Firefighter Indoor Navigation using Distributed SLAM (FINDS)

April 13, 2012

Written By:

Matthew Zubiel

Nicholas Long

Advisers:

Dr. R. James Duckworth

Dr. David Cyganski

Project Number MQP RJD-1101

Acknowledgements

We would like to acknowledge the contributions of those
who made this project a success:

Joel Williams for his assistance pertaining to the UDP code and timely responses to all of our correspondence.

Matt Lowe for his selfless assistance and good-natured attitude, which without, this project would not have been a success.

Professors Duckworth and Cyganski for their unwavering input and support in the formulation and development of our project, and the inspiration for our work.

ABSTRACT

This project encompassed the design of an image capture and processing unit utilized for indoor tracking and localization of first responders, namely firefighters. The design implemented a Simultaneous Localization and Mapping algorithm, used to track users based on imagery. To predict location, features from each image were identified in real-time, and the difference in the location of those features was tracked frame to frame. Our design consisted of a camera for image capture, an FGPA for real-time processing, and a Simultaneous Localization and Mapping algorithm. Testing consisted of two indoor scenario based tests: a straight line walk and a straight line walk with a 90 degree right turn. Both scenario tests were successful, and accurately tracked our position in the indoor environment.

TABLE OF CONTENTS

TABLE OF FIGURES

TABLE OF TABLES

EXECUTIVE SUMMARY

In December of 1999, a fire broke out in a cold storage warehouse in Worcester, MA. A combination of poor building construction and faulty floor plans led to two firefighters becoming lost, and subsequently running out of air. The original search team was in radio contact with Incident Command, but they were unable to accurately provide their location. A Rapid Intervention Team (RIT), consisting of three firefighters, was assigned to locate the two lost firefighters. Since the floor plans were not an accurate portrayal of the building's structure, the RIT team became lost as well. Next, four more firefighters entered the building as part of the search and rescue operation, and split into two separate teams. One of those teams may have made contact with the initial RIT team, but were also unsure of where they were located. As a result, six firefighters lost their lives. It is generally believed that the crisis could have been avoided had an accurate position of the initial search team had been known and conveyed to the search and rescue teams. As a result of this tragedy, many in the fire service realized the need for firefighter location.

Indoor navigation and location provides great challenges. The current means of location and navigation is GPS, but this method cannot be used for indoor location and navigation. This is because the attenuation of building material weakens GPS signal strength, rendering it unreliable. Attempts at indoor navigation consist of dead-reckoning, RF localization, and inertial localization. Each of these solutions is prone to errors and unreliability. One promising alternative way of performing indoor location is known as Simultaneous Localization and Mapping (SLAM). SLAM provides a method of building a map of the environment as well as keeping track of the user's current location. The SLAM algorithm attempts to track the location based on a set of consistent features determined by a camera.

The main focus of our project was to use SLAM to track the location of a firefighter indoors. The tracking of the firefighter requires the firefighter to wear the sensing device and transmit data to a base station. Because of the processing required to predict location using SLAM, this information cannot be provided to the firefighter directly. Rather, the information collected from the firefighter is transmitted to and processed at the base station. The mobile unit (device worn by the firefighter) contains the camera and a processing unit to send the information gathered by the camera to the base station. This project uses a camera as the sensing device for the SLAM algorithm. Because of the restrictions on data transmission rates over a wireless medium, we determined that detecting features on the mobile unit and communicating the coordinates of those corners, rather than entire images captured by a camera, decreased the data transmission to an acceptable rate and provided detailed enough information to track the firefighter accurately. The mobile unit contains three major components: a sensing device (camera), a processing unit responsible for the feature detection, and a module used to send the processed data. The SLAM processing is done at the base station.

Upon completion of the project, we hoped to accomplish a variety of objectives:

- Capture and process images in real time
- Send resulting data to base station within wireless data restrictions
- Develop method to provide SLAM algorithm with comer-only
- Configure SLAM algorithm to accurately track motion using corner-only input

Design

The entire design was performed on three separate functional units. The first unit was a sensing device capable of capturing images. The second device was a processing unit responsible for processing the images. The third was a base station that runs SLAM software. A component diagram of our design can be seen below.

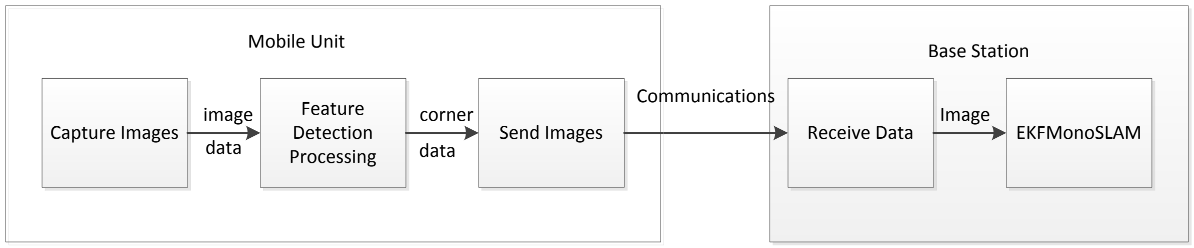


FIGURE 1: TOP-LEVEL BLOCK DIAGRAM

Conclusion

This project provided a way of performing indoor navigation and tracking for first responders using SLAM. Upon completion of this project, all of the goals set forth in were successfully accomplished. Table 4 below shows the project goals and the implementation to complete the goals.

Project Goal	**Implementation**
Capture and process images in real time	VmodCAM Stereo Camera Module with FPGA Processing
Send Resulting Data to Base Station	Ethernet Module on FPGA with base station receiver
Develop method to provide SLAM algorithm with input	Corner Detection on FPGA to base station receiver with black and white images
Configure SLAM algorithm to accurately track motion using corner-only input	Changed settings in EKFMonoSLAM to reflect differences with corner-only input

TABLE 1: DESIGN GOALS AND IMPLEMENTATION

The successful completion of the goals was verified by performing two scenario tests: walking in a straight line and walking in a straight line and performing a 90-degree turn. By accomplishing all of the project goals, the constraints for the project were also met. The VmodCAM and Atlys FPGA provided the firefighter with a light-weight portable unit that would not hinder and movement. Although our design was not implemented using a battery powered device, it would be possible to run the FPGA and camera on a battery. The complete design featured a data transmission rate of around 1 Mbps which is achievable using Wi-Fi. The UDP implementation also allows for easy WiFi expansion if desired.

The major problem with the SLAM algorithm was the processing time required to provide the user with output. This project helped to cut down on the processing time by performing feature detection remotely on the FPGA. This brings the SLAM algorithm one step closer to becoming a system that can be implemented and provide faster output to the base station regarding the firefighter's location.

The successful output from the scenario testing provides a basis for using SLAM for indoor firefighter location. The SLAM indoor location method may remove or lessen some of the challenges presented in previous indoor localization methods such as RF or inertial based tracking. SLAM does not necessarily need to be used independently, but rather it can be used in conjunction with one or more of the existing technologies to possibly provide a more accurate location.

Chapter 1 Introduction

In December of 1999, a fire broke out in a cold storage warehouse in Worcester, MA. A combination of poor building construction and faulty floor plans led to two firefighters becoming lost, and subsequently running out of air. The original search team was in radio contact with Incident Command, but they were unable to accurately provide their location. A Rapid Intervention Team (RIT), consisting of three firefighters, was assigned to locate the two lost firefighters. Since the floor plans were not an accurate portrayal of the building's structure, the RIT team became lost as well. Next, four more firefighters entered the building as part of the search and rescue operation, and split into two separate teams. One of those teams may have made contact with the initial RIT team, but were also unsure of where they were located. As a result, six firefighters lost their lives. It is generally believed that the crisis could have been avoided had an accurate position of the initial search team had been known and conveyed to the search and rescue teams. (Information obtained from National Institute for Occupational Safety and Health (NIOSH) report[1]). As a result of this tragedy, many in the fire service realized the need for firefighter location. There is currently no accurate firefighter location system in existence.

Firefighter location is an aspect of firefighter safety that is often overlooked. During an active structure fire, when a search team enters a building, the layout of that building is most likely unknown to the responders. Smoke fills each room, and because of this, their vision is largely impaired. Firefighters must rely on their sense of feel to navigate along walls through the building. In a sense, navigating through a smoke filled building is similar to navigating through a maze. The incident commander is a fire official charged with maintaining accountability of all firefighters located inside of a structure, as well as directing emergency operations. By providing the incident commander with the latest location of all first responders on scene, it can help him/her direct rescue teams as to where a downed responder may be, if the need arises. The unit at which the incident commander receives information is referred to as the base station.

Indoor navigation and location provides great challenges. The most common means of location and navigation is GPS, but this method cannot be used for indoor location and navigation. This is because the attenuation of building material weakens GPS signal strength, as well as signal degradation due to multipath, rendering it unreliable. Attempts at indoor navigation consist of dead-reckoning, RF localization and inertial localization. Dead-reckoning is subject to errors when speed and direction are not known at all times. This would be the case for firefighter location. Firefighters are moving at variable speeds and directions constantly, making dead-reckoning prone to errors. RF based systems have problems with received signal strength and multipath propagation delays. Inertial systems face the problems associated with inertial drift. One alternative way of performing indoor location is known as Simultaneous Localization and Mapping (SLAM). SLAM provides a method of building a map of an environment as well as keeping track of the user's current location. SLAM can be performed using a number of sensors. Two of the most prominent sensing techniques are using laser range-finders and using cameras. The SLAM algorithm attempts to track the location based on a set of consistent features determined by the sensing device.

The main focus of our project was to use SLAM to track the location of a firefighter indoors. The tracking of the firefighter requires the firefighter to wear the sensing device and transmit data to a base station. Because of the processing required to predict location using SLAM, this information cannot be provided to the firefighter directly. Rather, the information collected from the firefighter is transmitted to and processed at the base station. The mobile unit (device worn by the firefighter) contains the camera and a processing unit to send the information gathered by the camera to the base station. This project uses a camera as the sensing device for the SLAM algorithm. Because of the restrictions on data transmission rates over a wireless medium, we determined that detecting features on the mobile unit and communicating the coordinates of those corners, rather than entire images captured by a camera, decreased the data transmission to an acceptable rate and provided detailed enough information to track the firefighter accurately. The mobile unit contains three major components: a sensing device (camera), a processing unit responsible for the feature detection, and a module used to send the processed data. The SLAM processing was done at the base station.

This project details the research and design of the system in a number of areas. First, background information is provided on SLAM algorithms and feature detection algorithms. After the research on the various feature detection algorithms was completed, hardware components were selected to implement those algorithms. Once the algorithms were implemented on the hardware components and verified using simulation, testing was performed to ensure the implementations worked as they were designed. Once testing was completed, conclusions were drawn based on the results of the testing. The conclusions provide suggestions for further research in the area.

CHAPTER 2 BACKGROUND

The second chapter of this report describes the research performed prior to implementing the project. The background chapter is divided into two subsections: the background research regarding SLAM algorithms and the background research regarding feature detection algorithms.

As described above, the system was designed for a firefighter to wear a mobile unit, containing a camera and processing unit. The data is then transmitted outside to an incident commander. The base station receives and processes the data, and provides the Incident Commander with a graphical representation of where the firefighter is located.

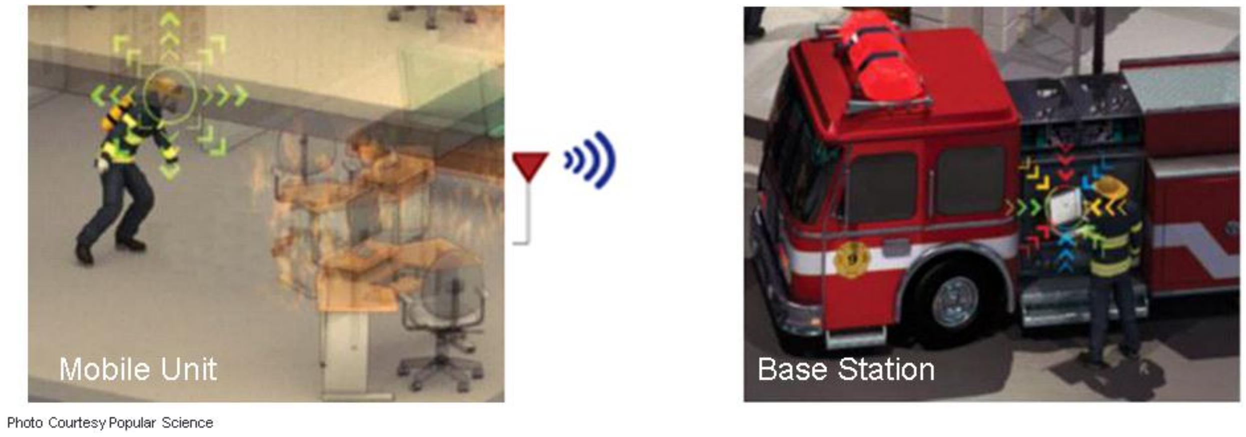


FIGURE 2: POSSIBLE FIREFIGHTING SCENARIO

The subsequent sections describe the research performed into both SLAM and feature detection.

2.1 SLAM

One method of locating people in an indoor environment is to use Simultaneous Localization and Mapping (SLAM): a method of using a series of images to build a map of an unknown environment. SLAM functions by identifying features in an image captured from a video capture device. As the camera moves, the features are tracked and correlated from image to image and movement of the camera is predicted, as well as a map of the environment is generated. The movement of the camera can be used to track its location, and is the primary focus of the project. The map generated of the environment generated by SLAM is needed for processing, but its output is not utilized for our application. We proposed that if a firefighter were to wear the camera, his/her position could be predicted using this method.

