Stoplight Detection and Image Processing with FPGA

A Major Qualifying Project Report

Submitted to the faculty of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements of the

Degree of Bachelor of Science

By:

Michael Derryberry, Electrical and Computer Engineering

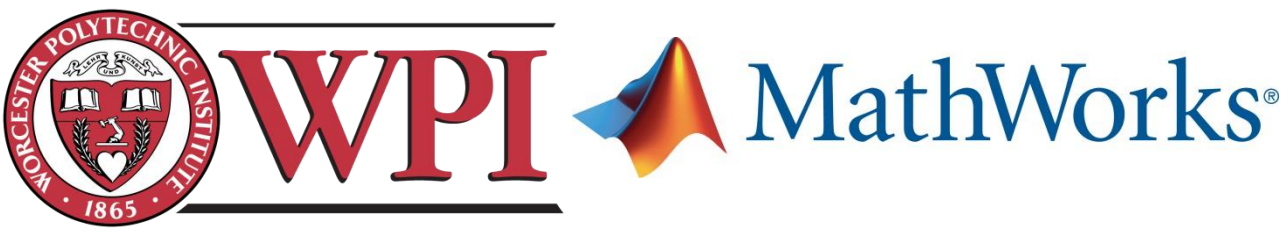
Jeremiah McCarthy, Electrical and Computer Engineering

Submitted to:

Project Advisor: Professor Xinming Huang, Department of Electrical and Computer Engineering

Submitted on:

March 27, 2015



Abstract

This project focuses on real-time stoplight detection for advanced driver assistance system using a Field Programmable Gate Array (FPGA). The main algorithms include Color Filtration, Blob Detection, and Histogram Analysis. In order to reduce the computational complexity of this process, the Color Filtration was to be accomplished by an FPGA while the more complicated Blob Detection and Histogram Analysis was to be accomplished on a microprocessor. The architecture is targeted on a Xilinx Zynq-7000 All Programmable SoC ZC702. A system on chip (SoC) device was selected in order to maximize performance and allow easy transition from the FPGA and the embedded processor on the same device. This implementation accurately detects stoplights and is able to alert the user through both audio and visual peripherals.

Acknowledgements

Our group would like to thank everyone that gave us the opportunity work on this project and to experience the success that resulted from it.

First, we would like to thank our advisor, Dr. Xinming Huang for his guidance and advice on this project. His input and experience helped us to develop the project as a whole and keep us on track to completing it.

We would also like to thank Yuteng Zhou, a graduate student in the Embedded Computing Lab at WPI. Without his extensive knowledge of the Zynq 702 board and Xilinx systems this project would not have been possible.

We would also like to sincerely thank the Worcester Polytechnic Institute (WPI) for providing the facilities and equipment for this research.

Without the assistance of these people, this project would not have been as successful as it was. Thank you!

Authorship

This paper was done in equal parts by both authors; Michael Derryberry and Jeremiah McCarthy. All sections were created and edited as a team.

Table of Contents

Table of Figures

Executive Summary

Automobile use is an integral part of everyday life in modern society. As more and more drivers have entered the roadways the number of injuries sustained from vehicle accidents has greatly increased. A staggering amount of these injuries have been due to ignoring stoplights. Over the past years, many solutions have been suggested, but very few of these are inexpensive, accurate, and fast enough for real-time processing. This project looks to address these issues through developing a system that is relatively inexpensive with real-time performance to assist drivers with stoplight detection in an automobile.

Accurately detecting stoplights is a challenging task due to many factors. The largest of these factors is light pollution and false positives. Light pollution can be caused by other street lights or sunlight causing obscurity of the stoplights in an image. False positives are also a large problem due to many other red objects in the environment such as car tail lights, street signs, and other red entities. Due to these problems, a Histogram Analysis approach was developed to verify that an object in question was indeed a stoplight. This process isolates possible stoplight candidates and takes a histogram of color values in the region. From this analysis, many false positives can be removed because each object has its own unique color distribution. Stoplights are fairly standard and were found to have the same histogram traits, which allowed them to be identified easily. Color filtration and Blob Detection were used in order to find the possible regions of red lights. This project attempts to use a System on Chip, or SoC, solution in order to perform these computations in real time.

Software implementation was first done in Matlab, which has built in support for Color Filtration, Blob Detection, and Histogram Analysis functions. This simulated code was performed to prove the viability of the algorithm and to check its accuracy. From there, the code was customized to meet other system requirements and the Blob Detection and Histogram Analysis were implemented in

C. This allowed the software to work faster and be usable on the microprocessor. This algorithm was tested using video streams recorded by dash board cameras driving around Worcester, MA. The final project was able to accurately identify red lights at approximately eighteen frames per second.

Hardware implementation was done using the Xilinx 7000 ZC702 evaluation kit. The algorithm was split into two major parts; basic filtration and advanced image analysis. The basic filtration was the most computationally expensive part of the algorithm, due to it looking at every pixel in the image multiple times; therefore, it was implemented in Verilog with Xilinx ISE Design Tools to be performed by the field programmable gate array, or FPGA. This system was tested by using a laptop connected to the board through a HDMI IN port on the board. The laptop streamed the dash cams video to the system, which then displayed the processed video on a display through a HDMI OUT port. Before the final output displayed, the project searched for and highlighted the stoplights. There were also user peripherals that were added that activated when a stoplight was detected. These included visual signals such as an LED and audio signals from a buzzer. In the future, this project would be adapted to work with a HDMI camera rather than a laptop and would be a part of an all-encompassing vehicle vision detection device. This project was meant to be just one of many parts to an advanced driver assistance system.

1. Introduction

1.1. Stoplight Detection

Stoplight detection is a problem that has been examined before for image processing. With the rise of vehicle assistance systems, detecting stoplights is an obvious choice. Many solutions have been presented, but very few of them do so in real time. Vehicles move at high-speed in a variety of environments, so any solution that is presented must operate in real time with high accuracy in a variety of situations.

There are thousands of accidents at intersections that cause damage or injuries every year. In the United States alone from 2007-2011, there were an average of 751 deaths and 165,000 injuries due to drivers running stoplights (U.S. Federal Highway Administration, 2014).

The major problem with creating a stoplight detection device is the speed at which it has to work. Many factors play into whether an object is a stoplight or not, and the system has to analyze these and make decisions extremely quickly. The average human reaction time is about 262 milliseconds (Human Benchmark, 2015), which means that if a car is traveling at thirty miles per hour, it moves about ten feet in the time it takes someone to react. Therefore, a system for driver assistance would need to work extremely fast in order to have a recognizable effect.

There are also many problems with classifying stoplights. Stoplights come in a variety of shapes and sizes. There is also no standard area that stoplights must be placed on the road. They can be directly ahead and above of the driver, to the side of the driver, or almost anywhere else in the driver's field of view. Light pollution also causes problems when classifying stoplights, as it saturates the color of the light which could cause a system to not register it. Another problem is other lights in the area, such

as brake lights from cars or street lights on the road. These lights can be confused as stoplights to a system with poor classification.

1.2. FPGA/SoC Systems

Real time image processing is an extensive task that needs the right hardware to be implemented. In recent studies, it has been proven that using a Field Programmable Gate Array (FPGA) is an efficient method of image processing, as opposed to using a microprocessor (Sparsh Mittal, 2008). An FPGA is an integrated circuit that is designed to be configured after manufacturing (Altera, 2015). Since the need for high speed performance has been established, the project cannot be run just on a processor because they are not fast enough (Sparsh Mittal, 2008). As explained in Mittal's journal article, real-time video rates of twenty five frames per second require about 33 million operations per second. A microprocessor cannot complete this many operations in such a short time, but the FPGA can because of its ability to do parallel processing. A study completed by National Instruments on the processing speed of FPGAs and microprocessors determined that an FPGA can deliver a solution many times the processing power per dollar in some applications (National Instruments, 2013). Therefore, an FPGA was selected to be used for this system as they have been proven to be useful in image processing projects.

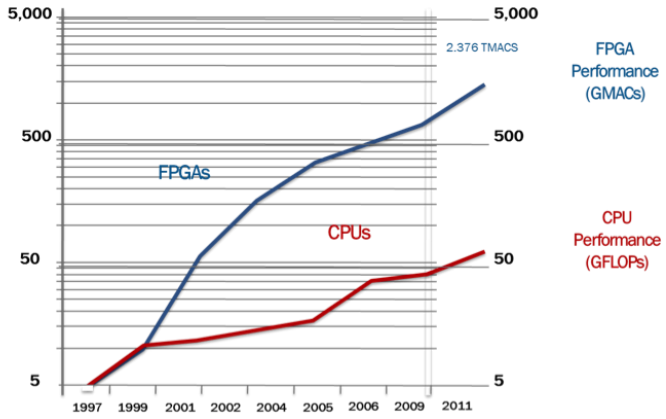


Figure 1: Comparison of Processing Power - FPGA vs Microprocessor (National Instruments, 2013)

FPGAs are extremely useful for tasks that need to be completed quickly. Applications for FPGAs are written in the hardware language VHDL or Verilog, which are different from other languages such as C or Java. Some FPGAs now come with an ARM processor. These systems are called System on Chips (SoC) and this project was chosen to include one. The benefits of a SoC are that it allows for a more customizable product because it gives extensive control over hardware, software, and I/O configuration (Xilinx Inc., 2015). They also allow for an increased system performance and reduced power consumption.

2. Algorithm

This section discusses the details of algorithms that were used to create the overall stoplight detection system. It will explain how each algorithm works, as well as why it was chosen for this application through example. The final implementation was based on simulations performed in Matlab, using the video and image processing toolboxes; however, due to different components of the algorithm being split between the hardware and software some functionality was modified in the final system. The procedures used consisted of basic image filters, such as color detection, and more advanced image processing techniques, such as blob detection.

2.1. Basic Image Filters

Three basic image filters were used in this system to perform the task of preprocessing each frame for further use. These basic filters performed simple tasks that were context free, meaning surrounding pixels did not matter, and required only discrete pixel values. The three filters were used for color extraction, grayscale conversion, and color inversion.

2.1.1. Color Extraction

Color extraction is a commonly used image processing technique that separates sections or components of an image based on hue. In the RGB color model shown in Figure 2, each pixel is composed of a 24-bit value. This value is split into three bytes, with each byte representing the value of red, green, and blue light that additively composes the pixel's color.

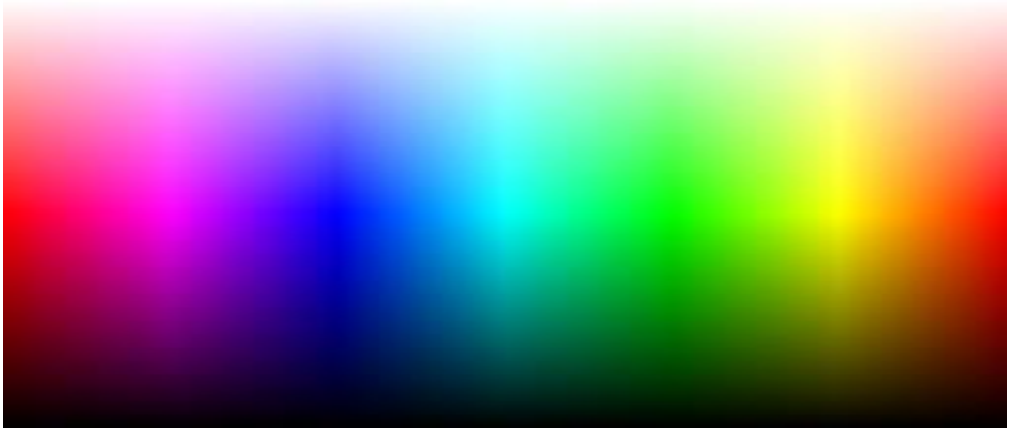


Figure 2: RGB Color Scheme (Phanomeme)

The color filtering was performed by either passing or rejecting certain values in each of the three RGB composing bytes. The appropriate passing bands were determined through Matlab simulations on static images, as shown in Figure 3 (See Appendix A for test code). The GUI developed for this testing had six user editable fields. These fields allowed the user to change the lower and upper pass band limits for each of the three RGB components. The Matlab code would check the RGB components of each pixel, changing them to a value of 0 if they did not fall inside the specified pass band.

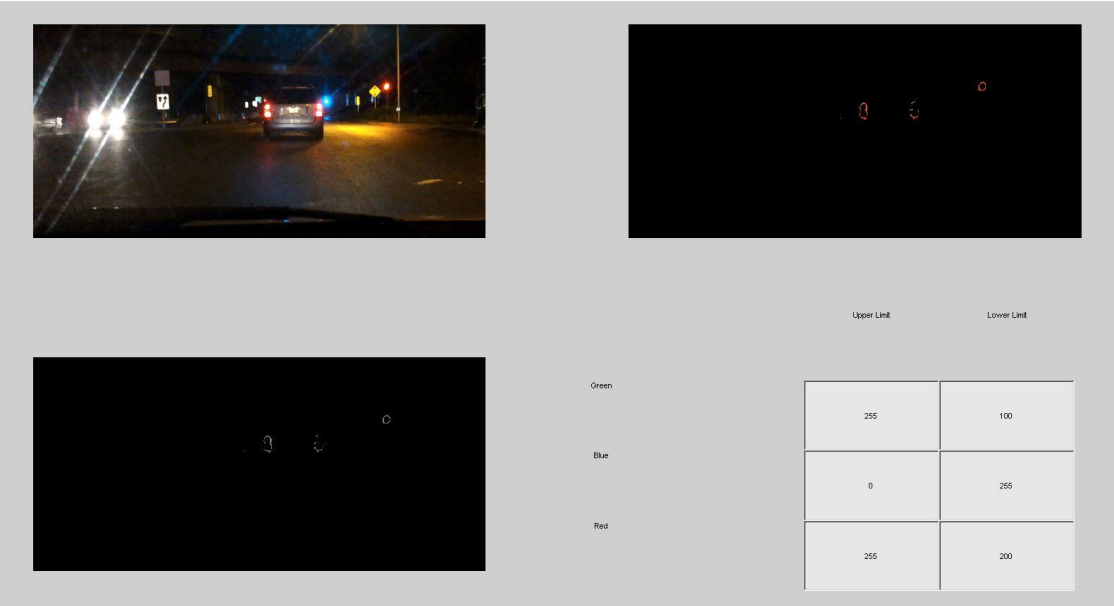


Figure 3: Matlab Color Extraction GUI

2.1.2. Grayscale Conversion & Color Inversion

The next basic filter in the system converted the color filtered image into its grayscale representation. Grayscale is a color encoding scheme which contains only information on intensity. This color structure is composed exclusively of shades of gray ranging from black, the weakest intensity, to white, the strongest. The major benefit of grayscale is that each pixel value can be represented by a single byte, instead of the three byte structure the RGB encoding required. This cuts the amount of processing that would need to be done later by 66% since a single byte now holds the relevant information that three bytes did before. This conversion was performed using the Matlab function rgb2gray, which takes an image that uses the RGB color map and converts it to grayscale.

Figure 4: Grayscale Color Scheme (Think Silicon)

The major downside of grayscale is the loss of hue and the inability to recover the original RGB components. This is not a downside for this application since the colors of interest were the only existing RGB components before this conversion, due to the prior color filter.

After the conversion to grayscale, the final basic filter converts the background black pixels to white. The purpose of this conversion is for the blob detection that will be described later in this section. To perform this conversion, each pixel value on the grayscale image is examined. If the pixel is solid black it is changed to white. This had been added to make the background more visually distinct during development, and was only used in the final implementation.

The final results of the basic filters are shown stage by stage in Figure 5. Each of the basic filters requires a full pass over each pixel in the entire 1920 by 1080 frame. Additionally, they are all computationally easy to perform. Each basic filter requires single value comparisons and a single variable change.

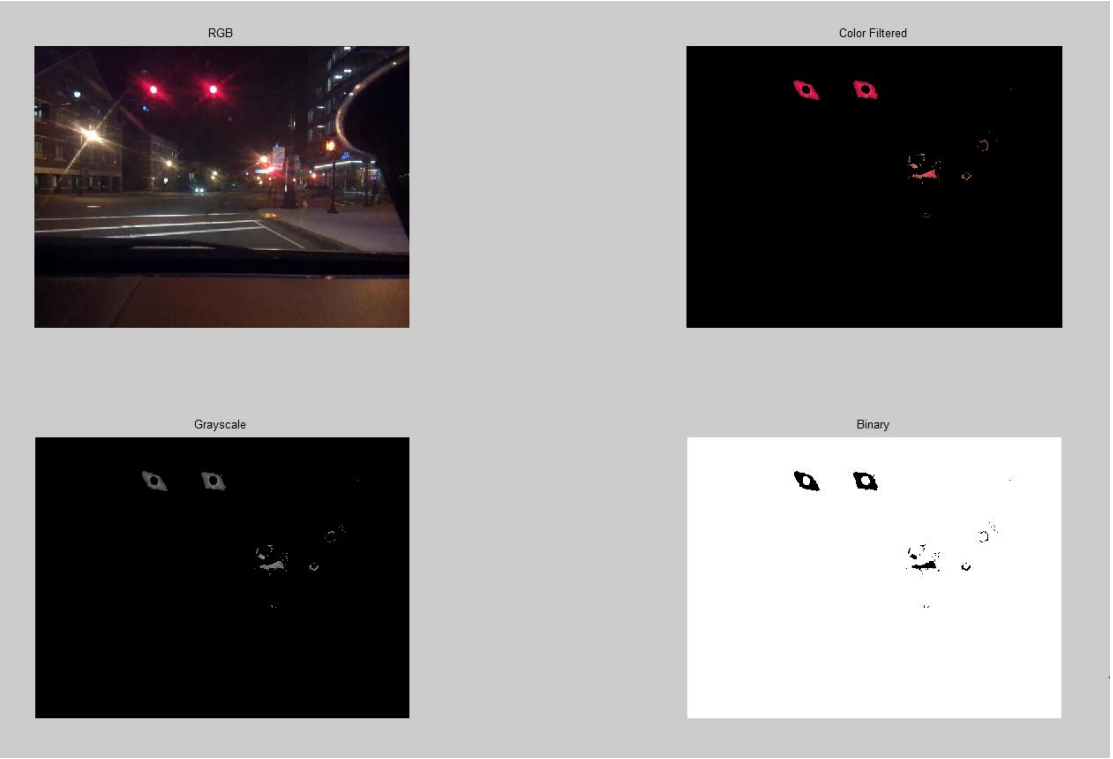


Figure 5: Basic Filters

2.2. Blob Detection

Once the image was preprocessed, the next step was to determine the regions of interest through the use of blob detection. A region of interest, or ROI, is an area of an image that has been identified for a specific purpose. For this application, the ROIs were the areas of an image that could potentially be stoplights. Blob detection is an image processing technique that is used to identify regions of an image that possess certain qualities compared to the surroundings. A blob is a region of an image that has a common property.

