references section [4]. This is the resource that I used to construct the DMX writer module.

Appendix B – Moving Light Basics

The pan of a light refers to its rotation as seen from a top view of the light. Lights can usually pan from 0 to 360 or 540 degrees. The tilt of a light refers to the movement of the barrel, usually about 120 degrees in either direction from the neutral downwards position. Figure 15 visually represents panning and tilting.

Figure 15: A picture of a moveable camera that can pan and tilt [6]. The panning is represented with the continuous, 360 degree circle around the device. The tilt is represented by the vertical curve which shows the range of the tilting motion. For most moving lights, the tilt range is not limited when the light is point straight up; lights can tilt either forwards or backwards from center.

Moving lights are controlled by the DMX protocol with a technique called addressing to manipulate their many controllable characteristics. Some of these characteristics include pan, tilt, intensity, and color, for example. Each moving light is assigned a starting address with a panel on the side of the

light. If a moving light has *n* different controllable parameters and starting address *z*, these parameters will then be controlled by z^{th} to the $(z + n - 1)^{th}$ frames of the DMX data that the light receives.

For example, let's consider the DMX addressing scheme that I used for this project. I used a Chauvet Intimidator 100 IRC moving light, and I referenced the user manual to figure out the *n* controllable parameters. The user manual including the addressing information can be found in the references section. Specifically, I used the 6-channel mode with a starting address of 1 (that is, n = 6 and z = 1), so my first six DMX frames were as follows:

1. **Pan**: This was calculated by the project
2. **Tilt**: This was calculated by the project
3. **Color**: I used a value of 90 which corresponded to red in my light; the orange and red colors were switched in my light, so this was a different value than the manual indicates for red
4. **Shutter**: I used a value of 254 signifying that the shutter should be open, not closed or strobing
5. **Intensity**: This value indicated the brightness of the lamp. I set it arbitrarily to 160
6. **Gobo**: I did not want any patterns, so I left this channel with value zero or did not write it at all, which is equivalent to setting it to zero

This project could conceivably be used with any type of moving light with pan and tilt attributes if the correct addressing information was given to the calibration block and the DMX processing block.

Acknowledgments

I would like to thank Peter Torpey for his guidance in refining the original project idea, as well as E33 Productions for letting me borrow equipment. Finally, I would like to thank the 6.111 course staff for the many lab hours spent staring at ModelSim waveforms with me, and for all of their other help, too.

References

My Verilog code has been uploaded to the course website and my github. It will not be included here.

[1] Data sheet for the MAX485 chip, MAXIM Integrated, revised February 1996, accessed December 2014. http://ecee.colorado.edu/~mcclurel/max485ds.pdf
[2] DMX 512-A Cable Diagram, Synapse, accessed December 2014. http://www.synapse-wireless.com/documents/products/Synapse-SNAP-DMX-DMXpro-Quick-Start.pdf
[3] Guidelines for Proper Wiring of an RS-485 (TIA/EIA-485-A) Network, MAXIM Integrated, released November 2001, accessed December 2014. http://www.maximintegrated.com/en/app-notes/index.mvp/id/763
[4] DMX 101: A DMX 512 Handbook, Elation Professional, revised May 2008, accessed December 2014. http://www.elationlighting.com/pdffiles/dmx-101-handbook.pdf
[5] Chauvet Intimidator 100 IRC user Manual, Chauvet, released July 2013, accessed December 2014. http://www.chauvetlighting.com/products/manuals/Intimidator_Spot_100_IRC_UM_Rev1_WO-logo-.pdf
[6] 12x Conference Camera picture, accessed December 2014. http://cdn.shopify.com/s/files/1/0155/0877/t/2/assets/g-ptz-tilt-full.jpg?448