2.2.1 SLAM ALGORITHMS

The SLAM technique is growing significantly in the field of robotics. A great deal of research is being done into the SLAM technique and various methods have come out of this research. The most common technique being employed is Kalman Filtering. Kalman filtering is a statistical algorithm used to compute an accurate value over time amid noise and uncertainty. The Kalman filter produces estimates of the true value of measurements and produces a calculated value and estimates the uncertainty of the value. Then a weighted average is computed using the predicted values and measured values. More weight is given to values with lower uncertainty. The Kalman filtering technique uses two stages: the prediction stage and the observation and update stage. In the prediction stage, the state ahead and error covariance ahead is predicted. In the observation and update stage, the

Kalman gain is computed, the estimate is updated with the observed measurement and the error covariance is updated [2]. Because the Kalman filter uses only linear assumptions, additions to the Kalman filter have been developed. The Extended Kalman Filter is a variation of the Kalman Filter that is used in SLAM algorithms. The Extended Kalman Filter does not make linear assumptions; rather it may use differentiable assumptions. The non-linear equations can be used to predict the state, from a previous estimate, but cannot be applied to the covariance. Instead, a matrix of partial derivatives is computed. The computation of the partial derivatives essentially linearizes the non-linear function around the current state[3].

There are also alternatives to using the Kalman Filtering technique and its variations in SLAM. Some issues with the Extended Kalman Filter include quadratic complexity and sensitivity to failures in data association [4]. An alternative approach known as FastSLAM attempts to address the problems with Kalman Filtering. The FastSLAM technique "factors the full SLAM posterior exactly into a product of the robot path posterior, and N landmark positions" [4]. A particle filter is used to approximate the factored posterior. The FastSLAM can be used in environments with highly ambiguous landmarks and can produce maps in extremely large environments [4]. The next section contains information about the software programs that were considered to implement the SLAM algorithms.

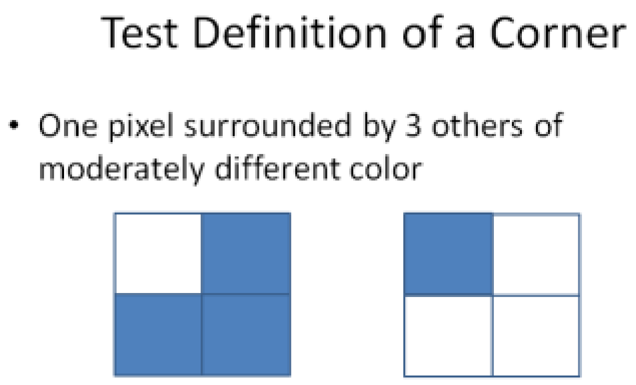
2.2.2 Existing SLAM Software

Two software variations were researched in order to implement our design. The first software program we initially researched was OpenCV. There were a number of factors that made the OpenCV software appealing to our application. The first is that it is free of cost. This would allow us to use our budget to enhance the other parts of the design. Also, the software features a great deal of libraries for image processing and camera calibration. OpenCV also contains functionality for object detection as well as built-in functionality for Kalman Filtering. The software contains a class for standard Kalman Filtering. The documentation suggests that the matrices can be modified to get Extended Kalman Filtering functionality [5].

An alternative software application is distributed by OpenSLAM. The software program is referred to as EKFMonoSLAM [5], and is designed to run in MATLAB. The code takes an input sequence from a monocular camera and uses Extended Kalman Filtering to perform the SLAM operation. It contains complete MATLAB code that needed to be slightly modified to fit our application. This algorithm seemed like our best option because it was completely functional and was suited for our application. Using this algorithm implied that the majority of the SLAM functionality would be run on the base station because of the MATLAB implementation. This implementation required the mobile unit to sample the camera and transmit images to the SLAM software over the wireless link. Our initial idea for sampling the camera and sending images over the wireless link was modified because of the limitations on sending wireless data. Because of these limitations, it was decided that image processing would be done on the mobile unit and the results of that processing would be sent to the base station. EKFMonoSLAM had to be modified because of the new implementation. Based on the reasons stated above, it was decided the EKFMonoSLAM software would be used for our application. The next section details the feature detection algorithms that were researched to cut down on the amount of data sent from the mobile unit.

2.2.3 Feature Detection Algorithms

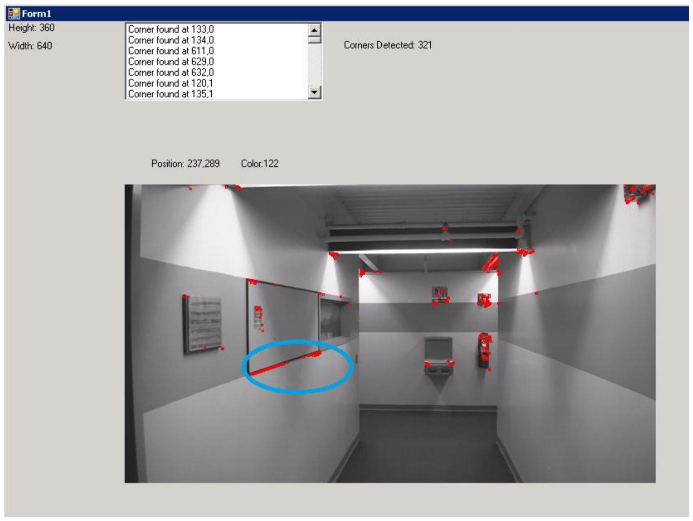
With indoor location being the main concern of the project, it was determined that the features that would remain consistent between frames were corners. Corners exist everywhere in indoor environments and remain constant from frame to frame. Some examples of corners include intersections of walls, door frames, mounted objects, etc. Initially, a test definition of a corner was developed. The test definition of a corner can be seen in Figure 3 below.


Test Definition of a Corner

- One pixel surrounded by 3 others of moderately different color

FIGURE 3: INITIAL TEST DEFINITION OF A CORNER

A simple algorithm for implementing this test definition was implemented and an output frame can be seen in Figure 4 below.



```
Height: 360
Width: 640

Corner found at 133,0
Corner found at 134,0
Corner found at 611,0
Corner found at 629,0
Corner found at 632,0
Corner found at 120,1
Corner found at 135,1

Corners Detected: 321

Position: 237,289     Color:122
```

FIGURE 4: INITIAL CORNER DETECTION OUTPUT

The major area of concern for this corner detection algorithm was that edges were detected along with corners. This effect can be observed in the circled region above. The tracking of edges would be more difficult to track than simply corners. In order to remove the edge detection, research was performed into existing corner detection algorithms.

The two most prominent corner detection algorithms that we found were the Harris Corner Detection and the KLT corner detection algorithm [7]. These corner algorithms were implemented in a similar fashion. The first step in the implementation was to take the derivatives of the image intensity in both the x and y direction. This was computed by taking a pixel below or to the right (depending on the x or y direction) of the pixel for which the derivative is desired and subtracting the intensity value from that of the pixel above or to the left (again depending on the desired derivative). In order to avoid image imperfections registering as corners, a block of 3 pixels in both the x and y directions was used in order to take the derivative as shown in Figure 5.

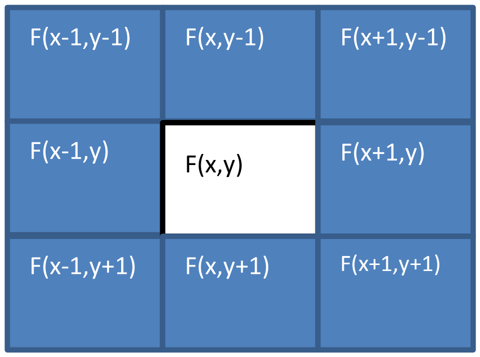


FIGURE 5: BLOCK FOR FINDING DERIVATIVES

Given Figure 5, the derivative of F(x,y) in the x and y directions was computed using the following equations:

$$F_X(x,y) = \frac{\left(F(x+1,y-1)+F(x+1,y)+F(x+1,y+1)\right)-\left(F(x-1,y-1)+F(x-1,y)+F(x-1,y+1)\right)}{6}$$

$$F_Y(x,y) = \frac{\left(F(x-1,y+1)+F(x,y+1)+F(x+1,y+1)\right)-\left(F(x-1,y-1)+F(x,y-1)+F(x+1,y-1)\right)}{6}$$

Based on these two derivatives, the rest of the derivatives necessary for the algorithms (F_x^2, F_y^2, and FxFy) were computed by simple multiplication.

Before performing the corner detection, a simple box averaging algorithm was used in order to smooth the image. This algorithm took the average of the entire 3x3 area (adding the value of each pixel and dividing by 9). The smoothing operation was computed for the derivatives mentioned above. Once the smooth derivatives were found, the matrix used for the Harris Corner detection was constructed. The matrix is shown below.

$$C_{str} = \begin{bmatrix} f_x^2 & f_x f_y \\ f_x f_y & f_y^2 \end{bmatrix}$$

The values of each of these derivatives were computed for every point in the image. In order to determine if the given point was a corner, the following equation was applied

$$H(x,y) = detC_{str} - \alpha(traceC_{str})^2$$

$$0 \leq \alpha \leq 0.25$$

A corner was detected when $H(X,Y) > H_{THR}$ where H_{THR} was specified as zero. Sample outputs from this corner detection algorithm can be seen below.

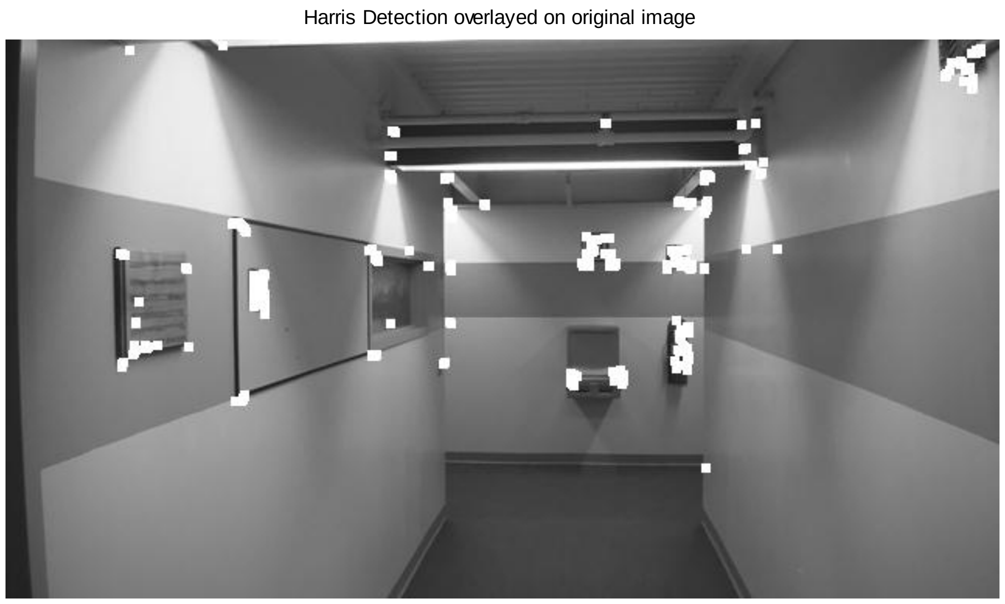


FIGURE 6: HARRIS CORNER DETECTION OUTPUT

Judging by the output shown in Figure 6 above, the Harris Corner Detection Algorithm seemed to suit our application perfectly and was implemented in the project. The next chapter details a top-level overview of the project and set forth goals to be accomplished over the course of completing the project.

Chapter 3 Project Overview

This chapter contains a top-level overview of the project. Also, it details the goals set forth for successful completion of the project.

Knowledge of a firefighter's location within a building is crucial to keeping him/her safe. Our approach was to utilize the SLAM algorithm to track the location of a firefighter by transmitting processed imagery real-time from the mobile unit to an Incident Commander's base station outside. In order for this process to be implemented, certain design constraints were realized.

First, the mobile unit must be small and light-weight enough for the firefighter to wear in order to not impede their ability to move as required.

Second, the mobile unit must have power consumption small enough to be able to be powered off a battery.

The mobile unit must also have the ability to transmit data efficiently enough for wireless communication.

The base station must be able to receive data in real-time in order to provide the SLAM algorithm with input.

In order to meet the constraints mentioned above, a series of project goals were developed. The goals are detailed in the section below.

3.1 Project Goals

The goals below provide the tasks necessary to abide within the constraints mentioned above.

- Capture and process images in real time
- Send resulting data to base station within wireless data restrictions
- Develop method to provide SLAM algorithm with comer-only
- Configure SLAM algorithm to accurately track motion using corner-only input

In order to determine if the goals were accomplished, two scenario based tests were performed. The first test consisted of walking in a straight line down a hallway. The second consisted of walking straight down the hallway and performing a 90-degree right turn.

The next section describes the top-level block diagram implemented in order to accomplish the project goals detailed above.

3.2 Top-Level Block Diagram

The entire design was performed on four separate functional units. The first unit was a sensing device capable of capturing images. The second device was a processing unit responsible for processing the images. The third unit was a device capable of sending data. The fourth device was a base station that is running SLAM software. A block diagram of our design can be seen below.