Matlab implements blob detection in the “regionprops” function. This function takes an image and calculates sets of properties for each connected blob, or object, it finds. To implement this in Matlab, a few additional side steps are required using the “regionprops” function. The image was converted into a binary image using the grayscale value of 51 as the cutoff value. This meant that each grayscale pixel that was greater than 51 in value was set to 0, and all others set to 1. The “regionprops” function was used on the created binary image to find all of the blobs.

Additionally, for debugging purposes, the function “bwboundaries” was used to highlight the regions of interest. The bounding boxes were determined from the binary image, however, the boundaries were drawn on the corresponding location in the original image, see Figure 6.

Figure 6: Bounding Boxes on Original Image

2.3. Image Histograms

The final stage in the algorithm was deciding if the identified ROIs were stop lights or not. In the final implementation, a histogram analysis was used. A histogram of an image is a representation of the distribution of pixel values. The distribution is calculated by setting up a number of bins. A bin stores

the number of times a pixel occurs in a set range of values. Having a large number of bins increases accuracy for analysis at the cost of space in memory to store the necessary values. For the simulation code, 256 bins were used, one for each pixel value on the entire grayscale spectrum. Next, every pixel in each of the regions of interested were examined and classified into their corresponding bin. To perform this task, the "imhist" function was used. This function takes a grayscale image and calculates the histogram with 256 bins by default. The results of examining a single frame of a video are shown in Figure 7.

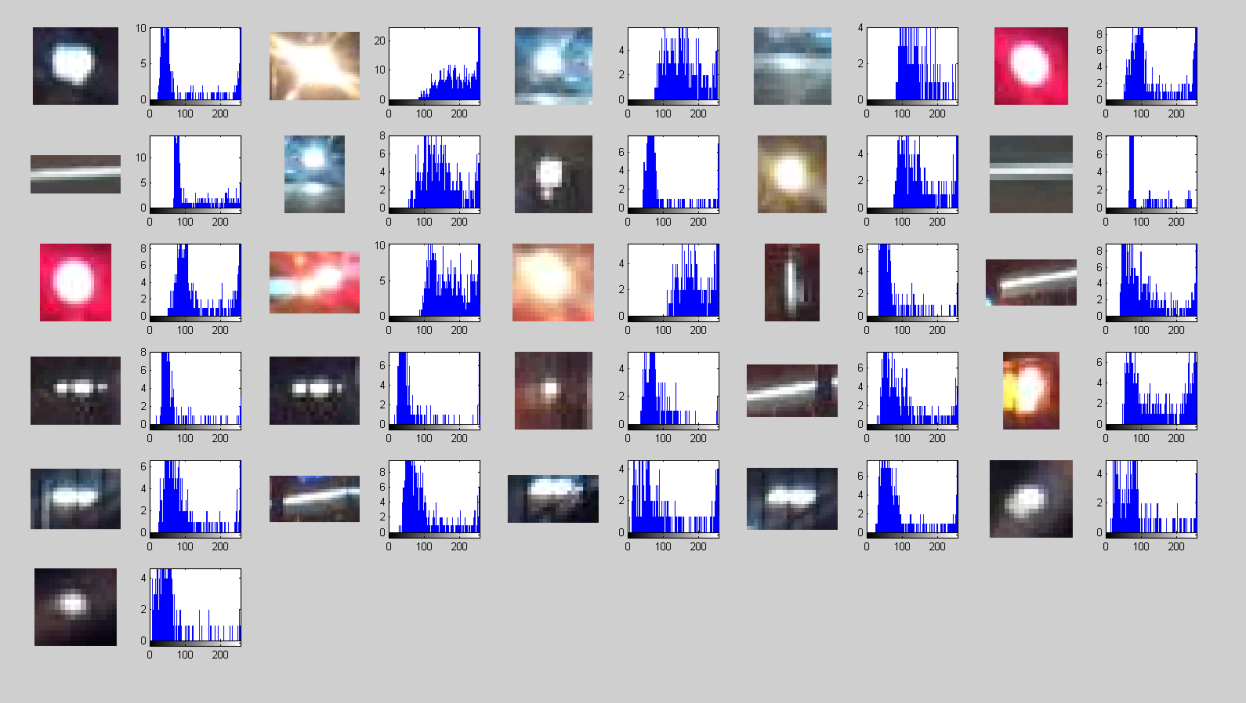


Figure 7: Example Histograms of Potential Stoplights

The next step in the histogram analysis was to determine what parameter checks were required to determine if the region was a stoplight or not. It was observed in the original tests that stoplights exhibited a high concentration of grayscale values in the 50-100 range. Additionally, they tended to include very few values below 50, as well as low intensity in the range of approximately 100 to 250. To determine the stoplight characteristic cutoff values for each bin, a histogram with the average of 30

stoplights was produced. The small sample size of 30 was chosen instead of a larger set because each stoplight had to be hand-picked from the data sets to ensure that only true stoplights were being factored into the histogram. The results of this data collection, shown in Figure 8, indicated that the original observations on stoplight histograms held true. There were no values fewer than 50 observed, a high concentration of values around 100, and a low concentration located approximately within 120 and 250.

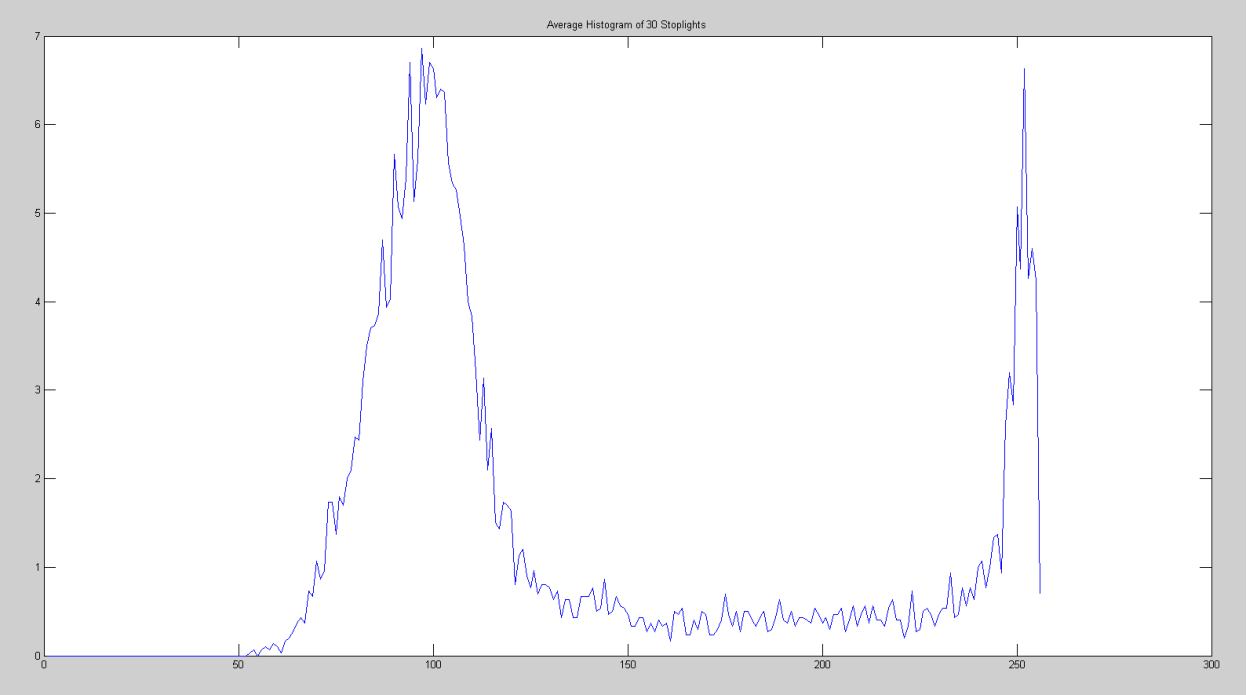


Figure 8: Average Histogram of 30 Stoplights

From the results of the 30 stoplight histogram, a brute force stoplight identification algorithm was created. This algorithm would be used on each object that was found by the blob detection. For the simulation code only 4 bins were used. The number of bins was chosen since there were four distinct areas in the histograms. Each bin was given a requirement based on previous observations of stoplight histogram characteristics. The bins are summarized in Table 1-1.

Table 1-1: Matlab Simulation Histogram Bins		
Bin	Grayscale Range	Stoplight Requirement
Hist0	0-50	<5
Hist1	50-110	>100
Hist2	110-240	<50
Hist3	240-255	>35, < 75

The resulting algorithm was tested over three videos of night time driving. An example of a single frame from one of these test videos is shown in Figure 9. In this image, the ROIs are drawn on the original frame to the left. It was observed that in this frame there were many blobs identified as possible stoplights. These blobs consisted of two stoplights and some other stray red lights including the taillights of a distant car and crosswalk signals. On the right hand side of this figure are the blobs that had been determined to be stoplights based on the histogram analysis. The analysis not only identified that there were two stoplights, but also filtered out the non-stoplight blobs, thus removing false positives.

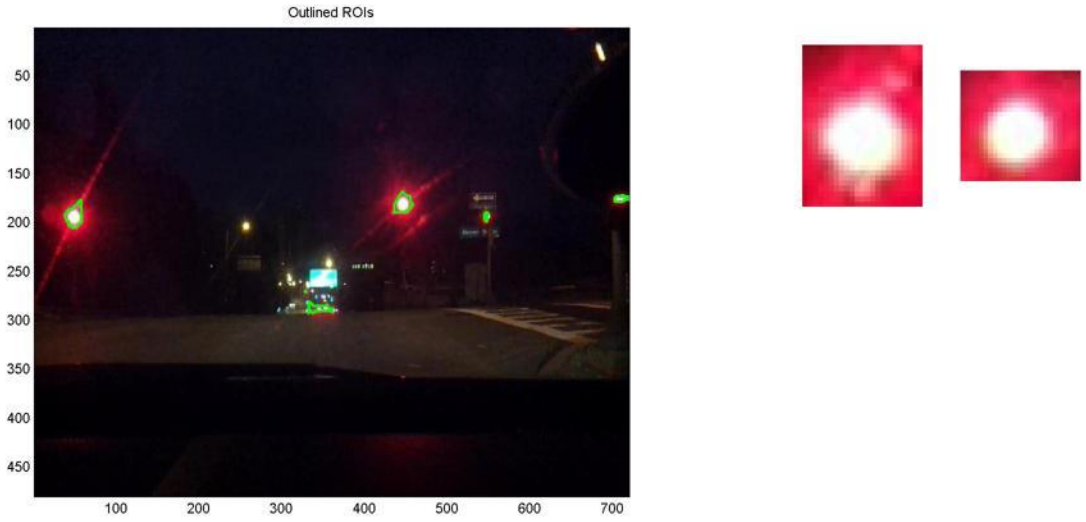


Figure 9: Final Algorithm Testing Example

2.4. Additional Simulation Tests

There were numerous temporary simulation versions created to test the effectiveness of other image processing methods. The goal of this testing was to find a way to increase the efficiency or accuracy of the overall algorithm. This section will discuss many of the alternative approaches tested for each stage of the final implementation.

2.4.1. Alternate Color Schemes

Since the algorithm required a conversion to the RGB color scheme, tests were performed on the hue-saturation-value, or HSV, format to see if results were more accurate. The HSV color scheme is a cylindrical-coordinates representation of the RGB format, as shown in Figure 10. The hue holds color information, and is represented by an angle around the z-axis. Saturation holds color intensity information, and is represented as the radial distance from the z-axis, or a radius within the HSV cylinder. Lastly, value holds brightness information, and is represented as the z-value or height within the cylinder.

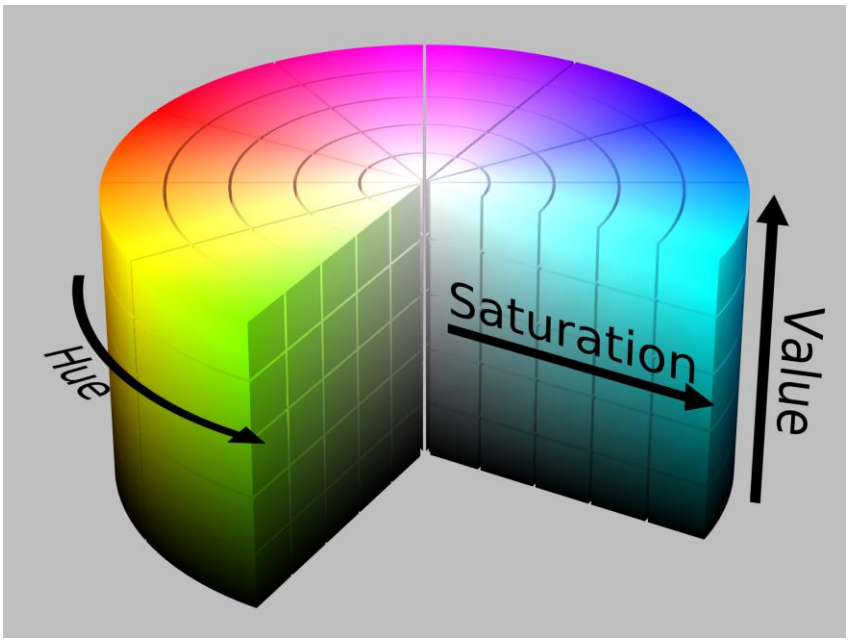


Figure 10: HSV Color Scheme (Wikipedia)

The idea behind the algorithm was to identify blobs from the saturation value, since it was observed that red lights appeared as high intensity circles surrounded by a low intensity halo. It was found that the identified blobs were no different than the results of the RGB testing, as shown in Figure 11. Due to this, the RGB algorithms were used since that format was readily available in the FPGA implementation that had already been designed.

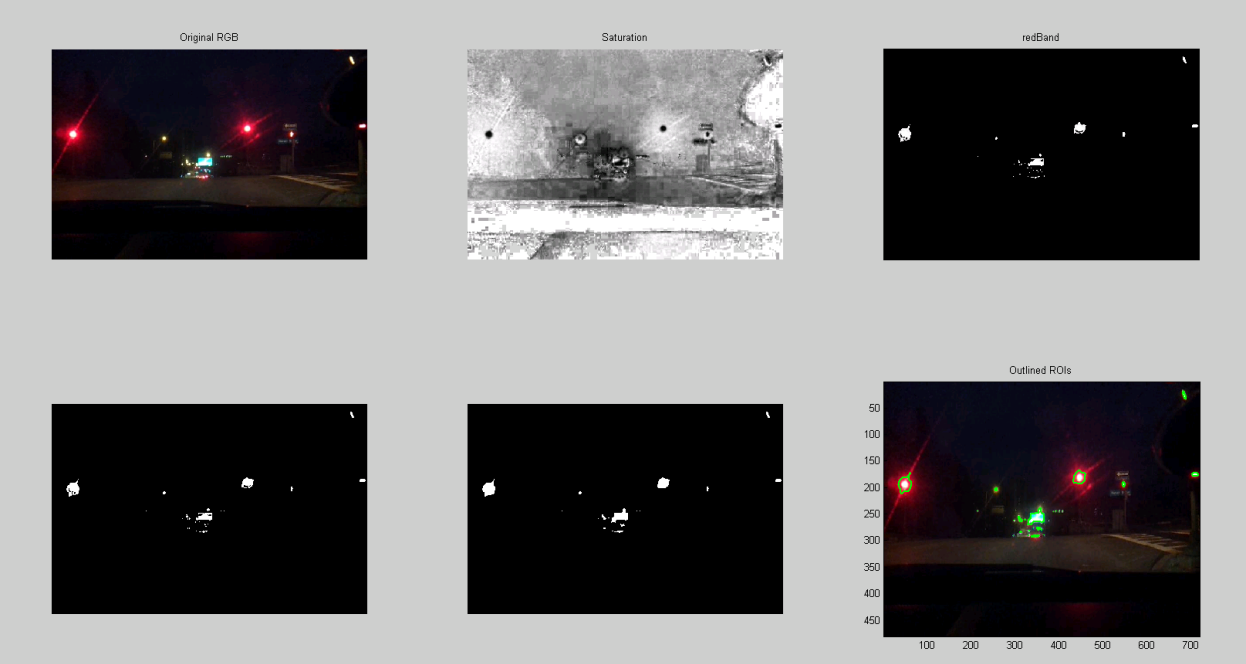


Figure 11: HSV Test Results

2.4.2. Blob Detection Accuracy

Before the blob detection was performed, one version of the code filled the "holes" inside the binary image. A "hole" is a white region in a binary image completely surrounded by black. This test was accomplished using the Matlab function "imfill". This was tested to see if the blob detection accuracy would increase with the holes of the binary image being filled. Another version attempted to increase this accuracy by filtering regions of circular objects in the binary image. This shape testing was performed with the Matlab function "strel" and "imclose". These functions would fill in the circular objects in the binary image to ensure that the blob analysis would find each stoplight. After testing

with both of these methods, it was determined that they did not improve the accuracy in detecting the regions that were stoplights.

These additional stages most likely did not help the final algorithm because the color filter already removed a significant amount of the unwanted sections of the image. The remaining image consisted primarily of red lights and some noise. Since some unwanted red lights such as taillights are circular on some vehicle models, the additional shape checking provided no additional benefit. Filling in the holes did not increase accuracy since stoplights always retained their shape after the color filter in the video feeds tested. These additional stages were dropped from the final implementation to reduce the amount of computation.

2.4.3. Stoplight Identification

The final tests performed were to attempt to increase the accuracy of the stoplight verification. These tests consisted of adding additional requirements for a blob to be flagged as a stoplight. The first requirement tested was eccentricity. Eccentricity can be used as a measure of how circular an object is. Since stoplights should be circular, verifying that the eccentricity of a blob was between zero and one, meaning that the object was between a circle and eclipse in shape, was believed to increase the accuracy in stoplight identification. It was found that there was no increased accuracy, most likely due to the same reasons as the shape testing done previously.

The other supplemental assessments to verify that an object was a stoplight attempted to increase the distance in which a stoplight could be identified. Since the histogram approach was based on a discrete number of pixel values, stoplights that were far away, and thus being few pixels in size, would not be seen until the driver moved closer. This test attempted to fix this by having the histogram requirements normalized by the area of the blob. Due to the testing being done at 640 by 480 pixels at the time, this failed to increase accuracy since the stoplights were too small.

After testing with these methods, it was determined that the final results of the system was not affected, thus both additional stages were dropped from the final implementation to reduce the amount of computation.

3. Hardware Design

In this project, a system was used that integrated both hardware and software solutions via a SoC system. The hardware design was accomplished on a FPGA which allowed for real-time processing. The approach used various modules to receive an incoming 1080p video and convert it to a format that was suitable for the project. Multiple filters are then applied to the image in order to attempt to isolate potential stoplights. Specifically, a red color filter, grayscale filter, and inversion filter were used to set the image up for the next steps of the algorithm, most notably blob detection. The hardware implementation was performed on a Xilinx ZC702 Evaluation Kit.

3.1. FPGA Selection

3.1.1 Altera DE1-SoC Development Kit

The first FPGA that was considered was the Altera DE1-SoC Development Kit. This kit is built around the Altera SoC FPGA, which combines a Cortex-A9 processor with programmable logic to increase design flexibility (Altera, 2013). It also includes Altera's design tools such as Quartus II Design Software and the Qsys System Design Tool. The DE1-SoC also boasts a variety of features including 1GB DDR3 SDRAM, 64MB SDRAM and an 800MHz processor. The combination of the Altera Cyclone V FPGA and Cortex-A9 processor make this device a suitable candidate for a real-time embedded image processing project.

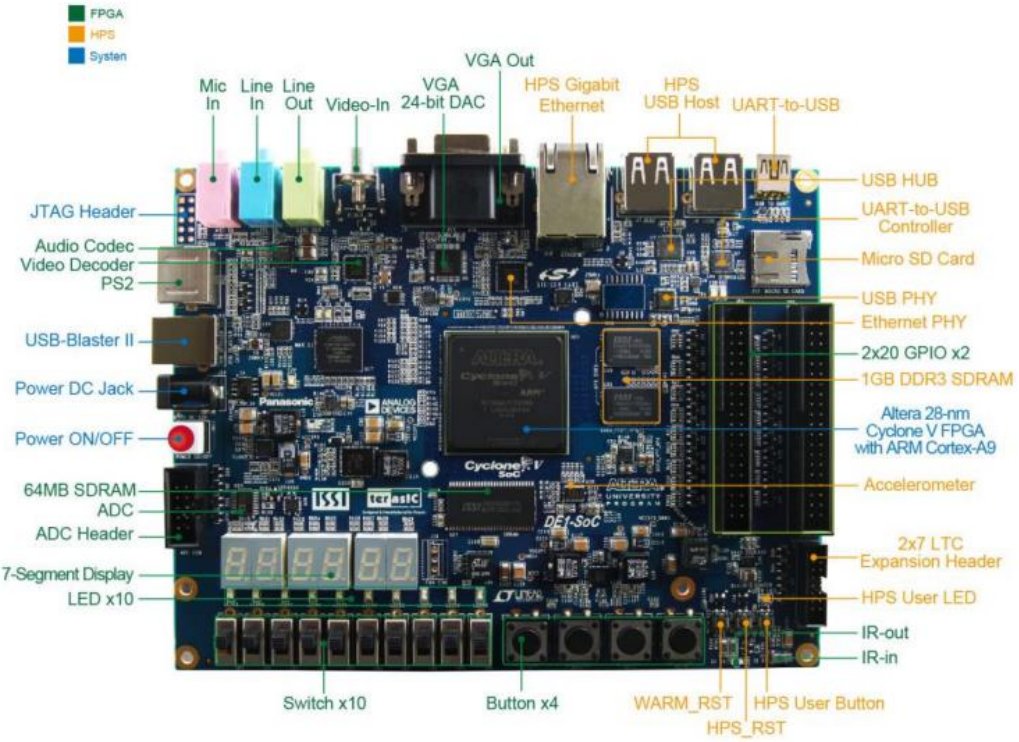


Figure 12: Altera DE1-SoC

The Quartus II Design Software that is included in the Altera DE1-SoC Development Kit is a FPGA integration tool. This software enables analysis and synthesis of HDL designs, which allows developers to compile their projects, perform timing analysis, and simulate a design. It also enables the developer to configure the target device with the program and load their project to the board (Altera, 2014). In Figure 13, the Quartus II software can be seen. The user interface includes the text window where files can be viewed, the command window where errors and messages can be seen, a project navigator that shows all the files in the project, and the compilation pane which allows compilation of the project and shows progress.

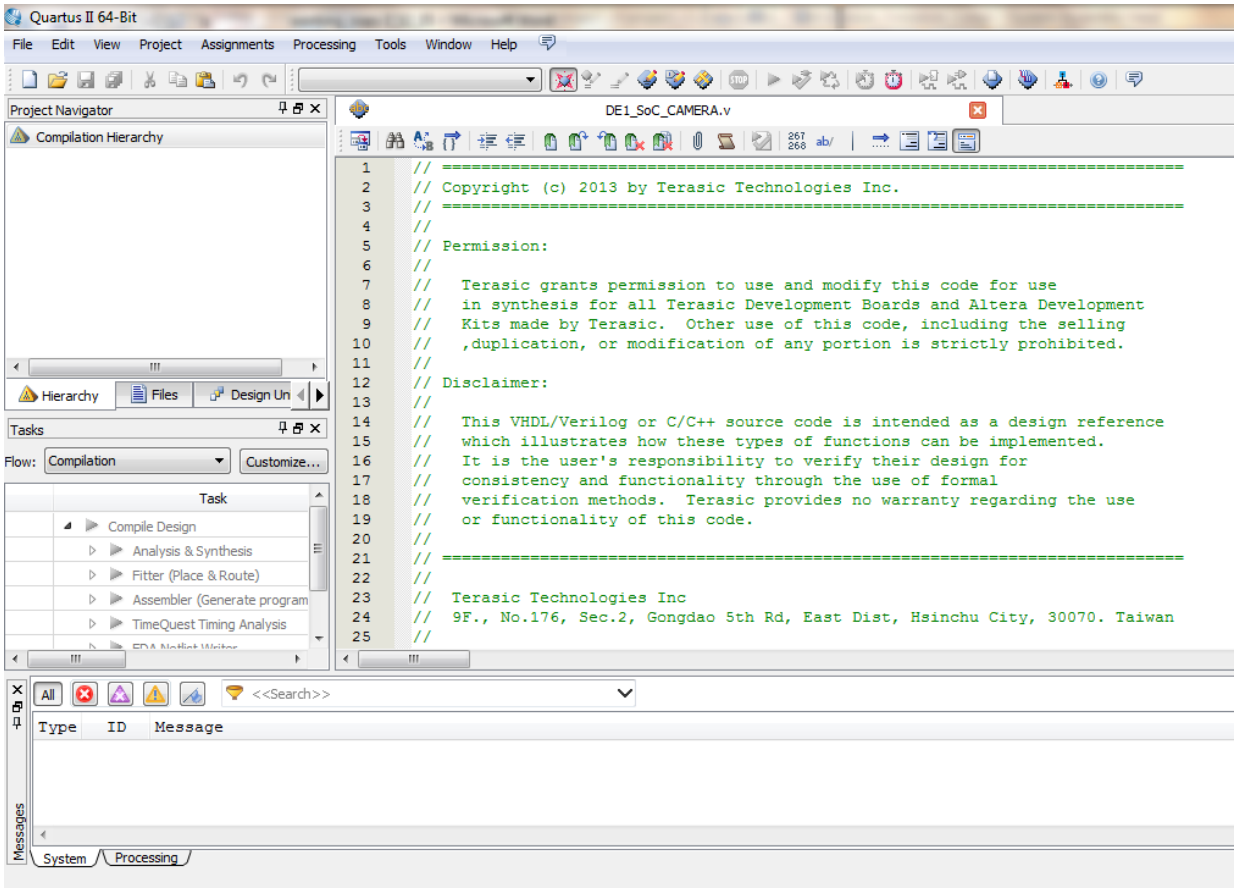


Figure 13: Quartus II User Interface

The Altera DE1-SoC Development Kit also comes with Qsys System Design Tools. Qsys is a program that automatically generates logic to connect intellectual property (IP) functions and subsystems. This makes the FPGA design process much easier and faster than. In Figure 14, the Qsys user interface can be seen. Qsys allows a developer to select IP cores that have been generated from the IP Catalog window. Once an IP core has been selected, it has been added to the system and will appear in the System Contents window. This window shows all included IP cores and their corresponding connections. From there, a developer can connect them how they see fit to customize the project and then generate FPGA logic automatically based upon the system design.

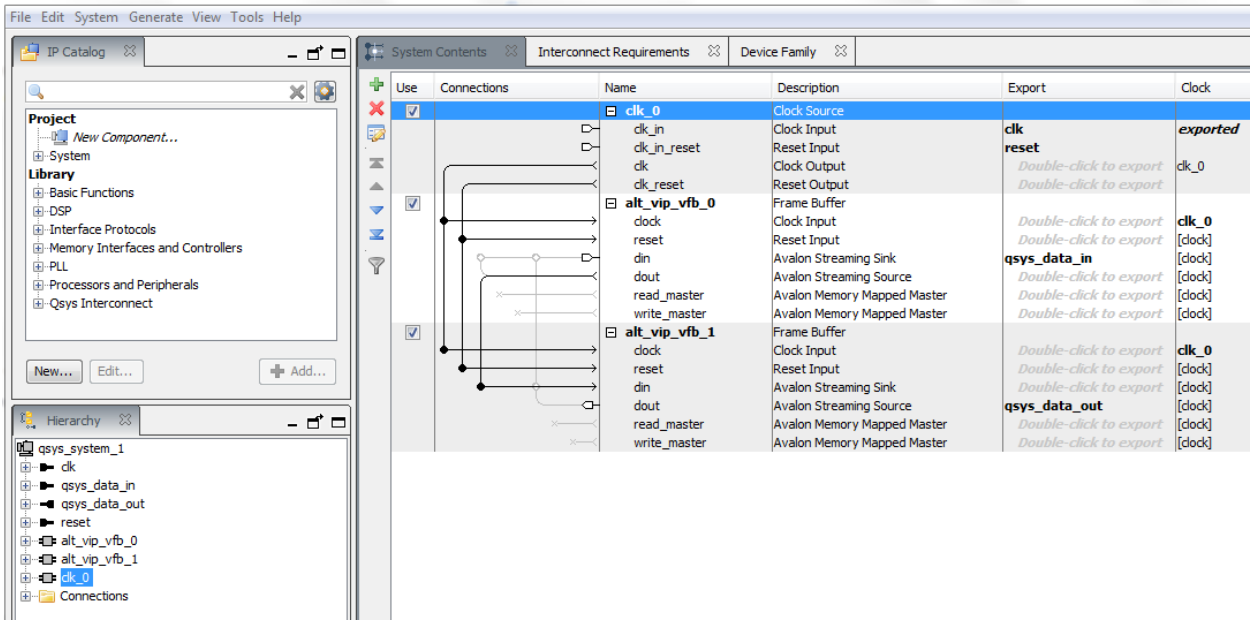


Figure 14: Qsys User Interface

While all of these tools point towards an appropriate solution to a real-time embedded platform for stoplight detection, this was not actually the case. While Quartus II and the DE1-SoC were suitable for the project, it was discovered that Qsys was not. There is very little documentation on Qsys available to the public and what is available is not very detailed. Due to this lack of readily accessible information, using Qsys in order to create a project that would accomplish the goal was an extremely difficult task. Therefore, a new board and development kit was sought out in order to streamline the development process of the project.

3.1.2. Xilinx ZC702 Evaluation Kit

The Xilinx ZC702 Evaluation Kit provides developers with a complete platform including hardware, development tools, pre-verified reference designs, and IP. It also includes the Xilinx ISE Design Suite, which is ideal for developing embedded systems on a Xilinx FPGA. The Zynq 702 FPGA also boasts an ARM dual-core Cortex-A9 processor. These components are complimented by a variety of features including a maximum frequency of 667 MHz, 85000 logic cells, 53200 LUTs, 560KB of block ram,

1 GB DDR3 DRAM, a variety of user GPIO, and compatibility with a variety of peripherals (Xilinx Inc., 2014). By combining the power of an ARM processor with FPGA programmability makes this device ideal for a real-time stoplight detection application.

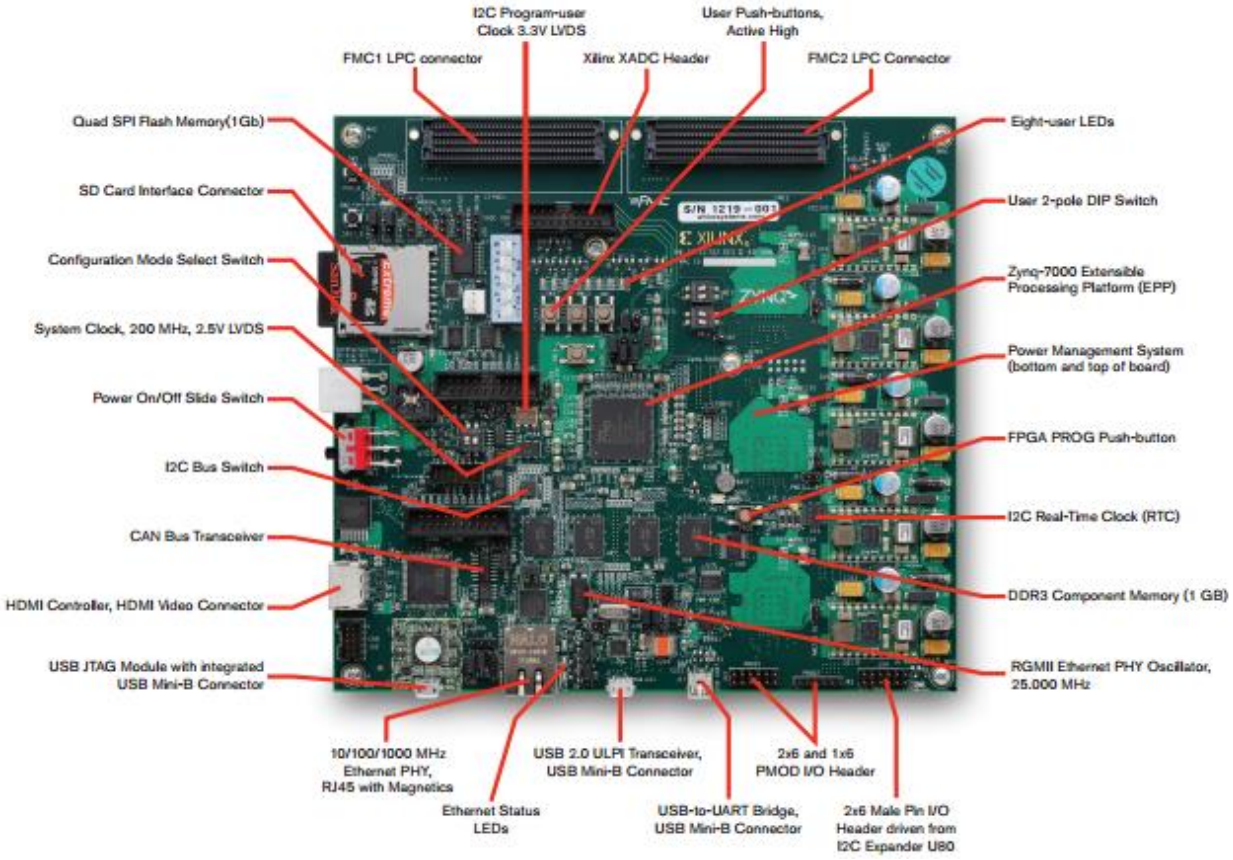


Figure 15: Xilinz ZC702 FPGA

The Zynq 702 FPGA can be programmed using tools from the Xilinx ISE Design Suite. The first tool of the design suite is the Xilinx Platform Studio (XPS). XPS allows developers to build, connect and configure embedded processor-based systems through the use of graphical design views and sophisticated wizards (Xilinx, Inc., 2015). Much like Altera's Qsys system, XPS makes the FPGA design process much simpler and faster. However, XPS has much more documentation for image processing projects which makes it more of an ideal candidate for this project. The user interface for XPS can be

seen below. It is extremely similar to Qsys in that it has an IP catalog, all of the current IP cores in the system and their connections to other IP cores.

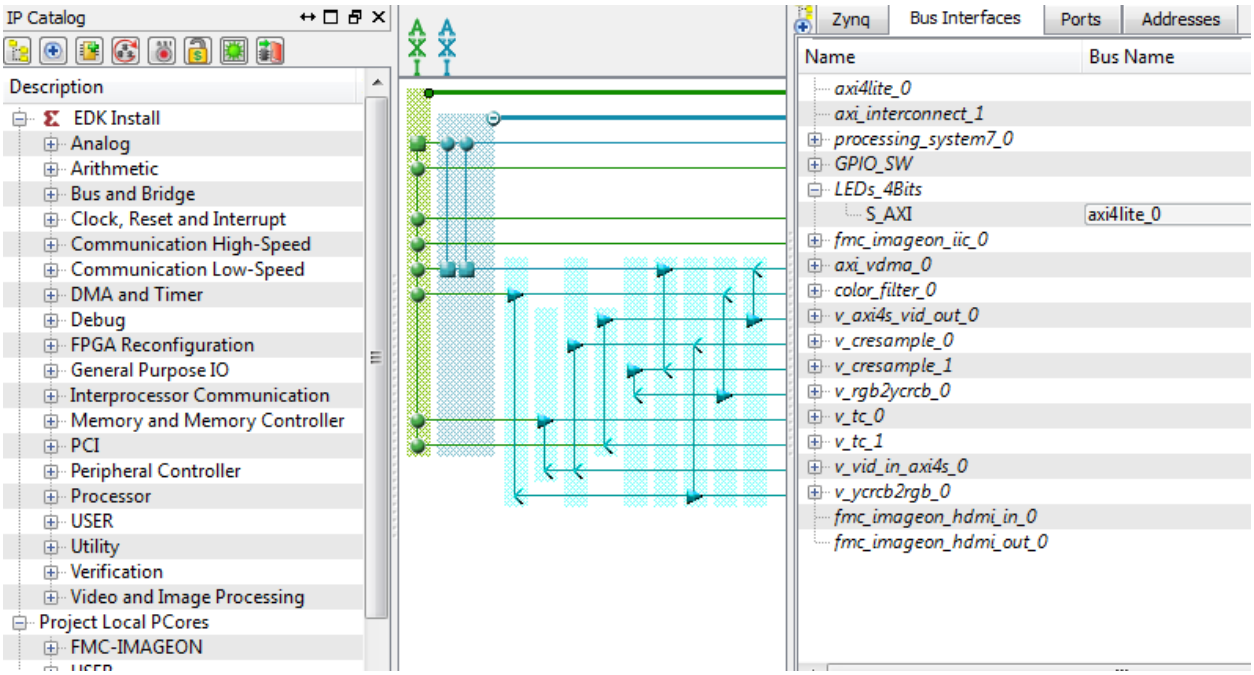


Figure 16: XPS User Interface

Another tool that comes with the Xilinx ISE Design Suite is the Xilinx Software Development Kit (SDK). The SDK functions in a similar manner to Eclipse in both functionality and appearance. The user interface can be seen in Figure 17. In the SDK, the developer can write code in C or C++ to the processor in order to accomplish complicated tasks. The SDK generates header files for all of the port connections in the IP cores and all mapped pins on the board. This allows the SoC to interact directly with the FPGA. Due to the easy to use design suite that accompanies the Xilinx ZC702 Evaluation Kit as well as the physical capabilities of the board, this platform was chosen to accomplish a real-time embedded stoplight detection system.