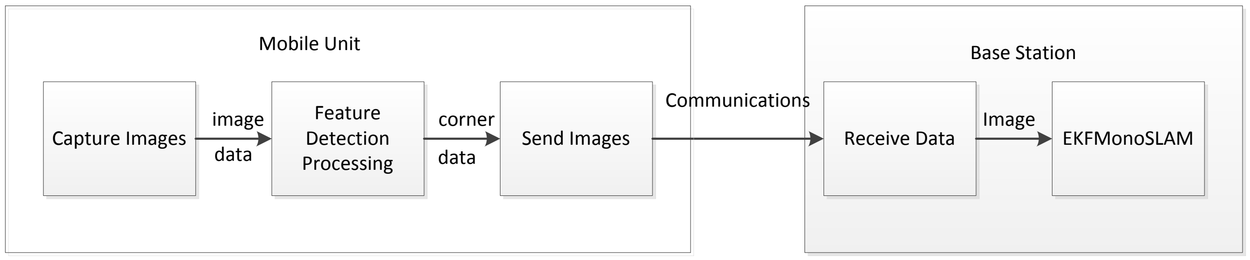


FIGURE 7: TOP-LEVEL BLOCK DIAGRAM

As described in the top-level block diagram above, the mobile unit contained the components necessary to capture the images, perform the feature detection and send the images. The base station contained the components necessary to receive the images and provide the SLAM algorithm with the necessary input. The hardware components selected to implement the functions described in the top-level block diagram can be seen below.

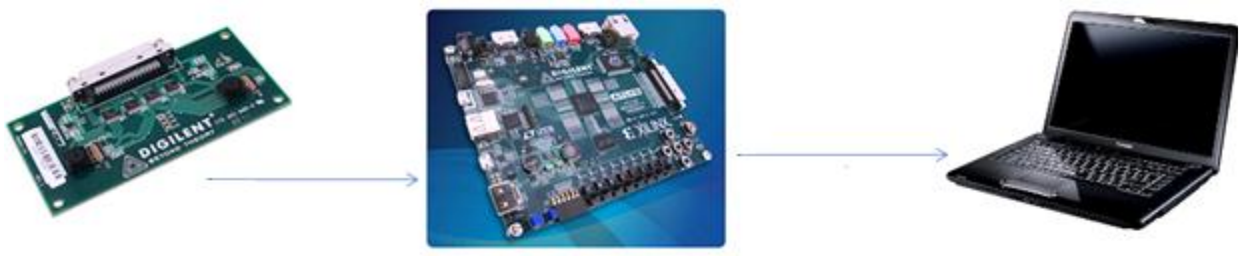


FIGURE 8: HARDWARE REALIZATION

The selection process for the hardware components described above is detailed in the following chapter.

Chapter 4 Component Selection

This chapter details the steps that were taken in order to select each of the hardware components used in our design. The chapter is broken into subsections detailing each of the major components in the design: the processing component, sensing component, and communications component. The base station component was decided to be a laptop running EKFMonoSLAM.

4.1 Processing Units

The two major areas considered for processing units were a Field Programmable Gate Array (FPGA) or an embedded micro controller. The SLAM process itself is an extremely parallelizable process, so the processing on an FPGA was much more efficient. An embedded micro controller would have to use loops to process images whereas an FPGA could perform the processing on one pixel or a group of pixels and replicate these units for the amount of pixels remaining. This allowed for much faster processing. Also, the FPGA allowed for an easier implementation of adding components and making a completely self-contained mobile unit. The ability to add components easily and the parallelizability of the FPGA made that the easy choice for our design implementation. The next section describes the image capturing devices that we used in conjunction with the FPGA.

4.2 Image Capturing Techniques

Research regarding the image capturing techniques was performed in two areas. The first area of research was thermal imaging, because it is better suited to the firefighting environment than a non-thermal camera. The second area was devoted to research into a standard camera that would be best suited for our application. Research was performed in this area because of the high cost and questionable availability of the thermal imagers for our project.

4.2.1 Thermal Imagers

The first part of our research was focused using thermal imaging devices, in order to provide images to EKFMonoSLAM, especially in smoke-filled environments and conditions. This research included comparing various image capturing techniques available. Our first approach was to use a thermal imaging camera in order to perform the image capture. There were various criteria examined when attempting to select the camera that would work best. Our major concentration was not focused on a camera that was completely productized and currently being used by first responders, but rather on camera cores. Thermal cameras that are used in the field often have accessories such as displays that would not be applicable to our application and would more expensive than the standalone (OEM) cores that we focused much of our research on.

There were various specifications that we looked at when examining the OEM cores. The first criterion was the cost of the core. The cost made a sizeable difference because the project has to be completed within a strict budget. The next element we focused our attention on was the type of infrared sensing technology used. We examined two different sensing technologies described below.

The first was microbolometer technology. The microbolometer technology searches for radiated heat that is coming off of objects in the room. The other type of technology examined was Charge Coupled Device (CCD) technology. Instead of looking for objects radiating heat, the CCD technology floods the scene with high IR and searches for objects with lower IR. Since the project entailed image processing, the resolution of the camera also had to be considered. The higher the resolution cameras were preferred, but they were also more expensive than the lower resolution cameras. We watched a demo of the SLAM technique being applied with a standard (non-thermal) camera with a resolution of 320 x 240 and determined that resolution would be the lowest resolution that could be effectively used. We also considered the output of the video from the camera. The output of the camera had to be examined because a decision has to be made on whether we processed the video on board or sent the video wirelessly to a laptop. The output would also determine how we would interface with the camera. Another important criterion was the field of view of the camera. Our system would be more effective with a wider field of view. The final specification that was examined was the spectral wavelengths that the camera could detect. Research was performed regarding which wavelengths were the most effective transmitting through smoke. It was determined that wavelengths in the far IR range (8-14 µm) were the most effective for our application. These wavelengths are associated with the microbolometer technologies. Wavelengths in the near IR range, associated with the CCD night vision cameras, have too much attenuation and would not be applicable for seeing through a smoke filled environment. If we were to use SLAM in a dark room, these cameras may have been suitable.

The research directed us to two distributors of thermal imagers. The first was FLIR. FLIR manufactures two models that would be appropriate for our application. The FLIR Tau 320 and Tau 640 were two models that we considered for our application. The 640 is about double the price of the 320 because of the higher resolution. Both of the cameras can run from 25-30 Hz, with a spectral response in the 7.5 – 14.5 micron range, which seemed ideal for our application. The field of view for the Tau 320 is about 51 x 38 with the widest lens equipped. The field of view for the Tau 640 is about 48 x 38 with the widest lens.

The other series of cameras that seemed to fit our application was the Thermal Eye Series. The cameras were manufactured by L-3 technologies, but distributed by Morovision. There were two Thermal Eye cameras that seemed the most applicable. As with the FLIR models, one has a resolution of 320 x 240 and the other has a resolution of 640 x 480. The video output from both of the cores is from the 25- 30 Hz range. These cores feature a spectral response of 7-14 microns. With the widest lenses equipped, both of the cores feature a field of view of about 50 x 37.

After the research on the thermal imaging cameras was completed, we decided that we would not be able to possess a thermal imager long enough in order to have it be part of our completed project. The cost of obtaining a thermal imager for the duration of the project was too high, so we decided to use a standard (non-thermal) camera for our demonstration.

4.2.2 Standard Image Capture Devices

Based on the above research into the thermal imaging cores, we determined that a standard camera would be more feasible to obtain for our application. The image capture device would be directly connected to our processing unit. There were three image capture devices that we found to be suited for FPGA development. These three units are described below.

4. 2.2.1 LinkSprite Weatherproof Camera and NEXYS2 FPGA

This option utilizes the Digilent Inc. Spartan 3E FPGA contained on the Nexys2 Board and the LinkSprite Weatherproof Camera. This option was appealing to us because the cost was relatively low ($89). The LinkSprite was specifically chosen because of its communications method (RS232) as well as the ability to use infrared imaging (not thermal) to allow usage in situations at night. The block diagram for this implementation can be seen below.

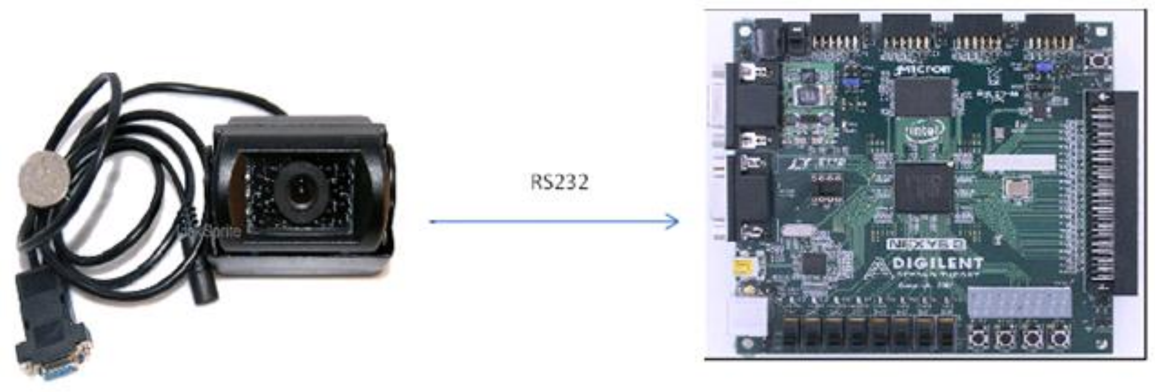


FIGURE 9: NEXYS2 LINKSPRITE IMPLEMENTATION

The major problem with this implementation was the performance of the camera. The camera could not operate at a high enough frame rate with an appropriate resolution for the SLAM software to work effectively.

4.2.2.2 Digilent VDEC1 Decoder Board and NEXYS2 FPGA

This option utilizes the Nexys2 board [8] and the Digilent VDEC1 Video Decoder board [9], [10]. The Nexys2 would send commands to the VDEC1 via the I^2C interface. These commands include a wide variety of clock settings, picture color settings, data throughput (such as closed captioning pass though), in addition to a lot more. The VDEC1 uses an Analog Devices Advanced Video Decoder (ADV7183) to capture the video. The block diagram for this implementation can be seen in Figure 10 below.

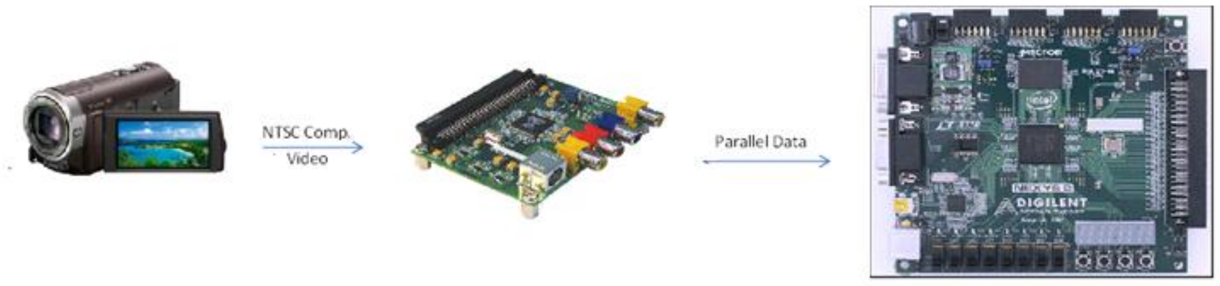


FIGURE 10: NEXYS2 VDEC1 IMPLEMENTATION

The VDEC board can be supplied data in three different formats: CVBS (composite video), SVHS (Y/C), and Component Video (YCrCB). Two 10 bit ADCs capture the input video at 27 MHz. This data is then transmitted by either 8 or 16 parallel data lines to the attached FPGA via a Hirose FX2 port.

This implementation was appealing because of its high expandability. If a thermal imaging camera was to be used in place of a standard camera in the future, this would be very easy to implement, regardless of what its output is. The VDEC1 module as well as either a new camera or HDMI converter would need to be purchased to supply the VDEC1 with analog video input. One downside of this implementation was that there were three components associated rather than just a camera and an FPGA. This implementation required a digital to analog conversion followed by an analog to digital conversion. This process seems inefficient, but it was the tradeoff for the high expandability of the system.

4.2.2.3 Atlys FPGA and VmodCAM

The third option uses the Atlys Spartan 6 FPGA [11] along with a VmodCAM [12] stereo camera module. The Spartan 6 has considerably higher RAM capabilities than that of the Nexys 2, at 128 Mbytes. It also features more logic capabilities and a VHDC connector. The Atlys is specifically designed for real-time video processing. The block diagram for this implementation can be seen in Figure 11 below.

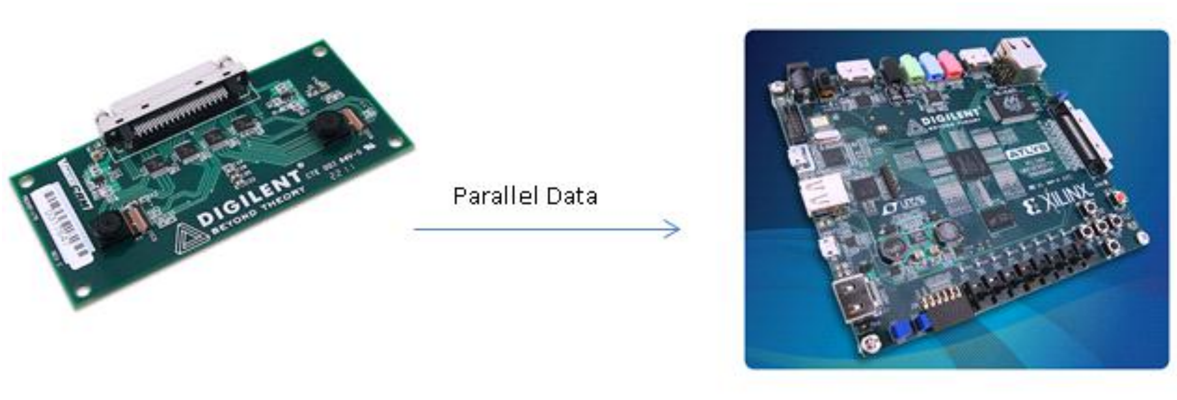
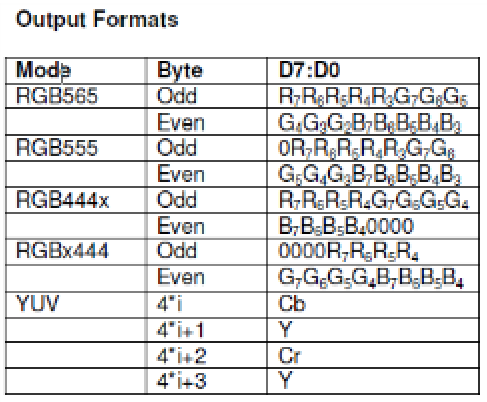


FIGURE 11: ATLYS VMODCAM IMPLEMENTATION

The VmodCAM is an add-on module to the Atlys and communicates in a similar fashion to the VDEC1 board as described previously. Capture commands and resolution settings were sent to the VmodCAM via I^2C protocol. Although the VmodCAM features 2 cameras, we only utilized 1 for our SLAM purposes. The data could be sent back in a pre-specified format: YCrCb, RGB, or Bayer. The output table formats are specified in Table 2 below.



Output Formats

Mode	Byte	D7:D0
RGB565	Odd	$R_7R_6R_5R_4R_3G_7G_6G_5$
	Even	$G_4G_3G_2B_7B_6B_5B_4B_3$
RGB555	Odd	$0R_7R_6R_5R_4R_3G_7G_6$
	Even	$G_5G_4G_3B_7B_6B_5B_4B_3$
RGB444x	Odd	$R_7R_6R_5R_4G_7G_6G_5G_4$
	Even	$B_7B_6B_5B_4 0000$
RGBx444	Odd	$0000R_7R_6R_5R_4$
	Even	$G_7G_6G_5G_4B_7B_6B_5B_4$
YUV	4*i	Cb
	4*i+1	Y
	4*i+2	Cr
	4*i+3	Y

TABLE 2: VMODCAM OUTPUT FORMATS

Some advantages to this design were that it was a completely self-contained module. It also features an extremely high resolution along with an acceptable frame rate for the SLAM algorithm to be effective. Also, the camera provided expansion for Stereo SLAM if the need arose in the future.

4.2.2.4 Comparison and Decision

The final two options were compared side by side in order to make an appropriate decision regarding which camera to proceed with. The comparison table can be seen in Table 3 below.

Specification	**Nexys2 VDEC1/Sony HDR-CX350**	**Atlys/VmodCAM**

Price	Nexys2 = $99
	VDEC1 = $69
	HDMI to comp. video converter = $150
	TOTAL = $308
		Atlys = $199
		VmodCAM = $79.99
		
		TOTAL = $278.99

Maximum Resolution	720x480 (maximum NTSC resolution)	1600x1200
Output file format	8 bit or 16 bit YCrCb	Bayer RGB (probably most useful) YCrCb
FPS	60 FPS (limited by the Sony camera (the camera used for demonstration))	15 FPS at max resolution, increasing with decreasing res
Communications Method	I^{2}C – Control Signal 8 or 16 bit parallel output (YCrCb)	I^{2}C – Control Signal 10 bit parallel output (Bayer, RGB, or YCrCb)
Pros	Works with a plethora of video capture devices with NTSC analog output Can sample at 54 MHz	Runs with the Atlys (Spartan 6) More logic capability 128 Mbytes of RAM vs. 16 with Nexys2 Expansion for stereo SLAM Self-Contained System
Cons	Conversion from digital to analog to digital More components	Can only function with one camera module (no thermal expansion)

TABLE 3: CAMERA MODULE COMPARISON

Based on the comparison above, the camera module chosen was the Atlys Board with the VmodCAM. This option seemed the best because it was self-contained and featured the capability for stereo SLAM expansion. Also, the Atlys Board was designed for video processing.

The final component of our design was the communications component.

4.3 Communications Component

Because of the FPGA implementation described above, the communications component chosen had to be able to interface with an FPGA. The Atlys FPGA allowed for two different communications options. The first option was to use a Pmod expansion component for wireless communication to the base station. The second option was to use the Gigabit Ethernet adapter on the FPGA itself to provide communication to the base station.

4.3.1 Wireless Component

As described above, the wireless component would interface with the FPGA using the Pmod expansion port. The first component that was examined was the PmodWifi Component [13]. This is an IEEE 802.11-compliant wireless transceiver. It features 1 and 2 Mbps data rates. We knew from prior experimentation that for SLAM to function properly, it needs at least 15 frames per second video input. The wireless device would not be able to support this high data requirement. Instead, we decided that we could get away with not sending imagery to the laptop for feature detection, but rather do the feature detection on the remote device itself. Another problem with this component would be the coverage inside of the building because the ability for the 2.4 GHz signal to penetrate through walls is not very effective.

Our second wireless option was to use the PmodRF1 wireless Radio Transceiver [14]. The advantage to this component is the 900 MHz range is much better for penetrating walls, giving us much better coverage inside a building. The downside to this component is the data rate is not as good as the PmodWifi component. In order to determine if a wireless component would be used, simple calculations regarding the amount of data to be sent was performed. It was estimated that 1000 corners would be found in an image. Each corner would require at least 3 bytes to transmit (for a 640x480 image) and the camera would capture 15 frames per second. This calculation yielded the following data rate of about .5 Mbps, which is much higher than the rate supported by the PmodRF1 component. The PmodWIFI component supports a rate up to 2 Mbps, but it was decided not to use this component because of the introduction of multipath and attenuation would add complications to the data link component that may hinder the possibility of getting good test data. It was decided that the Gigabit Ethernet port on the Atlys would be used for reliable data and a data rate much higher than needed. It would also provide a good foundation for WiFi wireless expansion in the future.

4.3.2 UDP Communications

Because we decided against using wireless communications for this project, we decided to implement a communications standard that would be easily transferable to wireless in the future. We decided to use the User Datagram Protocol (UDP) standard to send packets. UDP provides a connectionless way of sending packets to a host at a high rate of speed.

In the TCP/IP protocol, there are two standard ways of which to transmit data, UDP and TCP (transmission control protocol). By nature, UDP doesn’t guarantee delivery to a host, whereas TCP does. We believed that even though UDP may be less reliable, the added benefit of ease of programming to send UDP outweighed the guarantee of delivery: an inherent property of TCP. After testing performance with UDP, we made the determination that we could afford to lose a few packets and still create readable images.

The UDP protocol is shown in Figure 12 below.

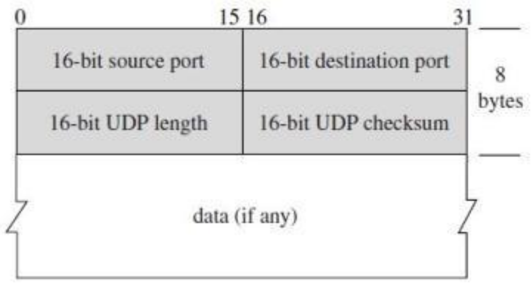


FIGURE 12: UDP STRUCTURE [15]

For every packet sent, we defined a source and destination port, length, and UDP checksum (which remained constant), as well as the data (which changed from packet to packet). The UDP checksum was set to ‘0’ in order to tell the host that it was not calculated. Because UDP is part of the IP Protocol, we wrapped the UDP header and data in the IP header, which is shown below.

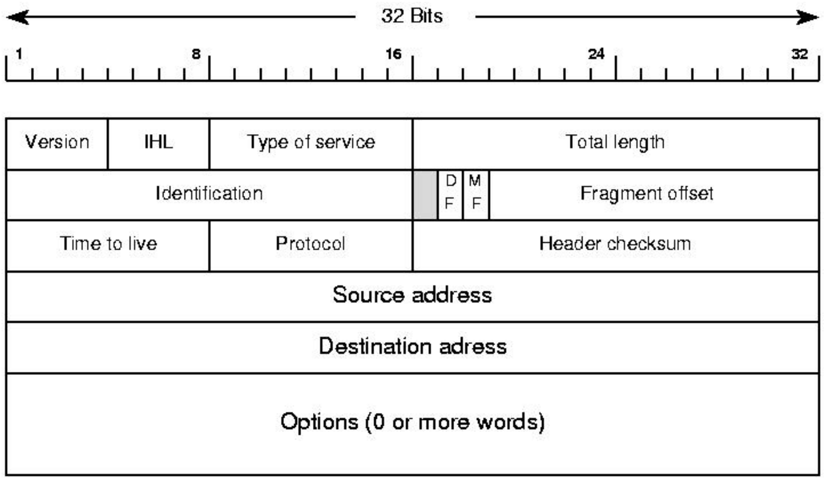


FIGURE 13: IP HEADER [16]

The IP header contains a number of different options, all of which remain constant except for the Checksum which is recalculated for each packet. The destination address was set to the auto-assign IP address of the computer of which we were directly connected to, and the source address was set to be offset by the destination by one address. The IP header was subsequently wrapped in the Ethernet header, as shown below.

Receiver MAC Address	Sender MAC Address	Length	Data

FIGURE 14: ETHERNET HEADER

All of these fields remained constant from packet to packet. Our implementation of the entire UDP Protocol is described in detail in Chapter 5.

After the components were decided, the next step was to implement the various modules described in the background section. The next section deals with the implementation of the modules and the testing of each of the individual components.

CHAPTER 5 MODULE IMPLEMENTATION

This section contains the steps taken in order to implement the various modules that were part of the design. Our design contains four major modules: the VmodCAM interface, the corner detection module, the Ethernet Module and the Ethernet receiver. Some of these modules were broken down into smaller parts for implementation.

5.1 VMODCAM INTERFACE

In order to interface the VmodCAM with the Atlys FPGA, a module needed to be developed. This module was taken from the Digilent website [17] and slightly modified to suit our specific application. The Digilent reference design utilizes two additional components on the Atlys Board: the DDR2 memory and the HDMI output. The Digilent code provided for two interfaces with the VmodCAM module. The first interface was designed to use the camera at its maximum resolution (1600 x 1200); the interface used the VGA (640x480) resolution. The VGA resolution offered higher frames per second rate, which was more applicable to our project, so it was determined that the VGA resolution project was the one that was going to be used. The VGA resolution project was downloaded and programmed to the Atlys Board. When the project was downloaded to the board, the board would display the output from the camera through the HDMI port. In order to implement the corner detection algorithm on the FPGA, the Digilent code had to be modified to perform calculations on the pixels. After some interpretation of the Digilent code, the following data path was determined. The VHDL Code would first capture the image (series of pixels) from the camera using a module entitled "TWI_Ctl". The module was responsible for using the I^{2}C interface to communicate with the camera. The output of this module would be a pixel (16-bits) from the camera. By default, the 16-bit pixels were formatted in RGB565 format. This means the upper 5 bits would correspond to the red component, the middle 6 bits would correspond to the green component, and the lower 5 bits would correspond to the blue component. The next step in the process was to store the pixel in the board's DDR2 memory. The final step in the process is to use another module "DVI transmitter" in order to display the pixels over the HDMI port. The VmodCAM contains two camera modules, "CAM_A" and "CAM_B"; for the purpose of this project, only one of the camera modules was used. The following block diagram depicts the basic flow of the pixels in the existing Digilent Code.

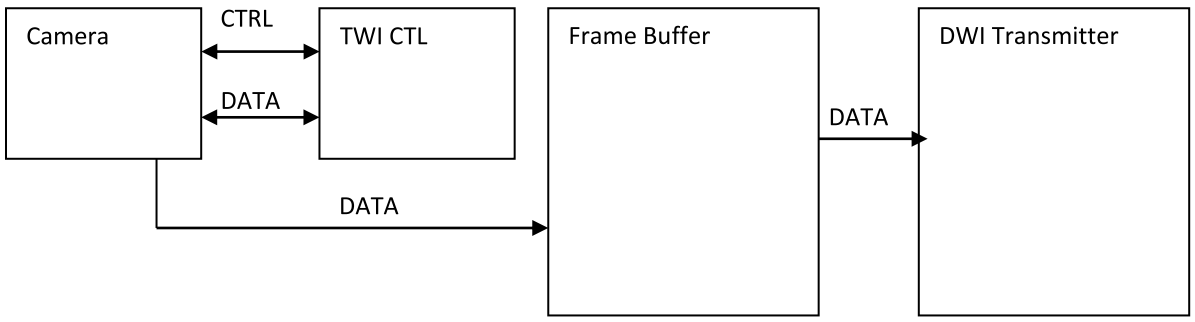


FIGURE 15: BLOCK DIAGRAM FOR DIGILENT CODE

As displayed by the block diagram above, four modules are used in order to implement this design. Each of the modules is described briefly in the subsequent sections. The first module that is described is the "TWI_Ctl" module.

5.1.1 TWI_CTL MODULE

This component is responsible for providing the I^2C interface with the camera. The process for capturing an image from the camera requires multiple steps which are described below:

1. When the DONE_O signal is LOW, it is ready to accept commands.
2. Transfers are performed by putting the TWI Slave Address on the A_I bus and activating the strobe (STB_I)
    a. The read/write is determined by the LSB of the address.
        i. If the transfer is a write the data must appear on the D_I bus before the strobe STB_I is activated.
3. Once the data is read or written, the DONE_I signal pulses for 1 clock cycle.
    a. It is a success if DONE_I = '1' and ERR_O = '0'.
4. A new transfer is started when the MSG_I signal is '1'.