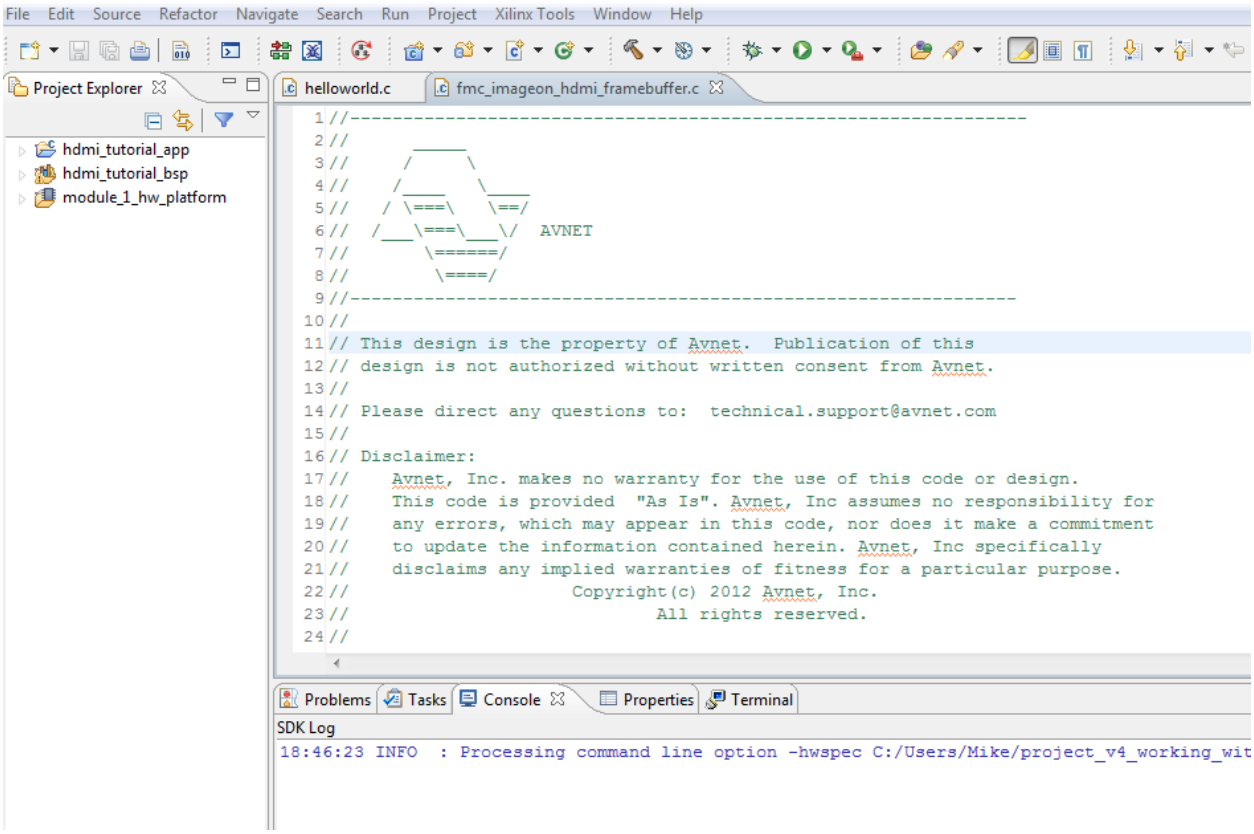


Figure 17: Xilinx SDK User Interface

3.3. FPGA Block Design

Now that the board was selected to implement the project on, a general block diagram was developed. The first steps in deciding on the hardware design came from a tutorial designed by Avnet Electronics to create an HDMI pass through (Avnet Electronics, 2013). A pass through is a simple application in which an HDMI image is input and then output without any changes to a monitor or other display device. In order to accomplish this, the Avnet HDMI Input/Output FMC Module was used. This module provides high-definition video interfaces to baseboards and allows HDMI video sources to provide video content and HDMI output to display any FPGA driven video content (Avent Electronics,

2015). The Avnet HDMI board was able to connect directly to the Xilinx ZC702 through one of its GPIO banks.

Figure 18: Avnet HDMI Daughter Board Connected to Xilinx ZC702

The tutorial provided a block diagram for the pass through which was successfully implemented on the ZC702 using a laptop as the input and a monitor as the output. That block diagram can be seen in the figure below. This system was implemented using the Xilinx Design Suite, specifically XPS. Two IP cores were used in this design; the HDMI input and HDMI output blocks. An AXI I2C module was also implemented (not shown) which allowed the FMC-IMAGEON module to be manipulated. The input and output blocks were interfaced to this module. This test was the first step taken in the image processing project as it showed how to take in video content and view the content on a different monitor. It also gave the basis for how to manipulate the video content even though it was not done in this tutorial.

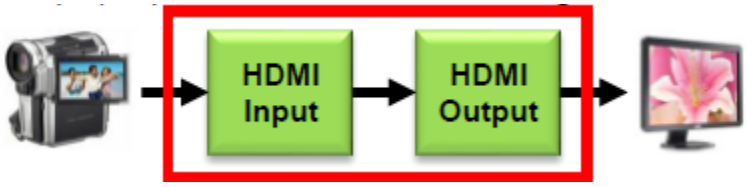


Figure 19: HDMI Pass Through Block Diagram

After completing the HDMI pass through tutorial, the final system block design was designed. It was built using the pass through tutorial as a base, but with more IP cores added to the system. The total design can be seen in the figure below with the added modules inside of the red box.

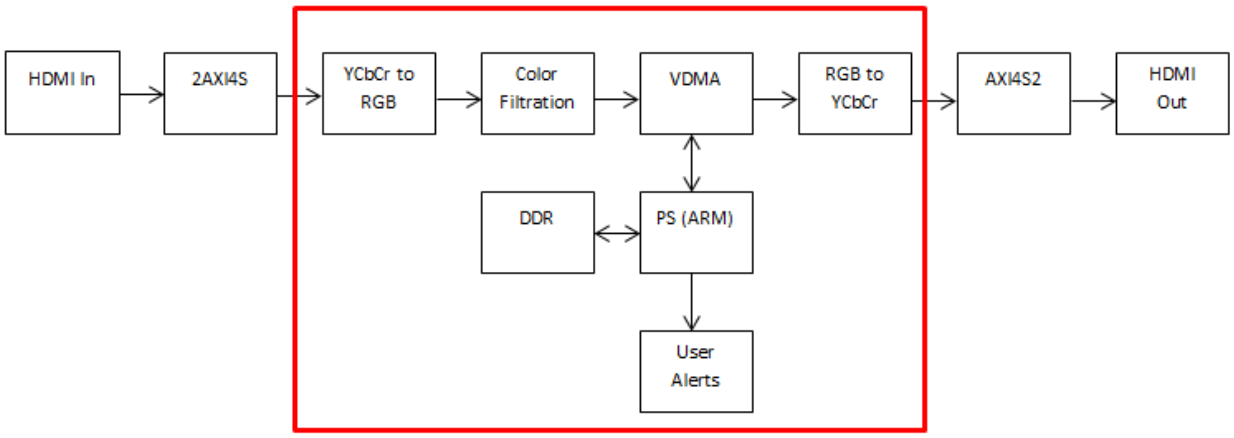


Figure 20: Proposed Block Diagram

The system begins similar to the pass through tutorial with the HDMI in block and 2AXI4S module. This takes in the video content and configures it such that the ZC702 can integrate and manipulate it. The next block that was added to the system was the YCbCr to RGB module. There are two standard image formats that are commonly used; YCbCr and RGB. YCbCr describes the luma, or light intensity, aspect of the image and the red-difference and blue-difference of the pixels. HDMI uses YCbCr by default for images which means that the input video content is in this format. RGB format breaks the pixels of the image into three channels representing the red content, green content, and blue content. RGB is a much easier format to use for color extraction, which is one of the steps used in the algorithm, and therefore a conversion had to be made. The conversion from YCbCr to RGB is fairly simple. It consists of matrix mathematics on the three channels of the YCbCr, which is defined in Figure 21.

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1.000 & 0.000 & 1.400 \\ 1.000 & -0.343 & -0.711 \\ 1.000 & 1.765 & 0.000 \end{bmatrix} * \begin{bmatrix} Y \\ Cb - 128 \\ Cr - 128 \end{bmatrix}$$

Figure 21: YCbCr to RGB Conversion

3.4. Color Filtration

At this point, the video content has been converted to RGB format and it is ready to be manipulated in order to find stoplights. The next module in the block diagram is the color filtration IP core. This module is used to do red color extraction, gray scale conversion, and inversion. The first step is color extraction. The goal of the project is to see red stoplights and therefore the only pixels that matter are those that are red. The video content is passed into the module and then each pixel is examined to see if its RGB values are within a specific range of values. Based upon the original Matlab testing, red lights were considered to be any pixels that had an R value greater than 150, a G value less than 110, and a B value less than 110. If a pixel meets these specifications, it is left as it was. Otherwise the pixel is set to black, which is a value of zero for all R, G, and B values.

The next step of the color filtration module is the grayscale conversion. This takes in the video image that is all red and black pixels and converts it to a grayscale image. This is done in preparation for the histogram analysis that occurs later in the algorithm. While in RGB format, each pixel has three channels that are each 8 bits, or 24 bits in total. However, when in grayscale format each pixel only has one channel that is 8 bits long. If the original RGB image was used later in the algorithm for histogram analysis, it would take three times as long to complete compared to the grayscale version since there are three channels to look at instead of just one. In order to accomplish the grayscale conversion, the following equation is used:

$$Gray = (Red \gg 2) + (Red \gg 5) + (Green \gg 1) + (Green \gg 4) + (Blue \gg 4) + (Blue \gg 5)$$

In the above equation, Red corresponds to the red channel, Green corresponds to the green channel, and Blue corresponds to the blue channel. The function was performed on every pixel in the frame in order to achieve a grayscale image that will be suitable for histogram analysis.

The last step in the color filtration module was to invert the image. During the blob detection step of the algorithm, it is important for there to be a sharp contrast between what could potentially be blobs and what is not. This is to make sure that the algorithm picks up on all of the potential stoplights. Therefore, it was decided that making the background pixels white instead of black would be better for contrast. In order to accomplish this conversion, each pixel in the image was looked at to determine if it was black or not. If the pixel was black, it was changed to white. Otherwise, it was left as the grayscale value that it was at.

3.5. Handoff to Processor

The video content is now fully pre-processed on the FPGA and is ready to be passed to the microprocessor for more complicated algorithms such as blob detection and histogram analysis. For this task, the next IP core in the design is the VDMA module. This block acts as a video buffer and sends the video content to external memory. This is important because it allows the input video and output video to run at different clock cycles and allows the microprocessor to access the video content after all of the color filtration has been done (Avnet Electronics, 2013).

4. Software Design

This section describes the software developed to run on the SoC. The C code used in the final design was developed using the Matlab algorithms as a basis. The software components of the system consisted of blob detection and the histogram analysis.

4.1. Blob Detection Algorithm

The blob detection algorithm used in the Matlab simulations could not be directly ported over to the SoC, so an implementation in C had to be designed. There are numerous ways to perform blob detection, however many of these require passing through the whole image multiple times. Since the SoC computes significantly slower than the FPGA it was already the system bottleneck. Due to this, the algorithm used needed to be as computationally simple as possible.

The code used in the final design was based on searching for areas of white surrounded by solid grayscale areas, shown in Figure 22 by examining transitions between the white and nonwhite objects. Each frame was processed pixel by pixel, starting in the top left corner moving to the right edge before going to the next consecutive row. Three flags were used in the implementation. One was used to make the first entry into a grey region, *entry_flag*, a second was used to mark the second entry into a grey region, *reentry_flag*, and lastly a flag to mark that the first entered gray region has been exited, *exit_flag*.

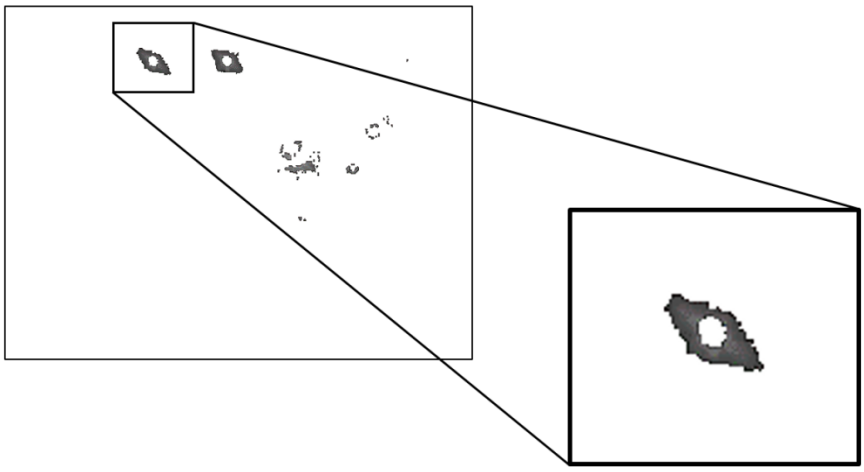


Figure 22: Example Blob to Detect

The algorithm was structured as a finite state machine, or FSM, with four states. The states were named based upon which transition they would represent: *white_to_white*, *white_to_grey*, *grey_to_grey*, and *grey_to_white*. A broad overview of the FSM and its transitions can be seen in Figure 22.

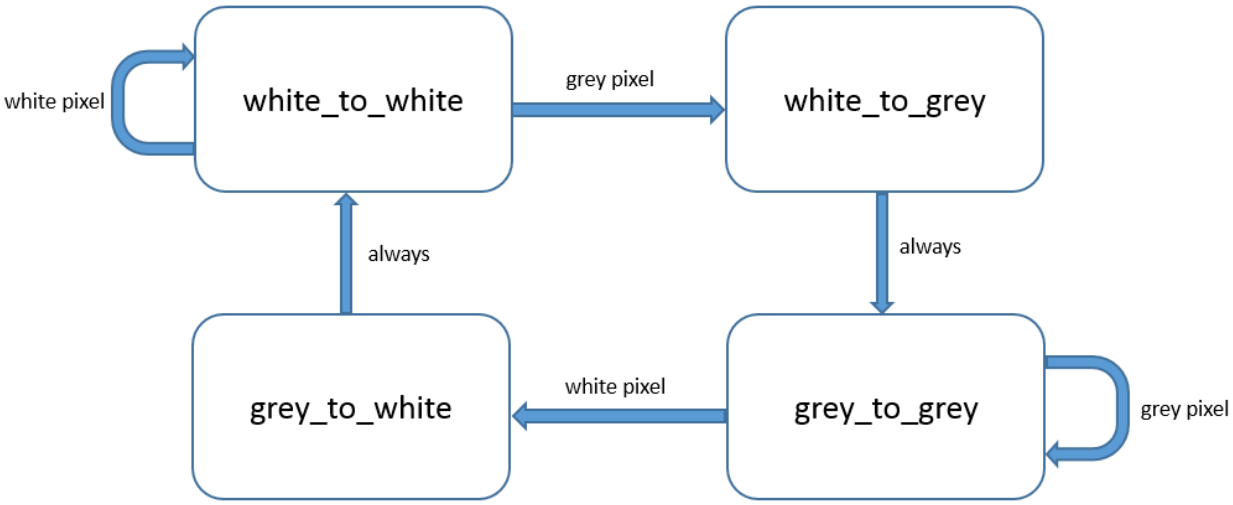


Figure 23: Finite State Machine Transitions

The default state was *white_to_white*. When a nonwhite pixel was found the FSM would transition to the *white_to_grey* state. Additionally, if the *entry_flag* was set, meaning that the current pixel was possibly in a white region surrounded by a grayscale region, the distance between the last gray to white transition and the current pixel were compared. If the distance between the last transition and the current point was too large, the *entry_flag* was unset. The purpose of this was to reset the FSM when it was believed that the current white area was not surrounded by gray.

In the *grey_to_grey* state each pixel is continued to be read until a white pixel is encountered. When this happens the FSM transitions into the *grey_to_white* state.

In the *grey_to_white* state the FSM checks if the *entry_flag* was set. If the *entry_flag* is set and the *reentry_flag* is not set the *exit_flag* is set, signifying that the next pixel is possibly surrounded by a

region of grayscale. The FSM transitions when the next pixel is read to the *grey_to_white* or *grey_to_grey* state, based on the matching pixel value.