The output from the TWI controller was the pixel[s] captured from the camera. This data is then stored in the CamCtl module. The output from the CamCtl module, again the image, was mapped through internal signals to the input data of the frame buffer. In the frame buffer, three ports were specified. This first port was where the data was read from; the next two ports corresponded to the write buffers for each of the cameras. Each of the cameras is allocated a 2**23 (2^23) bits of memory. This area is the equivalent of storing one image (640 *480*16).

5.1.2 FRAME BUFFER

The frame buffer read portion of the code is relatively short. The major portion of the code consists of a state machine that controls when the read command is applied. The module was set to read 32 32-bit words at a time (this is mainly for speed concerns).

5.1.3 DVI TRANSMITTER

Once the data was read from the frame buffer, it was then sent to the DVI Transmitter in order to be displayed on the screen. In the DVI transmitter module, the 16-bit data was translated into three 8-bit numbers in order to represent the RGB values. The Red value was given as bits 15-11 of the frame buffer data concatenated with three 0s. The Green value was bits 10-5 concatenated two with 0s. The blue value was bits 4-0 concatenated with three 0s. There were also encoding modules included in order to display on the screen. The process described in the three modules above used raw pixel for implementation. Our design required performing a corner detection algorithm on the pixels then sending them through a communications line to the base station. The DVI transmitter was not used explicitly in our design, except for verification.

In order to perform the corner detection algorithm on the pixels, it was observed that there were two potential locations to access the raw pixels. One location was after the pixels were stored in the frame buffer; the other was when the pixels were coming out of the "Cam_Ctl" module.

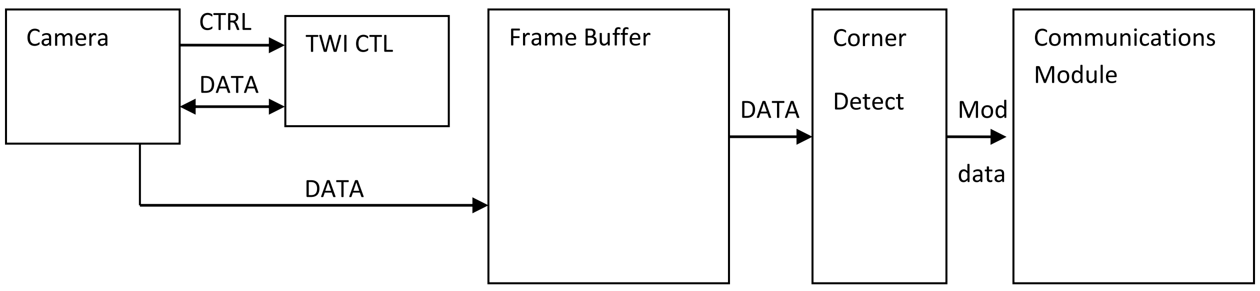


FIGURE 16: FIRST POTENTIAL CORNER DETECTION IMPLEMENTATION

As seen in the above block diagram, the corner detect module would be inserted after the frame buffer and before the DWI Transmitter. This would enable the output of the frame buffer to be the input of the corner detection algorithm. The implementation of this process required modifications to the read section of the frame buffer module. Instead of reading 32 32-bit words at a time, the read had to be modified to support the corner detection algorithm. As described in Chapter 2, the corner detection algorithm required groups of 9 pixels in order to calculate image derivatives. Reading continuous 32-bit words would not give the desired output. The major complication was that these 9 pixels were not stored contiguously in memory. The pixels that were needed to perform the algorithm were three rows in the image. The first three pixels were stored contiguously, then the next three were offset by 1 row vertically or 640 memory spaces. The next three were also stored 640 more memory spaces offset. With the original implementation, no address is specified to the frame buffer module. The address was automatically incremented to read the next 32 words. Some additional code had to be inserted into the read state machine in order to specify the correct addresses. Once these pixels were read from the frame buffer, values possibly would have to be stored again in memory in order to provide comparisons to adjacent patches in order to calculated derivatives, etc.

This implementation was determined to be too complicated for use in our application. First, the interface with the DDR2 memory proved to be more complicated than intended. The DDR2 memory would have to be modified to read non-sequential elements of memory. The DDR2 memory functions by sending out simultaneous addresses. In order to implement this, entire rows of memory would have to be read in order to have the necessary data. Also, timing considerations were a major factor. When the image was processed, new pixels would be coming in. As the new pixels arrived, they would have nowhere to be stored. Storing of these pixels would require the implementation of extra frame buffer[s] to store the next image. The amount of frame buffers required would depend on the speed of the execution. Because of the timing considerations and DDR2 interface, we realized that another location for the corner detection had to be determined.

In order to combat both the timing constraints and the complexity of the DD2 memory component, the second implementation was to operate on the pixels before they were stored in the memory. Once the data came in from the camera, it was immediately stored into a temporary shift register. There was a sequence of three shift registers each capable of storing 640 16-bit numbers. Once the third shift register began to fill, it was possible to calculate Harris values for the pixels in the second register (this process is seen below).

P(0)	P(1)	P(2)	P(3)	P(4)	P(5)	P(6)	P(7)	P(8)	P(9)
P(10)	P(11)	P(12)	P(13)	P(14)	P(15)	P(16)	P(17)	P(18)	P(19)
P(20)	P(21)	P(22)	P(23)	P(24)	P(25)	P(26)	P(27)	P(28)	P(29)

FIGURE 17: EXAMPLE SHIFT REGISTERS

Once pixel 22 was stored, the necessary values in order to compute the Harris value for pixel 11 had already been acquired. At this point, P(0) – P(2), P(10) – P(12) and P(20) – P(22) were sent to another functional unit that calculated the image derivatives and the Harris values. Once these values were calculated, they were the ones stored in memory instead of the original pixel value. The block diagram below shows the implementation of our system.

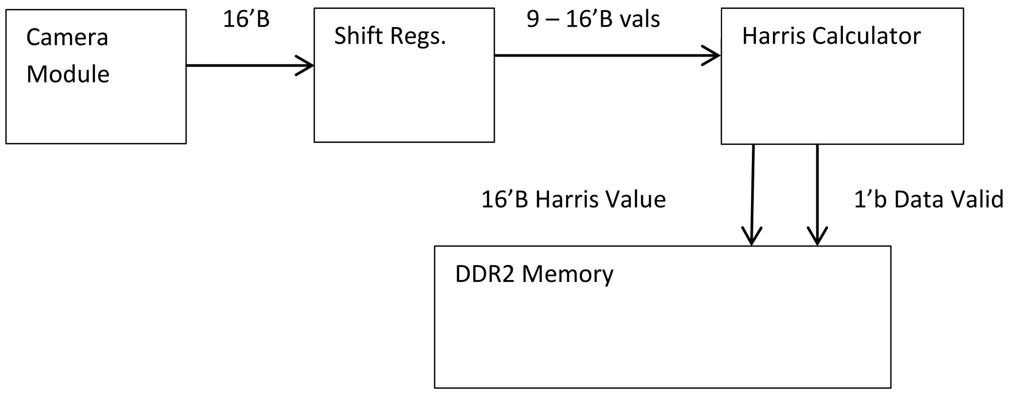


FIGURE 18: BLOCK DIAGRAM FOR SECOND LOCATION

The above block diagram suggests the 16-bit input from the camera would be read in and immediately operated on in order remove the interface with the DDR2 memory. Timing considerations, however, still exist. The calculator module had to be duplicated or modified in order to process values at or above the speed at which they were coming in. This location for accessing the raw pixel data was used in the corner detection algorithm moving forward. The next section describes the steps taken in order to implement the Harris Corner Detection Algorithm.

5.2 Corner Detection Implementation

The corner detection algorithm would be used in order to cut down on the amount of wireless data sent from the mobile unit to the base station. The corner detection algorithm was first implemented and tested in MATLAB in order to gain an understanding for how exactly it worked. After the MATLAB implementation, it was then implemented in VHDL.

5.2.1 MATLAB Implementation

The Harris Corner Detection Algorithm proved to be fairly trivial to implement in MATLAB using images that were already acquired. The first step in the implementation was to take a derivative in both the x and y direction for each pixel in the image. The process for the image derivative is described in Chapter 2. The next step was to square the derivative in the x direction and store it in its own matrix. The same was done for the derivative in the y-direction. The derivatives were also multiplied together and stored. Once the three matrices were created, an averaging box was used to “smooth” the derivatives. The averaging box simply consisted of averaging a group of 9 pixels and using the result as the value for the center pixel. The box was shifted in order to average every element in the matrix. After the averaging was completed, the Harris matrix was then constructed. The Harris matrix is shown below.

$$C_{str} = \begin{bmatrix} f_x^2 & f_x f_y \\ f_x f_y & f_y^2 \end{bmatrix}$$

Each of the elements in the matrix described above is an individual matrix. For instance f_x^2 is the matrix of the derivatives in the x-direction. This Harris value for a given pixel is described as:

$$H(x,y) = detC_{str} - \alpha(traceC_{str})^2$$

$$0 \le \alpha \le 0.25$$

A corner was detected when $H(X,Y) > H_{THR}$ where H_{THR} was specified as zero. When these operations were performed in MATLAB, the following results were obtained.

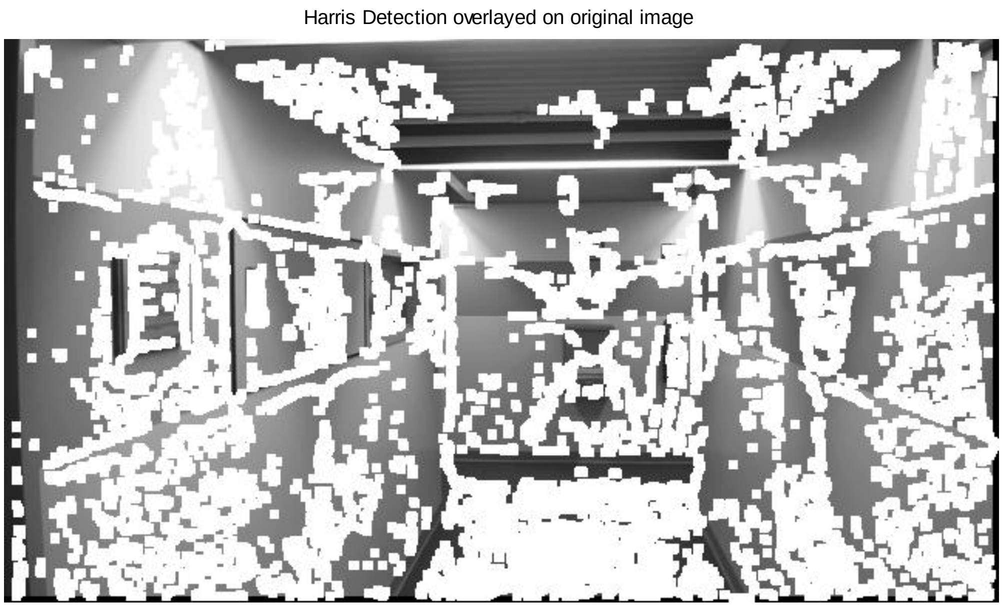


FIGURE 19: RESULTS FROM ORIGINAL CORNER DETECTION

In order to reduce the number of corners obtained, the threshold value was increased to 500. The following output was observed.

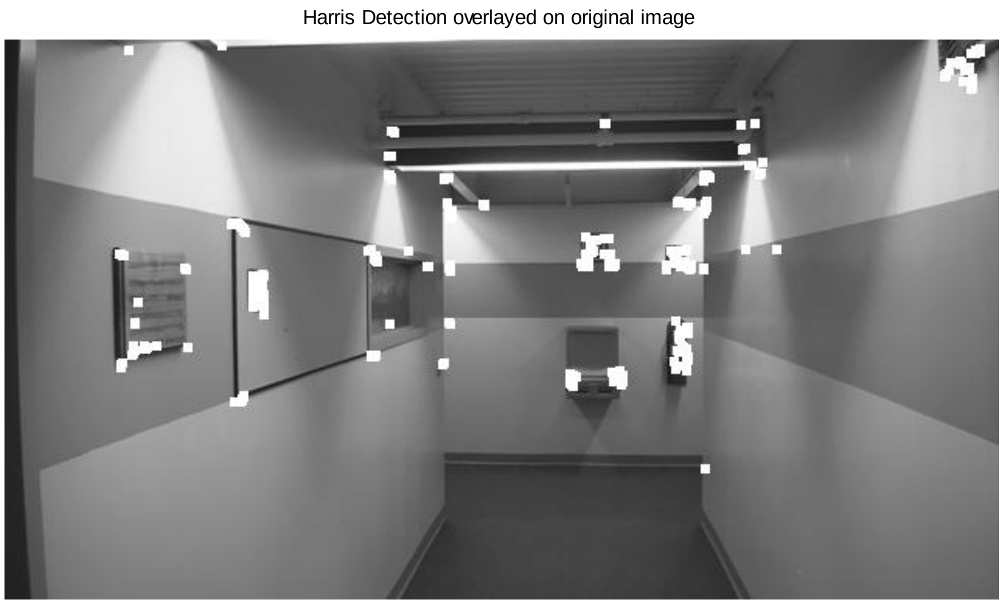


FIGURE 20: HARRIS CORNER DETECTION: THRESHOLD = 500

To ensure that the Harris Corner Detection would be best to suit our application, other attempts at corner detection were attempted in MATLAB. These attempts consisted of taking second derivatives, only using the derivatives and applying a threshold, and a combination of the two methods. The original Harris Corner Detection proved to be the most effective method (the results of the other attempts can be seen in the Appendix) and the Harris method was used moving forward. Once the Harris Corner Detection was implemented in MATLAB, the next step was to implement the module in VHDL.