The final state in the FSM was the *white_to_grey* state. The first time this state is entered the *entry_flag* is set to mark that entry into a blob has occurred. The next time this state is entered, if the *entry_flag* is still set, then it is reset and the *reentry_flag* is set. The FSM transitions to the *grey_to_white* or *grey_to_grey* state if the next pixel read is white or nonwhite respectively. If this state is entered and the *reentry_flag* is set the system first assumes that a blob was found. To verify this, the distance between the current white to grey, and previous grey to white transition is calculated. If this value is within an expected value, the horizontal center point is calculated. From this horizontal center point, the algorithm would check above and below to determine if the region was surrounded on all four sides by grayscale values. This was performed by checking the pixels in the frame buffer in the same column, but adjacent rows. If there were upper and lower boundaries within an accepted range the center point was marked by changing the pixel color.

4.1.1. Blob Detection Example

This section walks through the process of detecting a blob in the frame shown in Figure 24.

Figure 24: Blob Detection Example Frame

The system would step through pixel by pixel, remaining in the *white_to_white* state until it hit the first grayscale values, which in this frame would be the top few pixels of the first stoplight. During this transition, the *entry_flag* would be set. The system would stay in the *grey_to_grey* state for a few cycles as illustrated in Figure 25. Upon leaving the gray region, the *exit_flag* would be set as well as a value, *leftside_x*, which is used to remember when the region was exited. In the *white_to_white* state, after 100 white pixels, the *entry_flag*, *exit_flag*, and *leftside_x* are all reset since this is too far away from the grayscale region to be a stoplight. This same procedure would repeat for the next few rows of processing the frame.

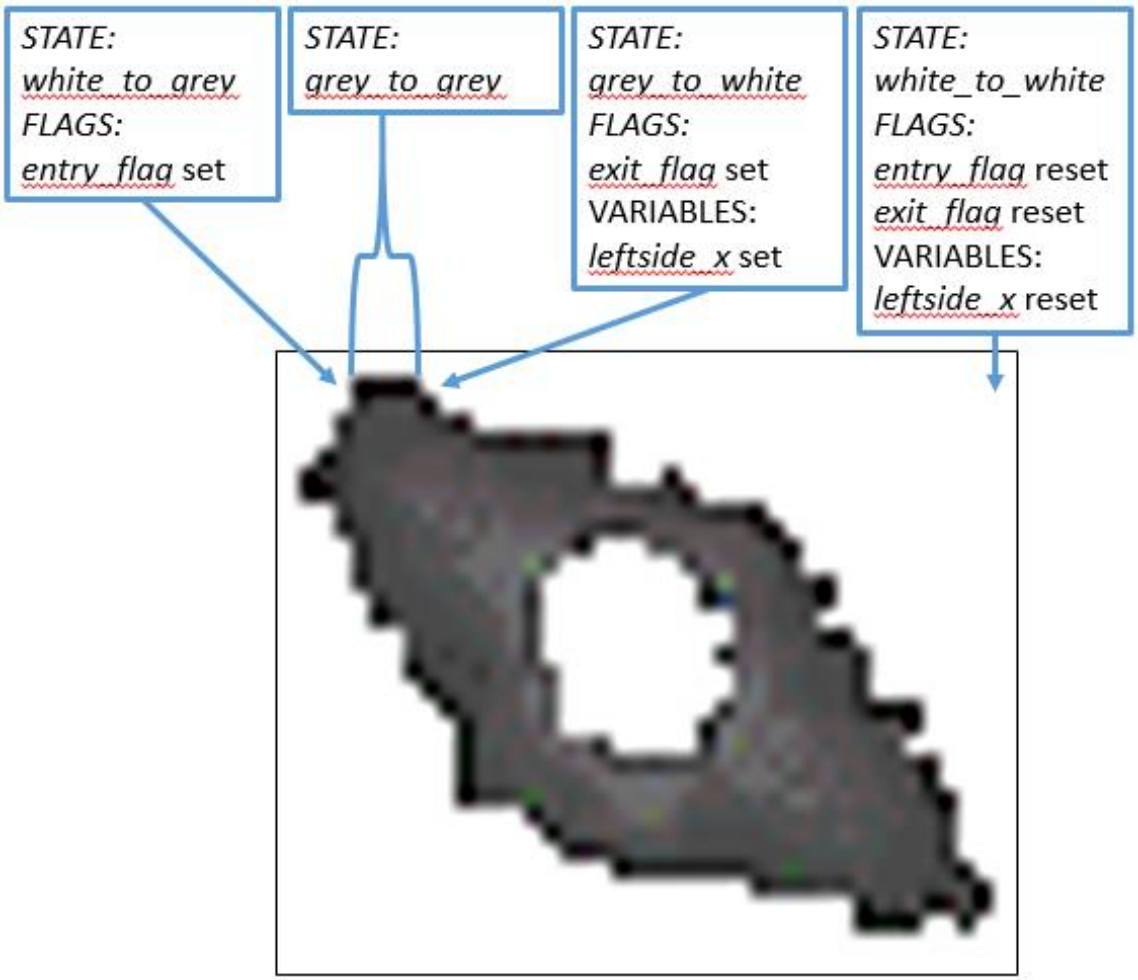


Figure 25: Processing of Grayscale Values Not Surrounded by A White Region

The first time a different path in the algorithm is taken, occurs at the point illustrated in Figure26. In this situation, a white region is found which is bounded by grayscale values to both the left and right hand side. The next step in the algorithm was to determine the center point. This was calculated by simply subtracting the *rightside_x* by the *leftside_x* value and dividing by two. This would give the distance, in pixels, to the centerpoint from the current point.

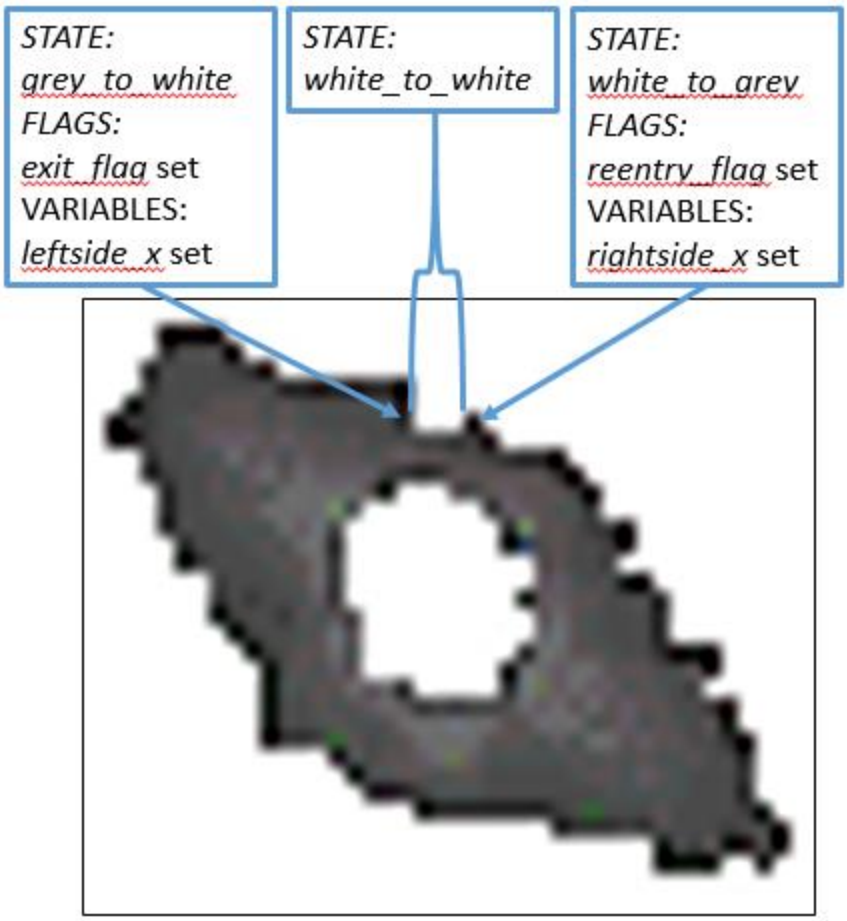


Figure 26: Processing of White Region Not Fully Surrounded by Grayscale, Horizontal

From this horizontal center point, the algorithm would check for vertical bounding by grayscale, as shown in Figure27. In this case, a lower bounding are is found, and a variable, *down_counter*, is set to remember this location. Additionally in this case, an upper bound is not found. After 100 pixels of searching upwards, it is determined that this region is not bounded by grayscale values, and therefore is not a stoplight.

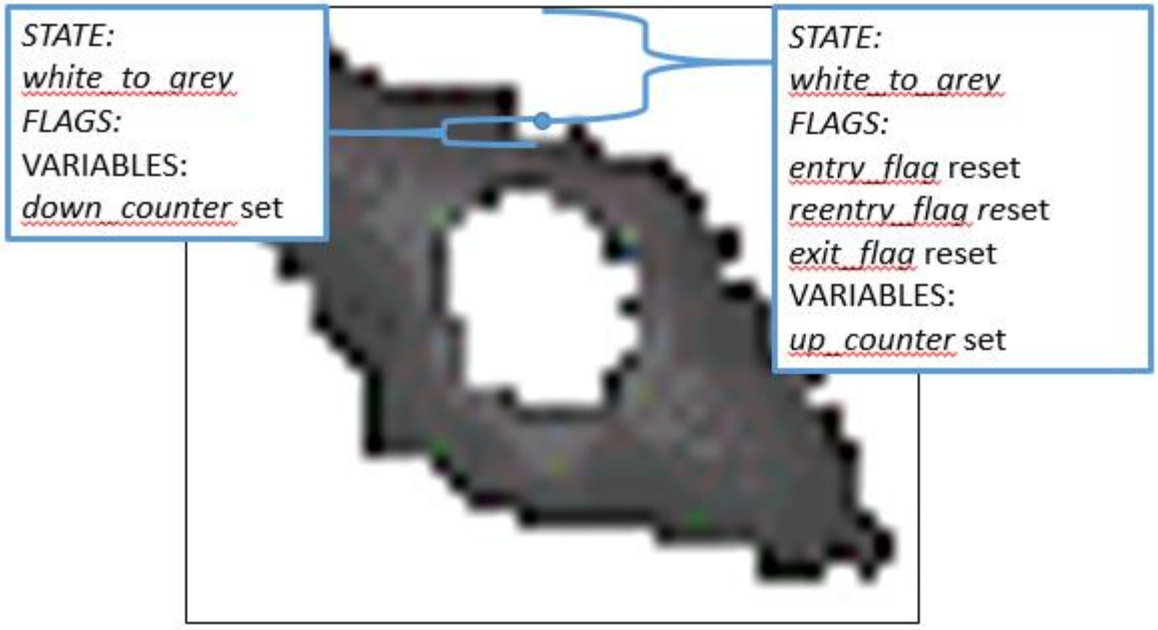


Figure 27: Processing of White Region Not Fully Surrounded by Grayscale, Vertical

The final possibility in processing a blob is detecting a stoplight. Similar to the previous example, the system searches for a region where it can find a horizontally bounded region of white. In the event that a vertical center point is found, the vertical center point would be marked by changing the cbcr value, to a non-grayscale value. The cbcr value can be changed without affecting the rest of the image since only the luma value is used for the transition detection. This is because the grayscale representation is only shown in luma values, with the cbcr value constant since the color does not change. As the system continues processing, if a horizontal center point is found later on, where the cbcr value was changed to a non-grayscale value, it is declared to be the centroid of a blob, and the histogram algorithm is performed from this centroid.

4.2. Histogram Decision Algorithm

Once a center point was found, the region was considered to be a blob, and thus possibly a stoplight. The final part of the algorithm was the histogram analysis to determine whether or not the detected blobs were stoplights. First, a box was created with a common centroid to the blob that had

been found. Next, each pixel in the box was examined and classified into one of the bins. When all pixels had been surveyed each bin was compared to a set of requirements that had been experimentally found to characterize stoplights. By increasing the bins, the accuracy was significantly increased from the initial Matlab simulations. The final requirements are provided in Table 4-1.

Table 4-1: Red Light Histogram Identification

Region	Grayscale Range	Stoplight Requirement
Hist0	0-25	<5
Hist1	26-50	<5
Hist2	51-75	<35
Hist3	76-100	<200
Hist4	101-125	>25
Hist5	126-150	>20
Hist6	151-175	<30
Hist7	176-200	<5
Hist8	201-225	<5
Hist9	226-250	<5

5. System Output

After the processor is completed doing blob detection and histogram analysis, the system knows whether or not there is a stoplight in view, but the user does not. Therefore, the user must be alerted in some way as to whether there is a stoplight or not in front of them. Two different methods of output

were suggested; video output through the HDMI output module and peripheral output that includes an audio buzzer and warning lights.

5.1. Video Output

The first method of user output that was explored was using video output. This would utilize the HDMI out module from the HDMI pass through tutorial to show where stoplights were on a monitor. This was done using a few IP cores, which can be seen in the block diagram in Figure 20. The first IP core was the RGB to YCbCr module. Much like the YCbCr to RGB module that was used on the incoming video content, this module is used to convert the format of the image. At this point, after the image has been processed by the ARM processor, its format is in RGB. However, in order to output the image through an HDMI output module, the format needs to be in YCbCr. Luckily, this conversion is rather simple and involves matrix multiplication much akin to the conversion from RGB to YCbCr. The formula can be seen in Figure 28.

$$\begin{bmatrix} Y \\ Cb \\ Cr \end{bmatrix} = \begin{bmatrix} 0 \\ 128 \\ 128 \end{bmatrix} + \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.169 & -0.331 & 0.500 \\ 0.500 & -0.419 & -0.081 \end{bmatrix} * \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

Figure 28: RGB to YCbCr Conversion

At this point, the content gets passed to the AXI4S2 and HDMI output module. These are the same as the modules used in the pass through tutorial. An HDMI monitor can then be connected to the HDMI out port and the filtered image with stoplights highlighted can be seen, as illustrated in Figure 30.

5.2. Peripheral Output

Along with video output, it was also decided that peripheral output would be useful. The video output is a great asset when debugging the system; however in actual application it would be less than ideal. Every car does not come equipped with an HDMI monitor to plug into and even if they do, a

driver should not have to stare at a monitor to know where a stoplight is. The video output also shows the exact position of stoplights, but a driver only worries about whether they are there, not their exact location in front of them. As such, two types of peripheral output were suggested; a visual indication and an audio indication.

A visual indication is useful in a system such as this because it acts as a binary on or off; the system either sees a stoplight or it does not. This can be accomplished with a simple LED and one of the GPIO pins on the Xilinx ZC702 board. When the system sees a stoplight, it sets the pin to high and the LED turns on. Otherwise, the pin is set to low and the LED turns off. The LED could be set in the dashboard of a vehicle so that it is in the driver's field of view at all times and be a useful indicator. However, if the driver is drowsy or not paying attention, a visual indicator may not be enough to alert them to the presence of a stoplight. In this case, a different kind of peripheral device could be used; an audio device.

An audio device, such as a buzzer, would work much in the same way as the LED. It would be attached to one of the Xilinx ZC702 board's GPIO pins and set to high when the system sees a stoplight. However, this method would be much more useful to a driver who is not paying attention. A loud noise is much more likely to get someone's attention than a light turning on. Therefore, an audio buzzer is the preferred peripheral device to use with this system in order to alert a driver to an incoming stoplight.

5.2.1 Circuit Description

In order to test the peripheral devices mentioned above, a test circuit was developed which can be seen in Figure 29. The GPIO pin from the board is attached to V_{SIG} and V^+ is a 5V supply from the board. As can be seen, the signal voltage attaches to an analog switch that, when high, sends 5V to the system. This voltage is used to power the visual aid (LED) and also a 555 timer. This 555 timer then supplies an output voltage, V_{OUT}. This output voltage is attached to an audio peripheral, the buzzer.

There is a potentiometer at R6 as well which functions as a volume control. This allows the user to turn the volume up or down depending on the situation. The volume control is also useful for debugging purposes.

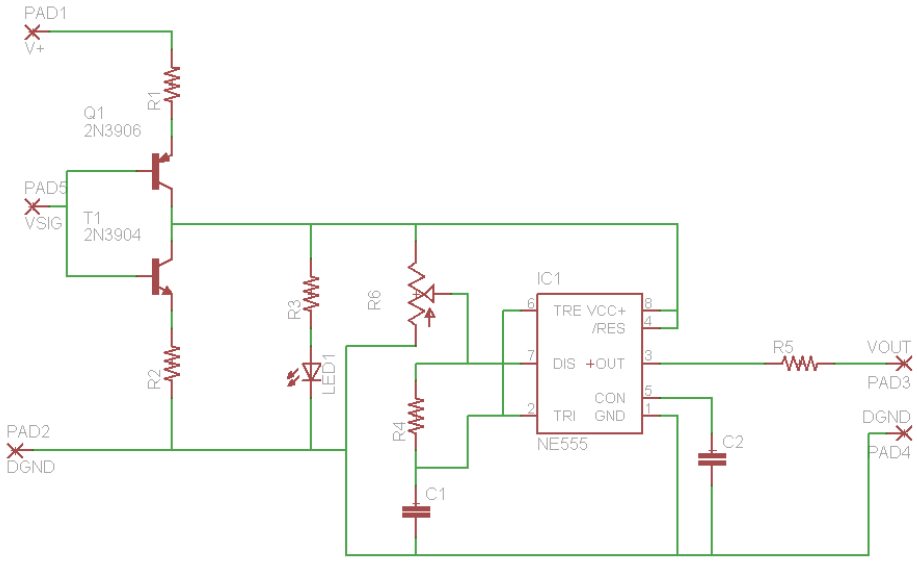


Figure 29: Peripheral Schematic

6. Closing Remarks

6.1. Final Results

The final system was able to successfully identify stoplights in 13 minutes and 12 seconds of test video collected from driving the streets of Worcester, Massachusetts. All test video was recorded with a dashboard mounted Samsung Galaxy S2 camera. Additionally, test video was only collected in a single vehicle, a 2003 Honda Accord. The results for different recording sources and vehicles are unknown.

Figure 30: Final Results Debug Screen Capture

6.2. Future Work

6.2.1. FPGA Only Implementation

One desired change to the system would be to port over the C code to VHDL or Verilog so that the entire system could run on the FPGA. The purpose of this would be to greatly increase the speed in which the system processes the frames. In the Matlab simulations, it was determined that blob detection took the longest time, roughly 67% of the total execution time per frame, as illustrated in Figure 31. Since blob detection was performed on the SoC in the final implementation, it was clear that this was the system's processing bottleneck. While the implementation was still able to alert the driver of stoplights faster than the average human reaction time, a decrease in blob detection execution time, would allow an increased amount of time for a more accurate and computationally intensive stoplight identification algorithm.

Function Name	Calls	**Total Time**	Self Time*	Total Time Plot (dark band = self time)
colorseparation640exetime	1	0.141 s	0.019 s	
regionprops	1	0.095 s	0.004 s	

Figure 31: Matlab Simulation Single Frame Execution Profile

6.2.2. Support Vector Machine

Another area to be researched would be to use a support vector machine (SVM) for the classification of stoplights. A SVM is based upon the concept of decision planes in order to separate objects that have different class memberships. An SVM is "trained" by providing it with many examples of different objects that fall into different categories. After this, when a SVM is provided with a new case of an object, it makes a calculated guess based upon its training as to what to classify the new object as (StatSoft Inc., 2015). In this case, the object classification would fall into two categories; a stoplight and a non-stoplight. The SVM could be trained by providing it examples of images that are stoplights and could then be implemented on the device. This would provide much more accurate results than the histogram analysis that was done in this project.

6.2.3. Daytime Functionality

Another area to investigate for future work is daytime stoplight detection. While nighttime detection was deemed more important due to drowsy or distracted drivers, there are still a large amount of daytime accidents caused by drivers missing stoplights. It was found that the algorithm used was not acceptable in daylight conditions. One approach to solve this problem would be to add a daylight sensor to the system, and implement a different algorithm for day and nighttime. Another approach would be to develop a completely new algorithm that would successfully identify stoplights in both light and dark conditions. Since many automobiles already have daylight sensing technology, currently used for automatically turning on headlights when it is dark, the former approach would most likely be the best solution.

Figure 32: Daylight Color Filter Testing

6.2.4. Interfacing with Other Driving Assistance Technologies

The final desired future work for this project is implementation with other driver assistance technologies. One technology that could be used to increase the accuracy of the stoplight detection is lane detection. The current implementation is unable to differentiate between stoplights in different lanes. Due to this, for multilane roads, false positives are possible since different lanes may have different signals active at any given time. By combining lane detection and stoplight detection, it could be possible to limit detection only to the appropriate lane stoplight.

Another technology that would work well with stoplight detection is range finding. Currently, 24GHz and 77GHz radars are used in driver assistance systems. The 24GHz systems are used for close range detection including parking aides and blind spot detection. The 77GHz systems are used for long

range detection purposes including adaptive cruise control and assisted braking. By combining this technology with stoplight detection, the vehicle could have the capability of braking for a stoplight if the driver fails to do so within safe stopping distances based on the speed they were traveling.

An additional automotive technology to integrate with would be automatic braking systems. Currently, these systems are used for collision avoidance and adaptive cruise control. Current technology has the ability to scan for large incoming objects with radar, laser, or visual technologies, and begin to brake without input from the driver (About.com, 2015). The addition of a stoplight detection system would allow for preventing automobiles from running through stoplights instead of solely large objects it can track.

References

About.com. (2015). *What Is An Automatic Braking System?* Retrieved March 24, 2015, from About Autos: http://cartech.about.com/od/Safety/a/What-Is-An-Automatic-Braking-System.htm

Altera. (2013). *DE1-SoC User Manual.* Retrieved November 12, 2014, from Altera Web Site: ftp://ftp.altera.com/up/pub/Altera_Material/13.1/Boards/DE1-SoC/DE1_SoC_User_Manual.pdf

Altera. (2014, December). *Quartus II Design Software.* Retrieved February 12, 2015, from Altera Literature: http://www.altera.com/literature/br/br-quartus2-software.pdf

Altera. (2015). *FPGAs*. Retrieved February 8, 2015, from FPGAs: http://www.altera.com/products/fpga.html

Avent Electronics. (2015). *HDMI Input/Output FMC Module*. Retrieved December 15, 2014, from Avnet Electronics Marketing: http://www.em.avnet.com/en-us/design/drc/Pages/HDMI-Input-Output-FMC-module.aspx

Avnet Electronics. (2013, January 3). *FMC-Imageon Building a Video Design from Scratch Tutorial.* Retrieved December 12, 2014, from Xilinx Application Notes, Reference Designs, Video IP and Development Kits: https://source.ece.iastate.edu/scm/viewvc.php/*checkout*/MP-2/docs/Camera/FMC-IMAGEON-Tutorial.pdf?root=cpre488

Human Benchmark. (2015). *Human Benchmark - Reaction Time Statistics*. Retrieved February 5, 2015, from Human Benchmark Reaction Time: http://www.humanbenchmark.com/tests/reactiontime/statistics

Jri Lee, Y.-A. L.-H.-J. (2010). A Fully-Integrated 770GHz FMCW Radar Transciever in 65-nm CMOS Technology. *IEEE Journal of Solid-State Circuits, Vol. 45, No. 12*, 11.

Mathworks, Inc. (2014). *Image Processing Toolbox User's Guide.* Retrieved February 18, 2015, from Mathworks Help Website: http://www.mathworks.com/help/pdf_doc/images/images_tb.pdf

National Instruments. (2013, May 29). *Smart Grid Ready Instrumentation.* Retrieved February 22, 2015, from Smart Grid Ready Instrumentation - National Instruments: http://www.ni.com/white-paper/14529/en/

National Instruments. (2013, 5 29). *Smart Grid Ready Instrumentation - National Instruments*. Retrieved 2 9, 2015, from National Instruments Website: http://www.ni.com/white-paper/14529/en/

Phanomeme. (n.d.). Retrieved from http://phaenomeme.sueddeutsche.de/image/110823081234

Sparsh Mittal, S. G. (2008). FPGA: An Efficient and Promising Platform for Real-Time Image Processing Applications. *Proceedings ofthe National Conference on Research and Development in Hardware & Systems*, 4.

StatSoft Inc. (2015). *Support Vector Machines (SVM) Introductory Overview*. Retrieved February 17, 2015, from Support Vector Machines (SVM): http://www.statsoft.com/Textbook/Support-Vector-Machines#overview

Think Silicon. (n.d.). Retrieved from http://www.think-silicon.com/images/grad640_dxt.png

U.S. Federal Highway Administration. (2014, September 4). *Red-Light Running*. Retrieved January 30, 2015, from U.S. Department of Transportation Safety: http://safety.fhwa.dot.gov/intersection/redlight/

Wikipedia. (n.d.). *HSV Cylinder.* Retrieved from Wikipedia: http://en.wikipedia.org/wiki/HSL_and_HSV#/media/File:HSV_color_solid_cylinder_alpha_lowgamma.png

Xilinx Inc. (2014, June 4). *ZC702 Evaluation Board for the Zynq-7000 XC7Z020 All Programmable SoC User Guide.* Retrieved December 3, 2014, from Xilinx ZC702 User Guide: http://www.xilinx.com/support/documentation/boards_and_kits/zc702_zvik/ug850-zc702-eval-bd.pdf

Xilinx Inc. (2015). *SoC*. Retrieved 2 9, 2015, from Xilinx Inc All Programmable SoC: http://www.xilinx.com/products/silicon-devices/soc.html

Xilinx, Inc. (2015). *Xilinx Platform Studio (XPS)*. Retrieved February 12, 2015, from Xilinx Platform Studio: http://www.xilinx.com/tools/xps.htm

Appendix A: Matlab Color Extraction GUI

```
% Color filter GUI
function colorseparationgui(image)
f = figure('Units', 'normalized', 'Position', [1/3, 1/3, 1/3, 1/3]);    %
create GUI figure
set(f, 'Name', 'GUI for Color Filtering');  % set GUI name
set(f, 'NumberTitle', 'off');

% set default values
rlimit = 230;
blimit = 0;
glimit = 0;
rhigh = 255;
ghigh = 240;
bhigh = 240;

colorFilter(image,rlimit,blimit,glimit,rhigh,bhigh,ghigh);  % run color
filter

% add static text fields
BF = uicontrol('Style', 'text', 'Units', 'normalized',...
    'Position', [0.7, 0.4, 0.1, 0.1], 'String', 'Upper Limit',...
    'BackgroundColor', [.8 .8 .8]);
BF = uicontrol('Style', 'text', 'Units', 'normalized',...
    'Position', [0.8, 0.4, 0.1, 0.1], 'String', 'Lower Limit',...
    'BackgroundColor', [.8 .8 .8]);
BF = uicontrol('Style', 'text', 'Units', 'normalized',...
    'Position', [0.5, 0.1, 0.1, 0.1], 'String', 'Red',...
    'BackgroundColor', [.8 .8 .8]);
BF2 = uicontrol('Style', 'text', 'Units', 'normalized',...
    'Position', [0.5, 0.2, 0.1, 0.1], 'String', 'Blue',...
    'BackgroundColor', [.8 .8 .8]);
BF3 = uicontrol('Style', 'text', 'Units', 'normalized',...
    'Position', [0.5, 0.3, 0.1, 0.1], 'String', 'Green',...
    'BackgroundColor', [.8 .8 .8]);

% add editable fields
BF = uicontrol('Style', 'edit', 'Units', 'normalized',...
    'Position', [0.8, 0.1, 0.1, 0.1], 'String', '240', ...
    'BackgroundColor', [.9 .9 .9],...
    'Callback', {@BF_Callback_low});
BF = uicontrol('Style', 'edit', 'Units', 'normalized',...
    'Position', [0.7, 0.1, 0.1, 0.1], 'String', '255', ...
    'BackgroundColor', [.9 .9 .9],...
    'Callback', {@BF_Callback_high});
BF2 = uicontrol('Style', 'edit', 'Units', 'normalized',...
    'Position', [0.8, 0.2, 0.1, 0.1], 'String', '200', ...
    'BackgroundColor', [.9 .9 .9],...
    'Callback', {@BF2_Callback_low});
BF2 = uicontrol('Style', 'edit', 'Units', 'normalized',...
    'Position', [0.7, 0.2, 0.1, 0.1], 'String', '255', ...
    'BackgroundColor', [.9 .9 .9],...
    'Callback', {@BF2_Callback_high});
BF3 = uicontrol('Style', 'edit', 'Units', 'normalized',...
```

```
    'Position', [0.8, 0.3, 0.1, 0.1], 'String', '200', ...
    'BackgroundColor', [.9 .9 .9],...
    'Callback', {@BF3_Callback_low});
BF3 = uicontrol('Style', 'edit', 'Units', 'normalized',...
    'Position', [0.7, 0.3, 0.1, 0.1], 'String', '255', ...
    'BackgroundColor', [.9 .9 .9],...
    'Callback', {@BF3_Callback_high});

% callback function that changes the plot
    function BF_Callback_low(hObject, handles)
        user_entry = str2double(get(hObject, 'string'));
        if isnan(user_entry)
            errordlg('You must enter a numeric value','Bad Input','modal')
            uicontrol(hObject); return;
        end
        % Proceed with callback...
        rlimit = user_entry;
        colorFilter(image,rlimit,blimit,glimit,rhigh,bhigh,ghigh);
    end

    function BF_Callback_high(hObject, handles)
        user_entry = str2double(get(hObject, 'string'));
        if isnan(user_entry)
            errordlg('You must enter a numeric value','Bad Input','modal')
            uicontrol(hObject); return;
        end
        % Proceed with callback...
        rhigh = user_entry;
        colorFilter(image,rlimit,blimit,glimit,rhigh,bhigh,ghigh);
    end

    function BF2_Callback_low(hObject, handles)
        user_entry = str2double(get(hObject, 'string'));
        if isnan(user_entry)
            errordlg('You must enter a numeric value','Bad Input','modal')
            uicontrol(hObject); return;
        end
        % Proceed with callback...
        blimit = user_entry;
        colorFilter(image,rlimit,blimit,glimit,rhigh,bhigh,ghigh);
    end

    function BF2_Callback_high(hObject, handles)
        user_entry = str2double(get(hObject, 'string'));
        if isnan(user_entry)
            errordlg('You must enter a numeric value','Bad Input','modal')
            uicontrol(hObject); return;
        end
        % Proceed with callback...
        bhigh = user_entry;
        colorFilter(image,rlimit,blimit,glimit,rhigh,bhigh,ghigh);
    end

    function BF3_Callback_low(hObject, handles)
        user_entry = str2double(get(hObject, 'string'));
        if isnan(user_entry)
```

```
            errordlg('You must enter a numeric value','Bad Input','modal')
            uicontrol(hObject); return;
        end
        % Proceed with callback...
        glimit = user_entry;
        colorFilter(image,rlimit,blimit,glimit,rhigh,bhigh,ghigh);
    end

    function BF3_Callback_high(hObject, handles)
        user_entry = str2double(get(hObject, 'string'));
        if isnan(user_entry)
            errordlg('You must enter a numeric value','Bad Input','modal')
            uicontrol(hObject); return;
        end
        % Proceed with callback...
        ghigh = user_entry;
        colorFilter(image,rlimit,blimit,glimit,rhigh,bhigh,ghigh);
    end
end
```

Appendix B: Matlab Implementation

```
% Final simulation Matlab code.
% Note: All test code and debugging code has been removed
% Purpose: Processes a single frame for stoplight detection
% Input: RGB frame
% Output: Outputs 1 if stoplight detected, else 0

function out =  colorseparation640exetime(image)
out = 0;    % initialize output to 0, meaning no stoplight detected by
default
redBand = image(:,:,1); % create vector with only the red band of the RGB
greenBand = image(:,:,2);   % create vector with only the green band of the
RGB
blueBand = image(:,:,3);    % create vector with only the blue band of the
RGB

redMask = (redBand > 149);  % create vector for red color filtering
greenMask = (greenBand < 110);  % create vector for green color filtering
blueMask = (blueBand < 110);    % create vector for blue color filtering

redobjectsmask = uint8(redMask & greenMask & blueMask); % create a color
filter mask

maskedrgb = uint8(zeros(size(redobjectsmask))); % initalize an empty vector
maskedrgb(:,:,1) = redBand .* redobjectsmask;   % filter the red component
maskedrgb(:,:,2) = greenBand .* redobjectsmask; % filter the green component
maskedrgb(:,:,3) = blueBand .* redobjectsmask;  % filter the blue component

binaryImage = ~im2bw(maskedrgb, 0.2);   % create binary image for blob
detection

blobMeasurements = regionprops(binaryImage, 'Area','BoundingBox');  % perform
blob detection, calculate area and bounding boxes

numberOfBlobs = size(blobMeasurements, 1);  % calculate the number of blobs
found

for k = 2:1:numberOfBlobs   % for each blob
    if blobMeasurements(k).Area > 10    % if the blob's area is greater than
10 pixels
        thisBlobsBoundingBox = blobMeasurements(k).BoundingBox;  % get the
corners of this blob

        % extend the blob size by 4 in each direction
        thisBlobsBoundingBox(1) = thisBlobsBoundingBox(1) - 4;
        thisBlobsBoundingBox(2) = thisBlobsBoundingBox(2) - 4;
        thisBlobsBoundingBox(3) = thisBlobsBoundingBox(3) + 8;
        thisBlobsBoundingBox(4) = thisBlobsBoundingBox(4) + 8;

        subImage = imcrop(image, thisBlobsBoundingBox); % crop out the blob
region from the original frame

        temp = rgb2gray(subImage);  % convert the cropped image to grayscale
```

```
        hist = imhist(temp);    % create the histogram of the cropped image
        part1 = sum(hist(1:50));    % sum up the values for bin1
        part2 = sum(hist(90:110));  % sum up the values for bin2
        part3 = sum(hist(151:220)); % sum up the values for bin3
        part4 = sum(hist(241:255)); % sum up the values for bin4

        % stoplight identification parameters
        if ((part1 < 5) && (part2 > 100) && (part2 > part4) && (part3 > 0) &&
(part4 > 0) && (part2 > part3))
            out = 1; % stoplight detected, set output
        end % end stoplight identification
    end % end if area too small
end % end this blob testing
end % end function
```

Appendix C: Color Filtration – Verilog

```
module  Color_Filter (
        input                   clk,
        input                   reset,
        input           [7:0]   oVGA_Red,
        input           [7:0]   oVGA_Green,
        input           [7:0]   oVGA_Blue,
        output  reg     [23:0]  filtered_color);

        //temp channels for color extraction
        reg     [7:0]   filtered_Red;
        reg     [7:0]   filtered_Green;
        reg     [7:0]   filtered_Blue;

        //temp channels for grayscale conversion
        reg     [7:0]   greyscale_Red;
        reg     [7:0]   greyscale_Green;
        reg     [7:0]   greyscale_Blue;

        //temp channels for black -> white conversion
        reg     [7:0]   updated_greyscale_Red;
        reg     [7:0]   updated_greyscale_Green;
        reg     [7:0]   updated_greyscale_Blue;

        //values for color extraction
        parameter redFilterValue = 150;
        parameter greenFilterValue = 110;
        parameter blueFilterValue = 110;

        parameter white = 8'b11111111;

        //filter out all color that isn't red
        always @ (posedge clk) begin
                filtered_Red <= (oVGA_Red >= redFilterValue && oVGA_Green <= greenFilterValue &&
                        oVGA_Blue <= blueFilterValue) ? oVGA_Red : 0;
                filtered_Green <= (oVGA_Red >= redFilterValue && oVGA_Green <= greenFilterValue &&
                        oVGA_Blue <= blueFilterValue) ? oVGA_Green : 0;
                filtered_Blue <= (oVGA_Red >= redFilterValue && oVGA_Green <= greenFilterValue &&
                        oVGA_Blue <= blueFilterValue) ? oVGA_Blue : 0;
        end

        //RGB->Greyscale
        always @ (posedge clk) begin
                greyscale_Red <= (filtered_Red>>2) + (filtered_Red>>5) + (filtered_Green>>1) +
                        (filtered_Green>>4) + (filtered_Blue>>4) + (filtered_Blue>>5);
                greyscale_Green <= (filtered_Red>>2) + (filtered_Red>>5) + (filtered_Green>>1) +
                        (filtered_Green>>4) + (filtered_Blue>>4) + (filtered_Blue>>5);
                greyscale_Blue <= (filtered_Red>>2) + (filtered_Red>>5) + (filtered_Green>>1) +
                        (filtered_Green>>4) + (filtered_Blue>>4) + (filtered_Blue>>5);
        end
```

```
        //Changes all greyscale values below 51 to white (black -> whtie)
        always @ (posedge clk) begin
                updated_greyscale_Red = (greyscale_Red < 51) ? white : greyscale_Red;
                updated_greyscale_Green = (greyscale_Green < 51) ? white : greyscale_Green;
                updated_greyscale_Blue = (greyscale_Blue < 51) ? white : greyscale_Blue;
        end

        //appends the three channels together to be the output 24 bit image data
        always @ (posedge clk) begin
                filtered_color <= {updated_greyscale_Red, updated_greyscale_Blue, updated_greyscale_Green};
        end

endmodule
```