5.2.2 VHDL Corner Detection Algorithm Implementation

This section describes the process for implementing the Harris Corner Detection in VHDL. In order to test the algorithm, a test bench was constructed. The next section describes the steps used to create the test bench the algorithm would operate on.

5.2.2.1 Simulation of Real Time Data

In addition to developing the VHDL code to process each image, we created a text file to input the color components of each pixel in a test image to the Xilinx Test bench Component. This allowed us to run simulated data through the VHDL code and see output at each step of the code.

The data output from the VmodCAM module is in a format called RGB565. This means that the output is 16 bit, and is in an RGB format. 5 bits were dedicated to the red component, 6 for the green component, and 5 for the blue component.

A C# program was developed to ingest a picture and output each pixel in the RGB565 format. The bitmaps that we were inputting were in a 24 bit format. Because of that, we needed to reduce the color depth of each pixel to fit into a 16 bit format.

C# has a built in class to return a Color structure that contains the Red, Green, and Blue component, each in an integer form from 0 to 255. Because we needed these value to be constrained to a value between 0 and 31 (5 bits for Red and Blue) and 0 to 63 (6 bits for Green), we had to divide this number down.

$$Red_{5\ bit} = Red_{8\ bit} / 8.2258$$

$$Green_{6\ bit} = Green_{8\ bit} / 4.1803$$

$$Blue_{5\ bit} = Blue_{8\ bit} / 8.2258$$

The above equations returned values between 0 and 31 for the Red and Blue components, and 0 and 63 for the Green component. Essentially the threshold of what defines change in color was reduced. The program then output a binary representation of the RGB component (16 bits). Special consideration was given to ensure that 5 or 6 bits would be output for each color, regardless of whether leading zeros were present.

In addition, an extra bit was added to the beginning of the bit stream as a “valid” bit. The VmodCAM module outputs all 16 bits of color data, and then outputs a 1 if the data transmitted was valid.

Example Output: 11001010111001110

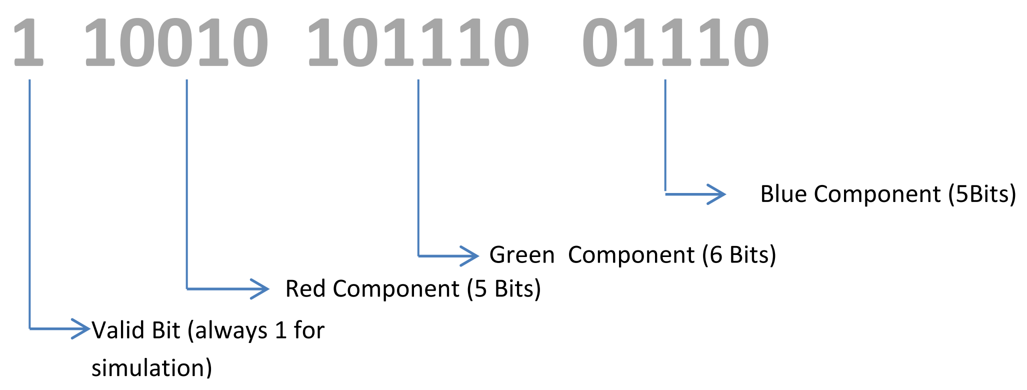


FIGURE 21: RGB565 FORMAT

5.2.2.2 Initial VHDL Implementation

The initial VHDL implementation was designed based on the implementation described above. The calculations were performed before the images were stored in the frame buffer. The corner detection was performed using a pipelined approach. The corner detection algorithm was broken up into four stages. The first stage was to fill 3 shift registers with the values that were coming in from the camera. The second stage was to compute the x and y derivatives. The third stage was to square the derivatives and multiply the x derivative by the y derivative in order to get the components necessary for performing the Harris Corner Detection. The fourth and final stage was to do the corner detection computation in order to determine if the point was a corner or not. The initial pipeline can be seen in Figure 22 below.

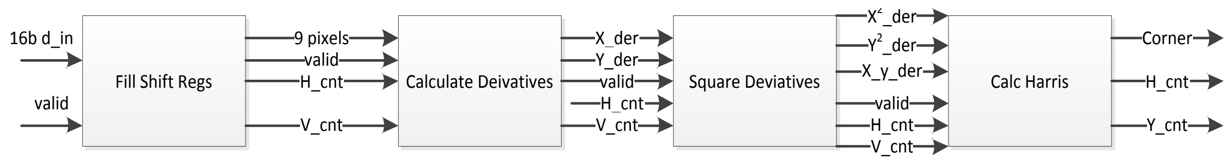
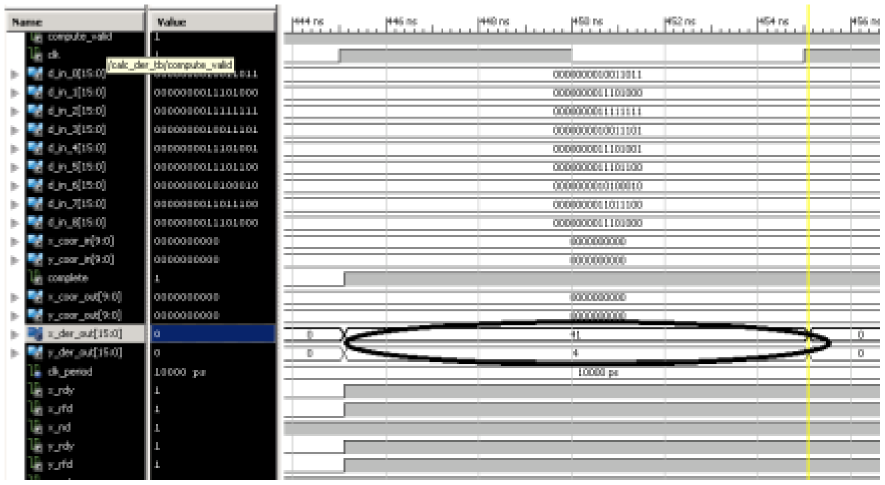


FIGURE 22: BLOCK DIAGRAM FOR INITIAL CORNER DETECTION IMPLEMENTATION

The pipelined approach would be used to ensure the most efficient implementation of the algorithm. This was the case because each new pixel was processed as it arrived. This was an alternative to waiting for entire images to be read and processed.

The first stage in performing the corner detection was to acquire the data required to perform the calculations. In order to determine if a pixel is a corner or not, a block of 9 pixels must be used for computation, with the pixel of interest being in the center. Since the pixels arrived sequentially, it made it difficult to get the surrounding pixels quickly. For example the pixel below and to the left of the pixel of interest took a lot longer to acquire because it has to finish the current row and start acquiring pixels from the next row. In order to perform this function, three “shift registers” that were 640 elements long and contained 16-bit values were used. At the initial start, the first shift register is filled with each incoming pixel. A counter was incremented each time a new pixel is acquired and stored. Once the counter reaches 639, the entire shift register was copied into the second shift register and the new value was loaded into the first element of the first shift register. The first shift register again continued to fill until the count reaches 639 again. At this point, the values from the second shift register populated the third shift register and the first shift register was populated again. Computation began when two elements were in the third shift register. At this point, the module sent the 9 16-bit values, a bit to indicate that the values were valid, and the horizontal and vertical coordinates of the pixel to be examined to the next module. The next module was responsible for computing derivatives.

The second pipeline stage was responsible for computing the image derivatives in both the x and y direction. The x and y derivative computations are explained in the MATLAB section of this report. This process was accomplished in VHDL using the XILINX Core Generator. The most difficult part of the computation was the division by 6. The division would be a lot easier and more efficient if it was by a power of 2. The Core Generator was used to create a divider block. The divider block divided a 16-bit number by 6. The core generator block had a new data input, a divisor input, a dividend input and a clock input. The outputs of the block was a signal to indicate the data was ready, an output to indicate the block was ready for new data, and a 16-bit quotient and a fractional output (which was ignored). The datasheet for the block also indicates that the process would take 20 clock cycles to complete. Two of these divider blocks were created: one was for the x-derivative and one for the y-derivative. This block was verified in a test bench using data acquired from MATLAB and compared to MATLAB outputs for a single pixel. The output from the test bench can be seen below:



```
Name              Value
compute_valid     1
clk               1
d_in_0[15:0]      0000000010011011
d_in_1[15:0]      0000000011101000
d_in_2[15:0]      0000000011111111
d_in_3[15:0]      0000000010011101
d_in_4[15:0]      0000000011101001
d_in_5[15:0]      0000000011101100
d_in_6[15:0]      0000000010100010
d_in_7[15:0]      0000000011011100
d_in_8[15:0]      0000000011101000
x_coor_in[9:0]    0000000000
y_coor_in[9:0]    0000000000
complete          1
x_coor_out[9:0]   0000000000
y_coor_out[9:0]   0000000000
x_der_out[15:0]   0
y_der_out[15:0]   0
clk_period        10000 ps
x_rdy             1
x_rfd             1
x_nd              1
y_rdy             1
y_rfd             1
```

FIGURE 23: TEST BENCH OUTPUT

As displayed by the output of the test bench, the module output the image values of the image derivatives based on the 9 input values. These values were also verified in MATLAB in order to ensure that they were correct.

A state machine was used in this block to control the timing and the outputs. The state machine consisted of two states: busy and idle. The module would be in the idle state until the valid output from the previous module was 1. The module then transitioned to the busy state in order to perform the calculations. The module remained in the busy state until the ready for new data output from the divider block was 1. The outputs of this block were the outputs from the calculations when the ready signal was present on both of the divider units. A valid bit was also sent out to ensure that data is valid. The x and y coordinates of the pixel were also passed through this component.

In order for the algorithm to execute correctly, the derivatives had to be squared and the x derivative times the y derivative had to be computed. Since the derivatives were 16-bit vectors, the derivative squared had to be a 32-bit vector. The core generator was used to perform the 16-bit by 16-bit multiplication. The main motivation for using the core generator was for timing. A simple counter was implemented in order to provide an output to indicate when the multiplication had finished. This module featured the same type of state machine as the module that calculates derivatives. The module remained in the idle state until the valid strobe from the previous module pulses. It then transitioned into the busy state in order for the multiplication to finish. The component could accept new data when it was in the idle state. The outputs of this block were the x derivative squared, the y derivative squared, the x derivative times the y derivative, a valid bit, and the x and y coordinates. The block diagram for this module can be seen below.

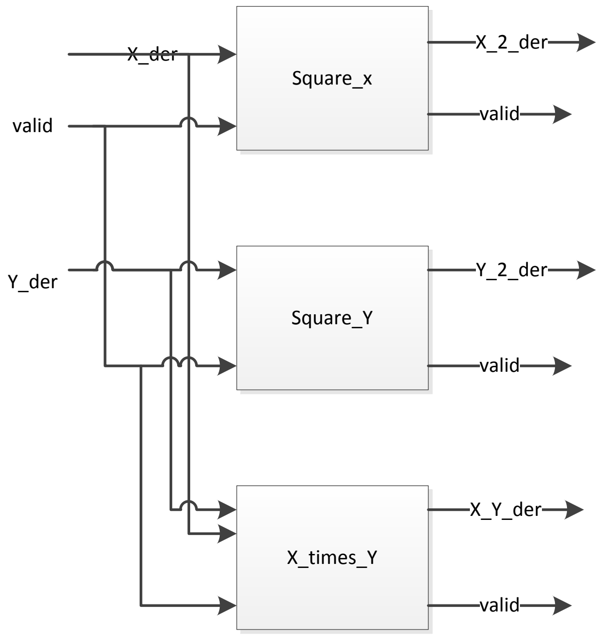


FIGURE 24: BLOCK DIAGRAM FOR SQUARING MODULE

As displayed by the block diagram above, the module contained three multipliers each having a different output. The multipliers also had a valid output that determined when the multiplication is finished. The next step in the corner detection process was to calculate the Harris values for each pixel.

The Harris value was determined using the following equation:

$$H = [(X_{der}^2 * Y_{der}^2) - (XY_{der}^2)] - 0.2((XY_{der} + XY_{der})^{\ 2})$$

In order to remove the division from this equation (because of latency issues) the entire equation was multiplied by 5. This yields the following equation.

$$5H = 5[(X_{der}^2 * Y_{der}^2) - (XY_{der}^2)] - ((XY_{der} + XY_{der})^{\ 2})$$

The ramifications of this multiplication were the Harris value was 5 times what it should have been. This was accounted for when the value was compared to a threshold value determining if the value was a corner. The switches on the Atlys were configured to dynamically control the threshold. This module was implemented using four multipliers, one to multiply the x squared derivative by the y squared derivative, one to square the xy derivative, one to square the xy derivative plus the xy derivative and one to multiply the x squared derivative by 5. The block diagram for this module can be seen below.