Appendix D: Frame Buffer Initialization

```
#include <stdio.h>
#include "platform.h"
#include "fmc_imageon_hdmi_framebuffer.h"


#include "xgpiops.h"
#include "xparameters.h"


fmc_imageon_hdmi_framebuffer_t demo;
void print( const char *ptr);

int main(){

	init_platform();

	//initialize framebuffer
	demo.uBaseAddr_IIC_FmcImageon = XPAR_FMC_IMAGEON_IIC_0_BASEADDR;
	demo.uDeviceId_VTC_HdmiiDetector = XPAR_V_TC_0_DEVICE_ID;
	demo.uDeviceId_VTC_HdmioGenerator = XPAR_V_TC_1_DEVICE_ID;
	demo.uDeviceId_VDMA_HdmiFrameBuffer = XPAR_AXI_VDMA_0_DEVICE_ID;
	demo.uBaseAddr_MEM_HdmiFrameBuffer = XPAR_DDR_MEM_BASEADDR + 0x10000000;
	demo.uNumFrames_HdmiFrameBuffer = XPAR_AXIVDMA_0_NUM_FSTORES;
	fmc_imageon_hdmi_framebuffer_init( &demo );

	cleanup_platform();
	return 0;
}
```

Appendix E: Blob Detection/Histogram Analysis Algorithm

```
#include <stdio.h>
#include "xgpiops.h"
#include "xparameters.h"
#include "fmc_imageon_hdmi_framebuffer.h"

#define offset 0x01000000
#define HISTTEST 1 //undefine to check histogram values in candidate regions
#define REDTEST 1 //define to draw box around stoplights
//#define SINGLELIGHT 1 //define in order to only look for one stoplight instead of all
#define LEDTEST 1 //define to use peripherals

#define PINNUMBER 11 //GPIO pin number for peripherals

#define hist0_v 5 //less than
#define hist1_v 5 //less than
#define hist2_v 35 //5 //less than
#define hist3_v 25 // 40 //greater than
#define hist3_v_high 200 //less than
#define hist4_v 20//100 //greater than
#define hist5_v 30 //less than
#define hist6_v 5 //less than
#define hist7_v 5 //less than
#define hist8_v 5 //less than
#define hist9_v 5 //less than

unsigned char saw_red_light = 0;

#ifdef SINGLELIGHT
unsigned char check_vertical_center_point(unsigned int found_center,  unsigned int extra, Xuint8 *filter,
        unsigned int i, unsigned char cbcr, unsigned char luma, unsigned int x, unsigned int y, XGpioPs my_Gpio);
#endif
#ifndef SINGLELIGHT
void check_vertical_center_point(unsigned int found_center,  unsigned int extra, Xuint8 *filter,
         unsigned int i, unsigned char cbcr, unsigned char luma, unsigned int x, unsigned int y, XGpioPs my_Gpio);
#endif

Xuint8 fmc_imageon_hdmii_edid_content[256] =
{
                0x00, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0x00,
                0x06, 0xD4, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                0x00, 0x16, 0x01, 0x03, 0x81, 0x46, 0x27, 0x78,
                0x0A, 0x32, 0x30, 0xA1, 0x54, 0x52, 0x9E, 0x26,
                0x0A, 0x49, 0x4B, 0xA3, 0x08, 0x00, 0x81, 0xC0,
                0x81, 0x00, 0x81, 0x0F, 0x81, 0x40, 0x81, 0x80,
                0x95, 0x00, 0xB3, 0x00, 0x01, 0x01, 0x02, 0x3A,
                0x80, 0x18, 0x71, 0x38, 0x2D, 0x40, 0x58, 0x2C,
                0x45, 0x00, 0xC4, 0x8E, 0x21, 0x00, 0x00, 0x1E,
                0xA9, 0x1A, 0x00, 0xA0, 0x50, 0x00, 0x16, 0x30,
                0x30, 0x20, 0x37, 0x00, 0xC4, 0x8E, 0x21, 0x00,
                0x00, 0x1A, 0x00, 0x00, 0x00, 0xFC, 0x00, 0x46,
```

```
                    0x4D, 0x43, 0x2D, 0x49, 0x4D, 0x41, 0x47, 0x45,
                    0x4F, 0x4E, 0x0A, 0x20, 0x00, 0x00, 0x00, 0xFD,
                    0x00, 0x38, 0x4B, 0x20, 0x44, 0x11, 0x00, 0x0A,
                    0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x01, 0x54,
                    0x02, 0x03, 0x1F, 0x71, 0x4B, 0x90, 0x03, 0x04,
                    0x05, 0x12, 0x13, 0x14, 0x1F, 0x20, 0x07, 0x16,
                    0x26, 0x15, 0x07, 0x50, 0x09, 0x07, 0x01, 0x67,
                    0x03, 0x0C, 0x00, 0x10, 0x00, 0x00, 0x1E, 0x01,
                    0x1D, 0x00, 0x72, 0x51, 0xD0, 0x1E, 0x20, 0x6E,
                    0x28, 0x55, 0x00, 0xC4, 0x8E, 0x21, 0x00, 0x00,
                    0x1E, 0x01, 0x1D, 0x80, 0x18, 0x71, 0x1C, 0x16,
                    0x20, 0x58, 0x2C, 0x25, 0x00, 0xC4, 0x8E, 0x21,
                    0x00, 0x00, 0x9E, 0x8C, 0x0A, 0xD0, 0x8A, 0x20,
                    0xE0, 0x2D, 0x10, 0x10, 0x3E, 0x96, 0x00, 0xC4,
                    0x8E, 0x21, 0x00, 0x00, 0x18, 0x01, 0x1D, 0x80,
                    0x3E, 0x73, 0x38, 0x2D, 0x40, 0x7E, 0x2C, 0x45,
                    0x80, 0xC4, 0x8E, 0x21, 0x00, 0x00, 0x1E, 0x1A,
                    0x36, 0x80, 0xA0, 0x70, 0x38, 0x1F, 0x40, 0x30,
                    0x20, 0x25, 0x00, 0xC4, 0x8E, 0x21, 0x00, 0x00,
                    0x1A, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01
};

int fmc_imageon_hdmi_framebuffer_init( fmc_imageon_hdmi_framebuffer_t *pDemo )
{
  int ret;
  Xuint32 timeout = 100;
  Xuint32 iterations = 0;

  xil_printf("\n\r");
  xil_printf("------------------------------------------------------\n\r");
  xil_printf("--      FMC-IMAGEON HDMI Video Frame Buffer       --\n\r");
  xil_printf("------------------------------------------------------\n\r");
  xil_printf("\n\r");

  xil_printf( "FMC-IMAGEON Initialization ...\n\r" );

  ret = fmc_iic_xps_init(&(pDemo->fmc_imageon_iic),"FMC-IMAGEON I2C Controller",
          pDemo->uBaseAddr_IIC_FmcImageon );
  if ( !ret )
  {
    xil_printf( "ERROR : Failed to open FMC-IIC driver\n\r" );
    exit(0);
  }

  fmc_imageon_init(&(pDemo->fmc_imageon), "FMC-IMAGEON", &(pDemo->fmc_imageon_iic));
  pDemo->fmc_imageon.bVerbose = pDemo->bVerbose;

  // Configure Video Clock Synthesizer
  fmc_imageon_vclk_init( &(pDemo->fmc_imageon) );
  fmc_imageon_vclk_config( &(pDemo->fmc_imageon), FMC_IMAGEON_VCLK_FREQ_148_500_000);

  // Initialize HDMI Input (including EDID content)
  xil_printf( "HDMI Input Initialization ...\n\r" );
```

```
  ret = fmc_imageon_hdmii_init( &(pDemo->fmc_imageon),
                        1, // hdmiiEnable = 1
                        1, // editInit = 1
                        fmc_imageon_hdmii_edid_content
                        );
  if ( !ret )
  {
    xil_printf( "ERROR : Failed to init HDMI Input Interface\n\r" );
    exit(0);
  }

  // Configure Video Clock Synthesizer
  xil_printf( "Video Clock Synthesizer Configuration ...\n\r" );
  fmc_imageon_vclk_config( &(pDemo->fmc_imageon), FMC_IMAGEON_VCLK_FREQ_148_500_000);
  sleep(1);

#if 0
  xil_printf( "Enabling spread-spectrum clocking (SSC)\n\r" );
  xil_printf( "\ttype=down-spread, amount=-0.75%%\n\r" );
  {
              Xuint8 num_bytes;
              int i;
              Xuint8 iic_cdce913_ssc_on[3][2]=
              {
                0x10, 0x6D, // SSC = 011 (0.75%)
                0x11, 0xB6, //
                0x12, 0xDB  //
              };
     fmc_imageon_iic_mux( &(pDemo->fmc_imageon), FMC_IMAGEON_I2C_SELECT_VID_CLK );
     for ( i = 0; i < 3; i++ )
     {
        num_bytes = pDemo->fmc_imageon.pIIC->fpIicWrite( pDemo->fmc_imageon.pIIC,
           FMC_IMAGEON_VID_CLK_ADDR, (0x80 | iic_cdce913_ssc_on[i][0]), &(iic_cdce913_ssc_on[i][1]), 1);
     }
  }
#endif

  // Set HDMI output to 1080P60 resolution
  pDemo->hdmio_resolution = VIDEO_RESOLUTION_1080P;
  pDemo->hdmio_width  = 1920;
  pDemo->hdmio_height = 1080;
/*
 *
  { "720P",   720,   5,   5,  20,   1, 1280,  110,  40,  220,   1 }, // VIDEO_RESOLUTION_720P
  { "1080P", 1080,   4,   5,  36,   1, 1920,   88,  44,  148,   1 }, // VIDEO_RESOLUTION_1080P
 * */



  //pDemo->hdmio_timing.IsHDMI       = 1; // HDMI Mode
  pDemo->hdmio_timing.IsHDMI       = 0; // DVI Mode
  pDemo->hdmio_timing.IsEncrypted  = 0;
  pDemo->hdmio_timing.IsInterlaced = 0;
```

```
pDemo->hdmio_timing.ColorDepth    = 8;

pDemo->hdmio_timing.HActiveVideo  = 1920;
pDemo->hdmio_timing.HFrontPorch   =   88;
pDemo->hdmio_timing.HSyncWidth    =   44;
pDemo->hdmio_timing.HSyncPolarity =    1;
pDemo->hdmio_timing.HBackPorch    =  148;/*
pDemo->hdmio_timing.HFrontPorch   =   110;
pDemo->hdmio_timing.HSyncWidth    =   40;
pDemo->hdmio_timing.HSyncPolarity =    1;
pDemo->hdmio_timing.HBackPorch    =  220;*/

pDemo->hdmio_timing.VActiveVideo  = 1080;/*
pDemo->hdmio_timing.VFrontPorch   =    5;
pDemo->hdmio_timing.VSyncWidth    =    5;
pDemo->hdmio_timing.VSyncPolarity =    1;
pDemo->hdmio_timing.VBackPorch    =   20;*/
pDemo->hdmio_timing.VFrontPorch   =    4;
pDemo->hdmio_timing.VSyncWidth    =    5;
pDemo->hdmio_timing.VSyncPolarity =    1;
pDemo->hdmio_timing.VBackPorch    =   36;

xil_printf( "HDMI Output Initialization ...\n\r" );
ret = fmc_imageon_hdmio_init( &(pDemo->fmc_imageon),
                              1,                 // hdmioEnable = 1
                              &(pDemo->hdmio_timing), // pTiming
                              0                  // waitHPD = 0
                              );
if ( !ret )
{
  xil_printf( "ERROR : Failed to init HDMI Output Interface\n\r" );
  //exit(0);
}

// Clear frame stores
Xuint32 i;
Xuint32 storage_size = pDemo->uNumFrames_HdmiFrameBuffer * ((1920*1080)<<1);
volatile Xuint8 *pStorageMem = (Xuint8 *)pDemo->uBaseAddr_MEM_HdmiFrameBuffer;
for ( i = 0; i < storage_size; i += 2 )
{
  // Black Pixel
  *pStorageMem++ = 0x80; // CbCr (chroma)
  *pStorageMem++ = 0x00; // Y (luma)
}


volatile Xuint8 *filter = (Xuint8 *)(pDemo->uBaseAddr_MEM_HdmiFrameBuffer+offset);
  for ( i = 0; i < storage_size; i += 2 )
  {
    // Black Pixel
    *filter++ = 0x80; // CbCr (chroma)
    *filter++ = 0x00; // Y (luma)
  }
```

```
// Initialize Output Side of AXI VDMA
xil_printf( "Video DMA (Output Side) Initialization ...\n\r" );
vfb_common_init(
  pDemo->uDeviceId_VDMA_HdmiFrameBuffer, // uDeviceId
  &(pDemo->vdma_hdmi)                 // pAxiVdma
  );
vfb_tx_init(
  &(pDemo->vdma_hdmi),                // pAxiVdma
  &(pDemo->vdmacfg_hdmi_read),        // pReadCfg
  pDemo->hdmio_resolution,            // uVideoResolution
  pDemo->hdmio_resolution,            // uStorageResolution
  (pDemo->uBaseAddr_MEM_HdmiFrameBuffer+offset),  // uMemAddr
  pDemo->uNumFrames_HdmiFrameBuffer     // uNumFrames
  );

// Configure VTC on output data path
xil_printf( "Video Timing Controller (generator) Initialization ...\n\r" );
vgen_init( &(pDemo->vtc_hdmio_generator), pDemo->uDeviceId_VTC_HdmioGenerator );
vgen_config( &(pDemo->vtc_hdmio_generator), pDemo->hdmio_resolution, 1 );

while (1)
{
 if ( iterations > 0 )
 {
  xil_printf( "\n\rPress ENTER to re-start ...\n\r" );
  getchar();
 }
 iterations++;

 xil_printf( "Waiting for ADV7611 to locked on incoming video ...\n\r" );
 pDemo->hdmii_locked = 0;
 timeout = 100;
 while ( !(pDemo->hdmii_locked) && timeout-- )
 {
  usleep(100000); // wait 100msec ...
  pDemo->hdmii_locked = fmc_imageon_hdmii_get_lock( &(pDemo->fmc_imageon) );
 }
 if ( !(pDemo->hdmii_locked) )
 {
  xil_printf( "\tERROR : ADV7611 has NOT locked on incoming video, aborting !\n\r" );
  //return -1;
  continue;
 }
 xil_printf( "\tADV7611 Video Input LOCKED\n\r" );
 usleep(100000); // wait 100msec for timing to stabilize

 // Get Video Input information
 fmc_imageon_hdmii_get_timing( &(pDemo->fmc_imageon), &(pDemo->hdmii_timing) );
 pDemo->hdmii_width  = pDemo->hdmii_timing.HActiveVideo;
 pDemo->hdmii_height = pDemo->hdmii_timing.VActiveVideo;
 pDemo->hdmii_resolution = vres_detect( pDemo->hdmii_width, pDemo->hdmii_height );
```

```
xil_printf( "ADV7611 Video Input Information\n\r" );
 xil_printf( "\tVideo Input      = %s", pDemo->hdmii_timing.IsHDMI ? "HDMI" : "DVI" );
xil_printf( "%s", pDemo->hdmii_timing.IsEncrypted ? ", HDCP Encrypted" : "" );
xil_printf( ", %s\n\r", pDemo->hdmii_timing.IsInterlaced ? "Interlaced" : "Progressive" );
xil_printf( "\tColor Depth      = %d bits per channel\n\r", pDemo->hdmii_timing.ColorDepth );
xil_printf( "\tHSYNC Timing     = hav=%04d, hfp=%02d, hsw=%02d(hsp=%d), hbp=%03d\n\r",
  pDemo->hdmii_timing.HActiveVideo,
  pDemo->hdmii_timing.HFrontPorch,
  pDemo->hdmii_timing.HSyncWidth, pDemo->hdmii_timing.HSyncPolarity,
  pDemo->hdmii_timing.HBackPorch
  );
xil_printf( "\tVSYNC Timing     = vav=%04d, vfp=%02d, vsw=%02d(vsp=%d), vbp=%03d\n\r",
  pDemo->hdmii_timing.VActiveVideo,
  pDemo->hdmii_timing.VFrontPorch,
  pDemo->hdmii_timing.VSyncWidth, pDemo->hdmii_timing.VSyncPolarity,
  pDemo->hdmii_timing.VBackPorch
  );
xil_printf( "\tVideo Dimensions = %d x %d\n\r", pDemo->hdmii_width, pDemo->hdmii_height );

if ( (pDemo->hdmii_resolution) == -1 )
{
  xil_printf( "\tERROR : Invalid resolution, aborting !\n\r" );
  //return -1;
  continue;
}

// Reset VTC on input data path
vdet_init( &(pDemo->vtc_hdmii_detector), pDemo->uDeviceId_VTC_HdmiiDetector );
vdet_reset( &(pDemo->vtc_hdmii_detector) );

xil_printf( "Video DMA (Input Side) Initialization ...\n\r" );

// Stop Input Side of AXI_VDMA (from previous iteration)
vfb_rx_stop(
  &(pDemo->vdma_hdmi)                 // pAxiVdma
  );

// Clear frame stores
Xuint32 i;
Xuint32 storage_size = pDemo->uNumFrames_HdmiFrameBuffer * ((1920*1080)<<1);
volatile Xuint8 *pStorageMem = (Xuint8 *)pDemo->uBaseAddr_MEM_HdmiFrameBuffer;
for ( i = 0; i < storage_size; i += 2 )
{
  // Black Pixel
  *(pStorageMem+2*i+1) = 0x80; // CbCr (chroma)
  *(pStorageMem+2*i+2) = 0x00; // Y (luma)
}

volatile Xuint8 *filter = (Xuint8 *)(pDemo->uBaseAddr_MEM_HdmiFrameBuffer+offset);
  for ( i = 0; i < storage_size; i += 2 )
  {
    // Black Pixel
    *(filter+2*i+1) = 0x80; // CbCr (chroma)
```

```
      *(filter+2*i+2) = 0x00; // Y (luma)
    }



  // Initialize Input Side of AXI VDMA
  vfb_rx_init(
    &(pDemo->vdma_hdmi),               // pAxiVdma
    &(pDemo->vdmacfg_hdmi_write),      // pWriteCfg
    pDemo->hdmii_resolution,           // uVideoResolution
    pDemo->hdmio_resolution,           // uStorageResolution
    pDemo->uBaseAddr_MEM_HdmiFrameBuffer,  // uMemAddr
    pDemo->uNumFrames_HdmiFrameBuffer      // uNumFrames
    );

  xil_printf( "HDMI Output Re-Initialization ...\n\r" );
  ret = fmc_imageon_hdmio_init( &(pDemo->fmc_imageon),
                                1,               // hdmioEnable = 1
                                &(pDemo->hdmio_timing), // pTiming
                                0                // waitHPD = 0
                                );
  if ( !ret )
  {
    xil_printf( "ERROR : Failed to init HDMI Output Interface\n\r" );
    //exit(0);
  }


#if 0 // Activate for debug
  sleep(1);
  // Status of AXI VDMA
  vfb_dump_registers( &(pDemo->vdma_hdmi) );
  if ( vfb_check_errors( &(pDemo->vdma_hdmi), 1/*clear errors, if any*/ ) )
  {
    vfb_dump_registers( &(pDemo->vdma_hdmi) );
  }
#endif

  //*(volatile int*) (0x7D800000) = 0;

  //-------------------------------------------------------------------------------------------

  //put image processing algorithm below


  Xuint32 new_storage_size = (storage_size*2)/5;
  unsigned int display_size = storage_size;
  unsigned int leftside_x, rightside_x = 0;
  unsigned char entry_flag, reentry_flag, exit_flag = 0;

  unsigned char cbcr = 0;
  unsigned char luma = 0;
  unsigned int pixel_count_total = 0;
```

```
  unsigned int x, y = 0;
  enum {white_to_white, white_to_grey, grey_to_grey, grey_to_white};
  unsigned char current_state, last_state = 0;

  unsigned int found_center, center_y_temp = 0;

  unsigned char stop_flag = 0;

  unsigned int found = 0;

  unsigned char break_flag = 0;

  unsigned int on_counter = 0;
  unsigned int off_counter = 0;


        XGpioPs_Config *GPIO_Config;
        XGpioPs my_Gpio;
        int Status;
#ifdef LEDTEST
        //set up GPIO peripheral pin to be an output and writable
        GPIO_Config = XGpioPs_LookupConfig(XPAR_PS7_GPIO_0_DEVICE_ID);
        Status = XGpioPs_CfgInitialize(&my_Gpio, GPIO_Config, GPIO_Config->BaseAddr);

        XGpioPs_SetDirectionPin(&my_Gpio, PINNUMBER, 1);
        XGpioPs_SetOutputEnablePin(&my_Gpio, PINNUMBER, 1);
#endif

#define TRUE 1
#define FALSE 0
  while(TRUE)
  {
          //initialize state variables
          break_flag = 0;
          stop_flag = 0;
          pixel_count_total = 1;
          found = 0;
#ifdef LEDTEST
          //if a stoplight was seen in the last frame
          if(saw_red_light == 1) {
                  on_counter++;
                  off_counter = 0;
                  //if a stoplight has been seen for 5 frames in a row
                  if(on_counter > 5) {
                          //turn on GPIO peripheral pin
                          XGpioPs_WritePin(&my_Gpio, PINNUMBER, 1);
                  }
          }
          else {
                  off_counter++;
                  on_counter = 0;
                  //if a stoplight has not been seen for 5 frames in a row
                  if(off_counter > 5) {
```

```
                        //turn off GPIO peripheral pin
                        XGpioPs_WritePin(&my_Gpio, PINNUMBER, 0);
                }
        }
        saw_red_light = 0;
#endif
        //load video data into filter
        memcpy(filter, pStorageMem, display_size);
        //look through half of the filter (ignore bottom half of image)
        for(i = 0; i < new_storage_size; i += 2) {
                if(break_flag == 1) {
                        break;
                }
                //set up ycbcr value of current pixel
                cbcr = *(filter + i + 1);       // cbcr
                luma = *(filter + i);           // y

                //get coordinates of current pixel indexed from 1, 1
                x = pixel_count_total % 1920;
                y = (pixel_count_total / 1920) + 1;
                pixel_count_total++;

                //state machine for blob detection
                switch (current_state) {

                        case (white_to_white):
                                if(luma == 255) {
                                        current_state = white_to_white;
                                }
                                else {
                                        current_state = white_to_grey;
                                }
                                //reset if distance is too great
                                if(((x - leftside_x) > 100) && entry_flag) {
                                        leftside_x = 0;
                                        rightside_x = 0;
                                        entry_flag = FALSE;
                                        reentry_flag = FALSE;
                                        exit_flag = FALSE;
                                }
                                last_state = white_to_white;
                                break;

                        case (white_to_grey) :
                                if(entry_flag) {
                                        reentry_flag = TRUE;
                                        entry_flag = FALSE;
                                }
                                if(reentry_flag) {
                                        rightside_x = x;
                                        found++;
                                        found_center = (rightside_x-leftside_x)/2;
```

```
                                        // check that vertical central point was found at
                                        //this location previously
                                        if ((*(filter + i - 2*(found_center) + 1)) == 127) {
#ifdef SINGLELIGHT
                                                break_flag = check_vertical_center_point(
                                                        found_center, 0, filter, i,
                                                        cbcr, luma, x, y, my_Gpio);
                                                if(break_flag == 1) {
                                                        break;
                                                }
#endif
#ifndef SINGLELIGHT

                                        check_vertical_center_point(found_center,
                                                0, filter, i, cbcr, luma, x, y, my_Gpio);
#endif
                                        }
                                        else {
                                                *(filter + i - 2*(found_center+1)+1) = 129;
                                        }

                                        if ((*(filter + i - 2*(found_center+1)+1)) == 127) {
// additional checks for odd right-left since c truncates
#ifdef SINGLELIGHT
                                                break_flag = check_vertical_center_point(
                                                        found_center+1, 0, filter,
                                                        i, cbcr, luma, x, y, my_Gpio);
                                                if(break_flag == 1) {
                                                        break;
                                                }
#endif
#ifndef SINGLELIGHT

                                        check_vertical_center_point(found_center+1,
                                                0, filter, i, cbcr, luma, x, y, my_Gpio);
#endif
                                        }
                                        else {
                                                *(filter + i - 2*(found_center+1)+1) = 129;
                                        }

                                                unsigned int center_y = 0;
                                                unsigned char up_counter = 0;
                                                unsigned int temp_color_up = 255;

                                                unsigned char down_counter = 0;
                                                unsigned char temp_color_down = 255;

                                                while(temp_color_up == 255 &&
                                                   up_counter < 100 &&
                                                   ((y - up_counter) > 0)) {
                                                        temp_color_up = *(filter + i -
                                                                2*(found_center)-
```

```
                                                                        (2*1920)*up_counter);
                                                                up_counter++;
                                                        }

                                                        while(temp_color_down == 255 &&
                                                           down_counter < 100 &&
                                                           ((y + down_counter) < 541)) {
                                                                temp_color_down = *(filter + i -
                                                                        2*(found_center)+
                                                                        (2*1920)*down_counter);
                                                                down_counter++;
                                                        }
                                                        center_y_temp =
                                                                up_counter+down_counter/2;
                                                        if(up_counter > down_counter) {
                                                                center_y = up_counter -
                                                                        center_y_temp;

                                                                if (*(filter + i- 2*(found_center) -
2*(center_y*1920) + 1) == 129)
                                                                {
#ifdef SINGLELIGHT
                                                                        break_flag =
check_vertical_center_point(found_center, -1*(center_y*1920), filter, i, cbcr, luma, x, y, my_Gpio);
                                                                        if(break_flag == 1) {
                                                                                break;
                                                                        }
#endif
#ifndef SINGLELIGHT

        check_vertical_center_point(found_center, -1*(center_y*1920), filter, i, cbcr, luma, x, y, my_Gpio);
#endif
                                                                }

                                                                else
                                                                        *(filter + i-
2*(found_center) - 2*(center_y*1920) + 1) = 127;
                                                        }
                                                        else {
                                                                center_y = down_counter -
center_y_temp;

                                                                if (*(filter + i- 2*(found_center) -
2*(center_y*1920) + 1) == 129)
                                                                {
#ifdef SINGLELIGHT
                                                                        break_flag =
check_vertical_center_point(found_center, center_y*1920, filter, i, cbcr, luma, x, y, my_Gpio);
                                                                        if(break_flag == 1) {
                                                                                break;
                                                                        }
#endif
#ifndef SINGLELIGHT
```

```
        check_vertical_center_point(found_center, center_y*1920, filter, i, cbcr, luma, x, y, my_Gpio);
#endif
                                                                        }

                                                                        else
                                                                                *(filter + i-
2*(found_center) - 2*(center_y*1920) + 1) = 127;
                                                                }
                                                                leftside_x = 0;
                                                                rightside_x = 0;
                                                                entry_flag = FALSE;
                                                                reentry_flag = FALSE;
                                                                exit_flag = FALSE;
                                                        }
                                                        else {
                                                                entry_flag = TRUE;
                                                                reentry_flag = FALSE;
                                                        }
                                                        if(luma == 255) {
                                                                current_state = grey_to_white;
                                                        }
                                                        else {
                                                                current_state = grey_to_grey;
                                                        }
                                                        last_state = white_to_grey;
                                                        break;

                                                case (grey_to_grey) :
                                                        if(luma != 255) {
                                                                current_state = grey_to_grey;
                                                        }
                                                        else {
                                                                current_state = grey_to_white;
                                                        }
                                                        last_state = grey_to_grey;
                                                        break;

                                                case (grey_to_white) :
                                                        if(entry_flag && !reentry_flag) {
                                                                exit_flag = TRUE;
                                                        }
                                                        if(exit_flag) {
                                                                leftside_x = x;
                                                        }
                                                        current_state = white_to_white;
                                                        last_state = grey_to_white;
                                                        break;
                                                default:
                                                        current_state = white_to_white;
                                                        break;
                                        }
                                }
```

```
            if (stop_flag == 1){
            for(i = 0; i < (1920*20); i += 2) {
                        *(filter + i + 1) = 255;       // cbcr
                        *(filter + i) = 90;// y
            }
            }
  }

  //put image processing algorithm above
  //-------------------------------------------------------------------------------------------------
  xil_printf("\n\r");
  xil_printf( "Done\n\r" );
  xil_printf("\n\r");

  sleep(1);
 }

  return 0;
}

#ifdef SINGLELIGHT
unsigned char check_vertical_center_point(unsigned int found_center,  unsigned int extra, Xuint8 *filter, unsigned
int i,
                                                                        unsigned char cbcr, unsigned char luma,
unsigned int x, unsigned int y, XGpioPs my_Gpio) {

            *(filter + i - 2*(found_center) + 2*(extra)) = 0;

            unsigned char flag = 0;
            signed int x_min, x_max, y_min, y_max = 0;
            unsigned long hist0;
            unsigned long hist1;
            unsigned long hist2;
            unsigned long hist3;
            unsigned long hist4;
            unsigned long hist5;
            unsigned long hist6;
            unsigned long hist7;
            unsigned long hist8;
            unsigned long hist9;
            signed int x_min_temp, x_max_temp, y_min_temp, y_max_temp = 0;
            unsigned char temp_cbcr, temp_luma;

            // crude go out and draw a box 31 by 31 (picked semiarbitrarily for proof of concept)
            // for each pixel in side the range
            x_min = -15;
            x_max = 16;
            y_min = -15;
            y_max = 16;
            x_min_temp = -35;
            x_max_temp = 40;
            y_min_temp = -35;
            y_max_temp = 40;
```

```
		hist0 = 0;
		hist1 = 0;
		hist2 = 0;
		hist3 = 0;
		hist4 = 0;
		hist5 = 0;
		hist6 = 0;
		hist7 = 0;
		hist8 = 0;
		hist9 = 0;
		for(x_min = -15; x_min < x_max; x_min++) {
			for (y_min = 0; y_min < y_max; y_min++) {
				cbcr = *(filter + i - 2*(found_center + x_min) - 2*(1920)*(y_min) + 1);  // cbcr
				luma = *(filter + i - 2*(found_center + x_min) - 2*(1920)*(y_min));      // y
#ifndef HISTTEST
				if (luma > 250) {  // sub test to check that we see the right region
					*(filter + i - 2*(found_center + x_min) - 2*(1920)*(y_min) + 1) = 255;  // cbcr
					*(filter + i - 2*(found_center + x_min) - 2*(1920)*(y_min)) = 0;        // y
				}
#endif
#ifdef HISTTEST
				if (luma < 25)
					hist0++;
				else if (luma < 50)
					hist1++;
				else if (luma < 75)
					hist2++;
				else if (luma < 100)
					hist3++;
				else if (luma < 125)
					hist4++;
				else if (luma < 150)
					hist5++;
				else if (luma < 175)
					hist6++;
				else if (luma < 200)
					hist7++;
				else if (luma < 225)
					hist8++;
				else if (luma < 250)
					hist9++;
				// if statement to check histogram values to determine if its a red light
				// do whatever when we know its a red light
				if(hist0 < hist0_v && hist1 < hist1_v && hist2 < hist2_v && hist3 > hist3_v && hist3 <
hist3_v_high && hist4 > hist4_v &&
						hist5 < hist5_v && hist6 < hist6_v && hist7 < hist7_v && hist8 <
hist8_v && hist9 < hist9_v) {
#ifdef LEDTEST
					saw_red_light = 1;
#endif
					flag = 1;
#ifdef REDTEST
					for(x_min_temp = -35; x_min_temp < x_max_temp; x_min_temp++) {
```

```
                                        for(y_min_temp = -35; y_min_temp < y_max_temp; y_min_temp++) {
                                                temp_cbcr = *(filter + i - 2*(found_center + x_min_temp) -
2*(1920)*(y_min_temp) + 1);         // cbcr
                                                temp_luma = *(filter + i - 2*(found_center + x_min_temp) -
2*(1920)*(y_min_temp)); // y

                                                if (temp_luma > 250) {
                                                        *(filter + i - 2*(found_center + x_min_temp) -
2*(1920)*(y_min_temp) + 1) = 255;             // cbcr
                                                        *(filter + i - 2*(found_center + x_min_temp) -
2*(1920)*(y_min_temp)) = 0;         // y
                                                }
                                        }
                                }
#endif
                        }
#endif
                }
        }
        return flag;
}
#endif



#ifndef SINGLELIGHT
void check_vertical_center_point(unsigned int found_center,  unsigned int extra, Xuint8 *filter, unsigned int i,
                                                                unsigned char cbcr, unsigned char luma,
unsigned int x, unsigned int y, XGpioPs my_Gpio) {
        *(filter + i - 2*(found_center) + 2*(extra)) = 0;

        signed int x_min, x_max, y_min, y_max = 0;
        unsigned long hist0;
        unsigned long hist1;
        unsigned long hist2;
        unsigned long hist3;
        unsigned long hist4;
        unsigned long hist5;
        unsigned long hist6;
        unsigned long hist7;
        unsigned long hist8;
        unsigned long hist9;
        signed int x_min_temp, x_max_temp, y_min_temp, y_max_temp = 0;
        unsigned char temp_cbcr, temp_luma;

        // crude go out and draw a box
        // for each pixel in side the range
        x_min = -15;
        x_max = 16;
        y_min = -15;
        y_max = 16;
        x_min_temp = -35;
```

```
		x_max_temp = 40;
		y_min_temp = -35;
		y_max_temp = 40;
		hist0 = 0;
		hist1 = 0;
		hist2 = 0;
		hist3 = 0;
		hist4 = 0;
		hist5 = 0;
		hist6 = 0;
		hist7 = 0;
		hist8 = 0;
		hist9 = 0;
		for(x_min = -15; x_min < x_max; x_min++) {
			for (y_min = 0; y_min < y_max; y_min++) {
				cbcr = *(filter + i - 2*(found_center + x_min) - 2*(1920)*(y_min) + 1);  // cbcr
				luma = *(filter + i - 2*(found_center + x_min) - 2*(1920)*(y_min));      // y
#ifndef HISTTEST
				if (luma > 250) {  // sub test to check that we see the right region
					*(filter + i - 2*(found_center + x_min) - 2*(1920)*(y_min) + 1) = 255;  // cbcr
					*(filter + i - 2*(found_center + x_min) - 2*(1920)*(y_min)) = 0;        // y
				}
#endif
#ifdef HISTTEST
				if (luma < 25)
					hist0++;
				else if (luma < 50)
					hist1++;
				else if (luma < 75)
					hist2++;
				else if (luma < 100)
					hist3++;
				else if (luma < 125)
					hist4++;
				else if (luma < 150)
					hist5++;
				else if (luma < 175)
					hist6++;
				else if (luma < 200)
					hist7++;
				else if (luma < 225)
					hist8++;
				else if (luma < 250)
					hist9++;

				// if statement to check histogram values to determine if its a red light
				// do whatever when we know its a red light
				if(hist0 < hist0_v && hist1 < hist1_v && hist2 < hist2_v && hist3 > hist3_v && hist3 <
hist3_v_high && hist4 > hist4_v &&
						hist5 < hist5_v && hist6 < hist6_v && hist7 < hist7_v && hist8 <
hist8_v && hist9 < hist9_v) {
#ifdef LEDTEST
					saw_red_light = 1;
```

```
#endif

#ifdef REDTEST
                                        for(x_min_temp = -35; x_min_temp < x_max_temp; x_min_temp++) {
                                                for(y_min_temp = -35; y_min_temp < y_max_temp; y_min_temp++) {
                                                        temp_cbcr = *(filter + i - 2*(found_center + x_min_temp) -
2*(1920)*(y_min_temp) + 1);         // cbcr
                                                        temp_luma = *(filter + i - 2*(found_center + x_min_temp) -
2*(1920)*(y_min_temp)); // y

                                                        if (temp_luma > 250) {
                                                                *(filter + i - 2*(found_center + x_min_temp) -
2*(1920)*(y_min_temp) + 1) = 255;               // cbcr
                                                                *(filter + i - 2*(found_center + x_min_temp) -
2*(1920)*(y_min_temp)) = 0;         // y
                                                        }
                                                 }
                                         }
#endif
                                }
#endif
                        }
                }
}
#endif
```