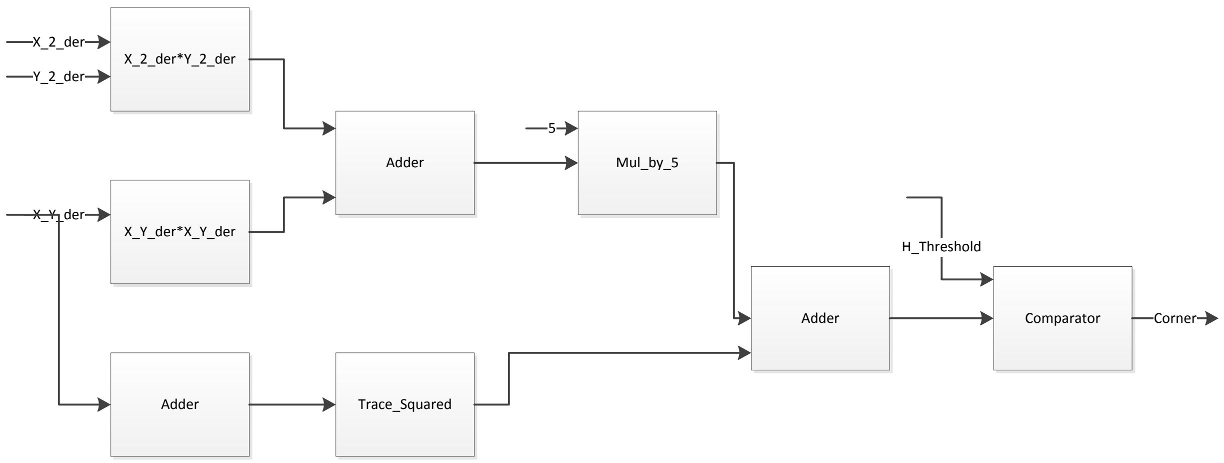


FIGURE 25: BLOCK DIAGRAM FOR HARRIS CALCULATOR

As described by the block diagram above, the module was mainly composed of multipliers and adders (or subtractors). The eventual output from this module was a Boolean determination of whether a pixel was also corner. This module was the final step in the corner detection algorithm.

After this design was synthesized in VHDL, the resource usage was more than the resources on the FPGA. Because of this error, some modifications to the code had to be performed.

5.2.2.3 First Round of Modifications

The second implementation of the VHDL code focused on making improvements to the module that stored incoming pixel data. The modification to this module reduced the resource usage. A block RAM component and a FIFO (First In/First Out) buffer were used in order to reduce the resource usage from the shift registers. The block RAM was responsible for storing a new pixel each time it was sent from the camera. The FIFO was responsible for outputting the 9 pixels that were required for computing the image derivatives. Since only three pixels were new from one implementation to the next, the FIFO would only output the three new pixels required for the calculation. The block diagram for this approach can be seen below.

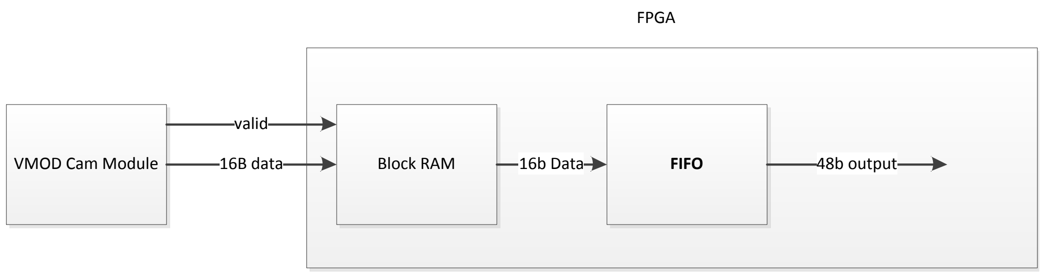


FIGURE 26: BLOCK DIAGRAM FOR ACQUIRING PIXELS

As seen in the block diagram above, the FIFO sent 48 bits to the next component in order to calculate the derivatives. These 48 bits were broken up in the next component for calculation. The major consideration for this component was the timing. Since the RAM could not read and write different addresses at the same time, there would have to be enough time in between acquiring pixels in order to write three new values to the FIFO.

Based on the camera datasheet, the camera is capable of acquiring 30 fps when it is in "capture mode". Based on a 640 x 480 resolution and each pixel being 16 bits, the following calculation can be derived.

$$1\ image = 307200\ pixels\ x\ \frac{30\ frames}{1\ second} = 9216000\ \frac{pixels}{second} = 9.216\ MHz$$

These calculations result in the fact that each pixel is arriving at a rate of 9.216 MHz. Using the 100 MHz clock on the Atlys, it was possible to read three pixels in the time in between each new pixel arriving. Based the new output from this module, the Calculate Derivatives module also had to be adjusted.

Since the input to this module is now 1 48-bit number instead of 9 16-bit numbers, some adjustment had to be performed for this module. When the 48 bits came in, the module broke these 48 bits up into three 16-bit numbers. Bits 15-0 were the upper left pixel, 31-16 were the middle left pixel, and bits 47-32 were the lower left pixel. This was implemented in this module using three shift registers. The block diagram for this module can be seen below.

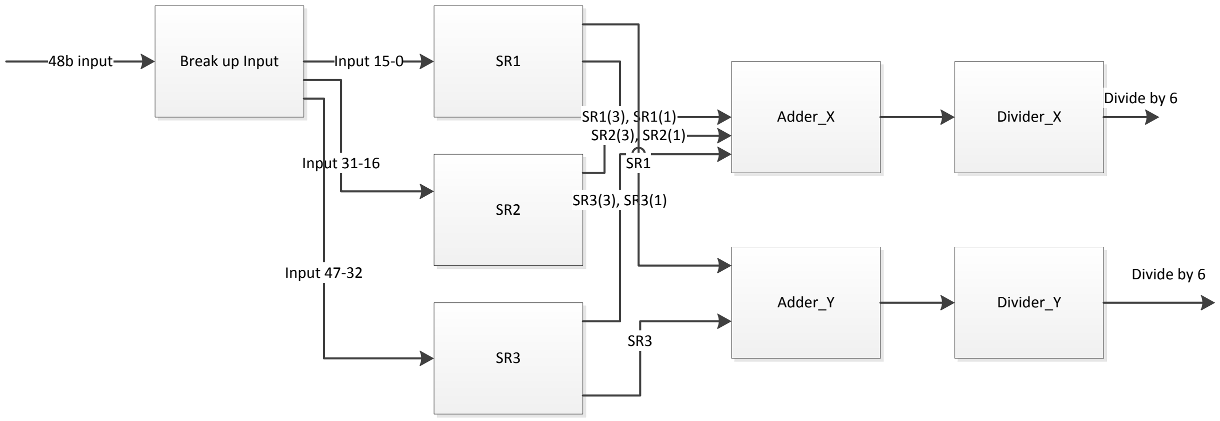


FIGURE 27: BLOCK DIAGRAM FOR DERIVATIVES

The block diagram above displays the implementation for the Calculate Derivatives module. Using this implementation, the rest of the components can remain the same because the output from the derivatives module is the same. After some testing was performed on this module, it was determined that the division took too long. Pixels entered the Calculate Derivatives module before the division using the previous pixels was finished. This division latency required code modifications.

5.2.2.3 Second Round of Modifications

The second round of modifications included improvements to the Acquire Data module and the Calculate Derivatives module. Also, a new module was added to account for the additions made to the Calculate Derivatives module.

After some additional testing, it was determined that the previously described Acquire Data module was not behaving exactly as it was designed to. The output from the FIFO was not always correct. Because of these difficulties, the module was slightly changed to gain a more accurate output. The FIFO component was eliminated, and simple flip flops were inserted instead. These flip flops would each store one of the pixels necessary for the next step in the process. After all three of the pixels were read, the valid output of this component would be high and the three pixels were output to the next component. The adjusted block diagram can be seen below.

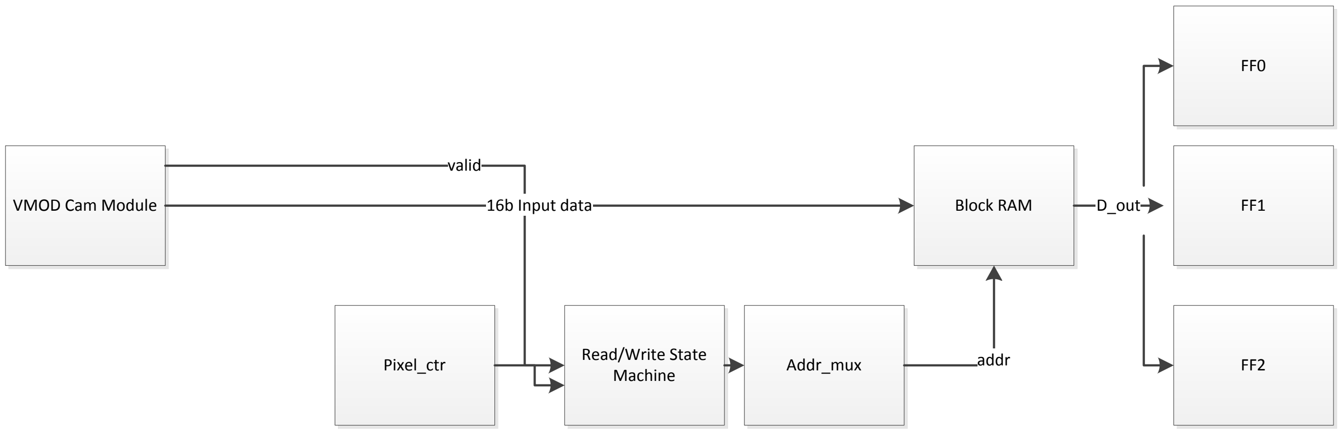


FIGURE 28: ACQUIRE DATA MODULE AFTER 2ND MODIFICATION

In the previous implementation, the next step was to feed these three data values into Calculate Derivatives. After some experimentation with this implementation, it was determined that division was a bottleneck. For each division, three new values came into the module. Because the divider module was busy, these new values were not calculated. This bottleneck required the implementation of three dividers instead of one. With this implementation, when one divider was busy, the input data went to the next divider and so on. In order for this to be implemented, another module was designed. This module was responsible for essentially being a multiplexer for the different divider modules.

The divider multiplexer had an input of the three data values from the Acquire Data. The output of the module was the dividend that was needed for the input to the dividers. It also output a value, which determined the divider to be used, as well as a bit that determined if the output from the module was valid. The main components of this module were three shift registers. Since 9 pixels were required to calculate the derivative for one pixel, and only three pixels were input to this module, 6 of the pixels from the previous calculation were retained. This is shown in Figure 29 below.

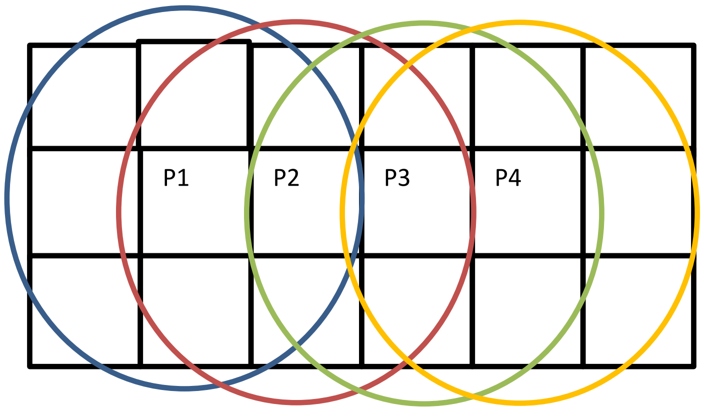


FIGURE 29: ILLUSTRATION OF PIXELS REQUIRED FOR DERIVATIVE CALCULATION

The above figure illustrates the pixels necessary for taking the derivatives of the pixels P1, P2, P3, and P4. The pixels required for P1 were enclosed in the blue circle, P2 enclosed in the red circle, P3 enclosed in the green circle and P4 enclosed in the orange circle. As displayed by the figure, only three new pixels were required to calculate the derivative for adjacent pixels. This is the reason behind the shift register implementation for this module. The three new values from the previous module were fed into the shift register and the previous values were shifted over and one of the values was removed. Once the shift register is filled, the required subtraction for calculating the derivatives in the x and y direction were computed. A counter was also used to determine which divider would be used to perform the calculation. The block diagram for this module can be seen below.

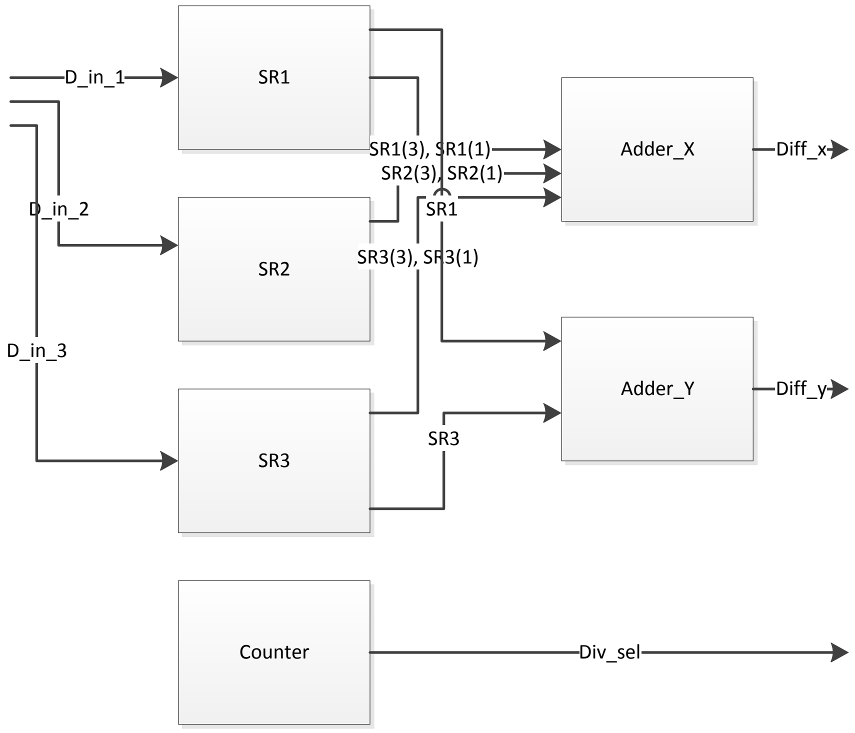


FIGURE 30: BLOCK DIAGRAM FOR DIVIDER MULTIPLEXER

The outputs of this module were then fed into the divider module. The Calculate Derivatives module was duplicated three times in order to accommodate the latency associated with division.

The Calculate Derivatives module was simply a hardware divider with an enable input, an x input and a y input. The output from this module was the x input divided by 6, the y input divider by 6, and a bit to signal when the division was finished. The block diagram for this module can be seen below.

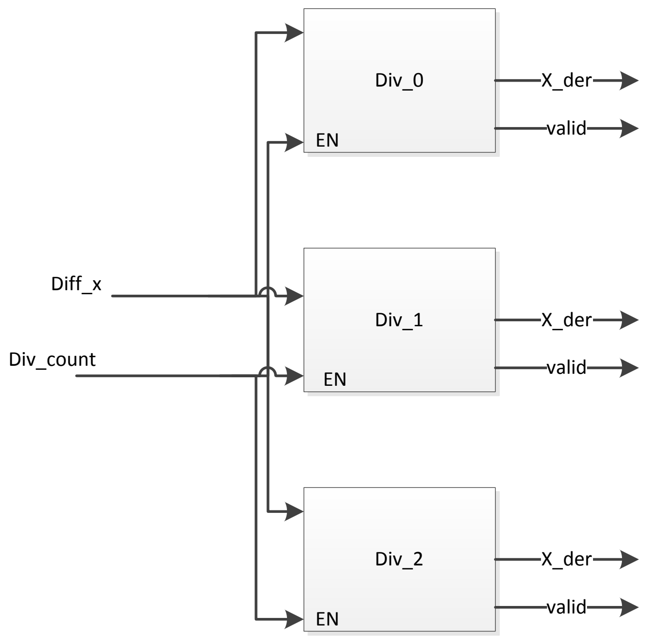


FIGURE 31: BLOCK DIAGRAM FOR CALCULATE DERIVATIVE MODULE AFTER 2ND MODIFICATION

The block diagram above shows the calculation of the derivative in the x direction. Each of the "Div" blocks represents the Calculate Derivatives modules. This was duplicated three times to meet timing constraints. The enable inputs of the dividers were selected with the counter output of the multiplexer module described above. This implementation produced the desired results. The next step in the corner detection process was to produce the squared values of x and y derivatives and multiply the x and y derivative. These calculations would result in the necessary values to calculate the Harris Value.

Because of the modifications made to this component, the outputs of the dividers were multiplexed at the top level. The input to the Square Derivatives module was the Calculate Derivatives module with a valid output. The implementation of the next two modules remained the same. After these modifications were made, the only remaining components were those that would smooth the averages of the pixels before they were fed into the Harris Module.

5.2.2.4 Third Round of Modifications

The Harris Corner Detection Algorithm used a box averaging technique to smooth all of the derivatives before performing any calculations on them. This was done by taking a group of 9 pixels, summing them and dividing by 9. The resulting value would represent the average value for the center position of the 3x3 array. This averaging box technique was completed for the x^2 derivative, the y^2 derivative, and the xy derivative. The figure below describes the pixels used for the averaging technique.

P0	P1	P2
P3	P4	P5
P6	P7	P8

FIGURE 32: AVERAGING BOX FOR 9 PIXELS

Using the 9 pixels shown in Figure 32, the following equation would compute the average.

$$\widehat{p_4} = \frac{1}{9}\sum_{i=0}^{8} p_i$$

The above equation states the “smooth” value for p4 was the sum of all of the surrounding pixels (including p4) divided by 9. This average was computed for all of the pixels mentioned above. The main problem with this implementation was the division by 9. As seen with the Calculate Derivatives module, division was the biggest bottleneck in the implementation and required three dividers. Also, because this average had to be computed for three separate sets of numbers, it required three dividers for each averaging component; amounting to 9 dividers in total. Because of this division bottleneck, experimentation was performed in MATLAB to modify the code from dividing by 9 to dividing by 16. This would present a much better alternative for the FPGA.

P0	P1	P2	P3
P4	P5	P6	P7
P8	P9	P10	P11
P12	P13	P14	P15

FIGURE 33: 16 PIXEL IMPLEMENTATION

Using the averaging box described in Figure 33, following equation would describe the implementation.

$$\widehat{p_5} = \frac{1}{16}\sum_{i=0}^{15} p_i$$

The equation above proves to be much more suitable for an FPGA implementation. This average has a greater number of pixels below and to the right of the target pixel. This new averaging box was tested in MATLAB before it was implemented on the FPGA. The MATLAB outputs can be seen in the Appendix. The images in the Appendix show there is a small difference between using the two different averaging boxes. Based on these observations, we felt that the 16 pixel averaging box would be adequate for the VHDL implementation.

Once the 16 pixel averaging box was tested in MATLAB, the next step was to interface it with the FPGA. This interface required 16 non-consecutive pixels for the computation. This was accomplished using methods similar to those used in the Acquire Data from the camera. The implementation required the use of 6 modules because the implementation for one derivative is performed using two modules and is duplicated three times because of the three derivatives required.

The first step in the computation was to obtain all of the derivative values required. This was not a trivial task because the derivative values did not arrive consecutively. The top 4 derivative values arrived; then, 636 derivative values later, the next 4 derivative values arrived and so on. In order to store all of the derivative values required for the computation, a block RAM component was used from the Core Generator. This component stored 4 rows of 638 32-bit derivative values. Only 638 derivative values were stored in a row because the derivative could not be calculated for the first derivative values in a row or the last derivative values in a row. The data coming in, (corresponding derivative) was stored in the block RAM as it arrives. The data was output when there was enough data for an average to be calculated. Four data elements was output to the next module in order to calculate the average. The block diagram for this module can be seen below.

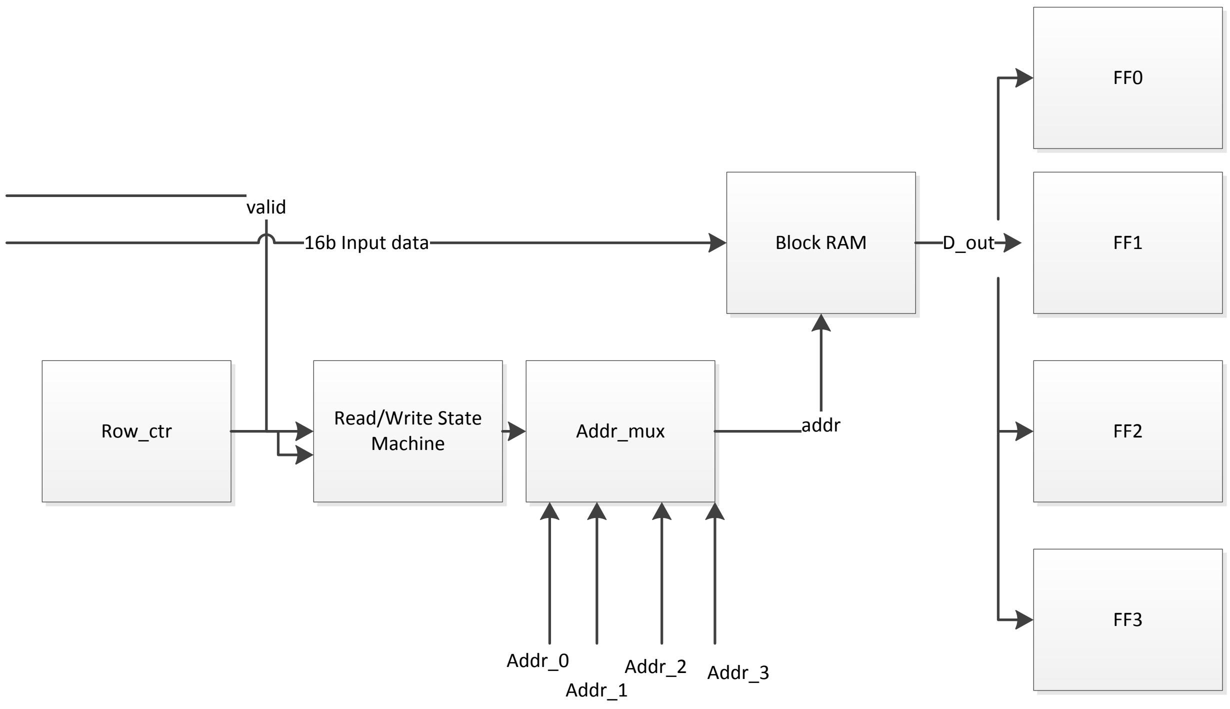


FIGURE 34: BLOCK DIAGRAM FOR ACQUIRING DERIVATIVES

The block diagram in Figure 34 above shows the implementation for the store module. The Row_ctr block incremented by 1 every time 638 derivatives were acquired. When the row counter reached 3, it was possible to output data. At this point, an address multiplexer was used to determine the address of the read. In order to determine which address to use in the block RAM, a counter was used as the input to the multiplexer. Flip flops were used to latch in the output data to transmit it to the next module. Test bench waveforms can be seen below.

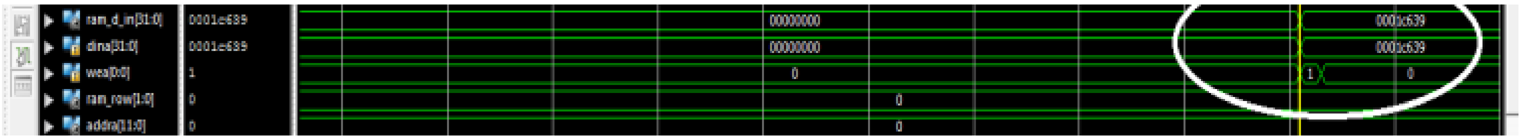


FIGURE 35: WRITE TO ADDRESS 0

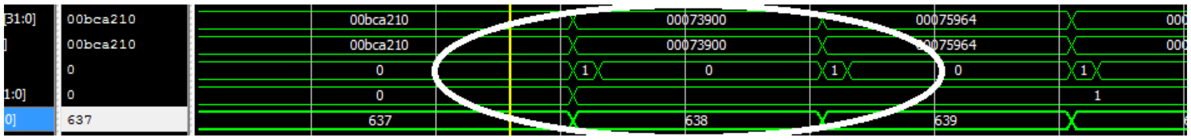


FIGURE 36: WRITE TO ADDRESS 638

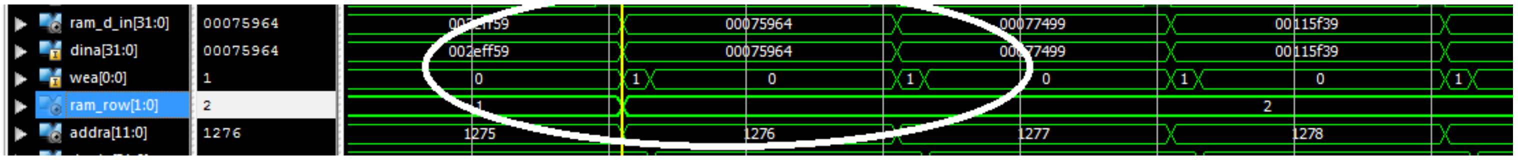


FIGURE 37: WRITE TO ADDRESS 1276

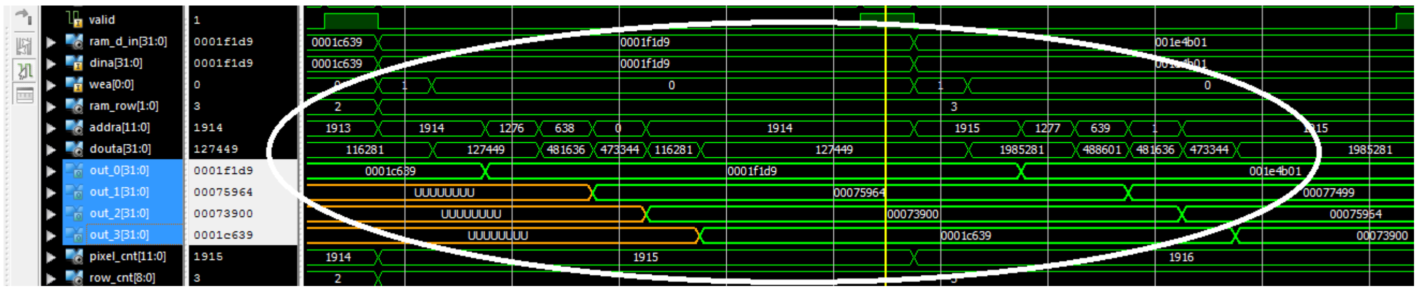


FIGURE 38: WRITE TO ADDRESS 1914 AND READ OTHER 3

The test bench waveforms above display writing four values and reading them in as out to the next module. The first value 0x1c639 is written to address 0, 0x73900 is written to address 638, 0x75964 is written to address 1276, and 0x1f1d9 is written to address 1914. Once address 1914 is written, the four values were then read back and latched into the out_0 – out_4 outputs. These outputs were then used in the next module to calculate the average.

This module was duplicated three times: one for the x^2 derivatives, one for the y^2 derivatives and one for the xy derivatives. The next module was responsible for performing the averaging of the 32-bit derivative values.

The data input to the averaging module was four 32-bit values representing the square of the corresponding derivative. Because 16 values were required for computation, a shift register was used. This was very similar to the module for the selecting dividers and calculating the differences. Again, only 4 of the values changed between iterations. The block diagram for this module can be seen below.

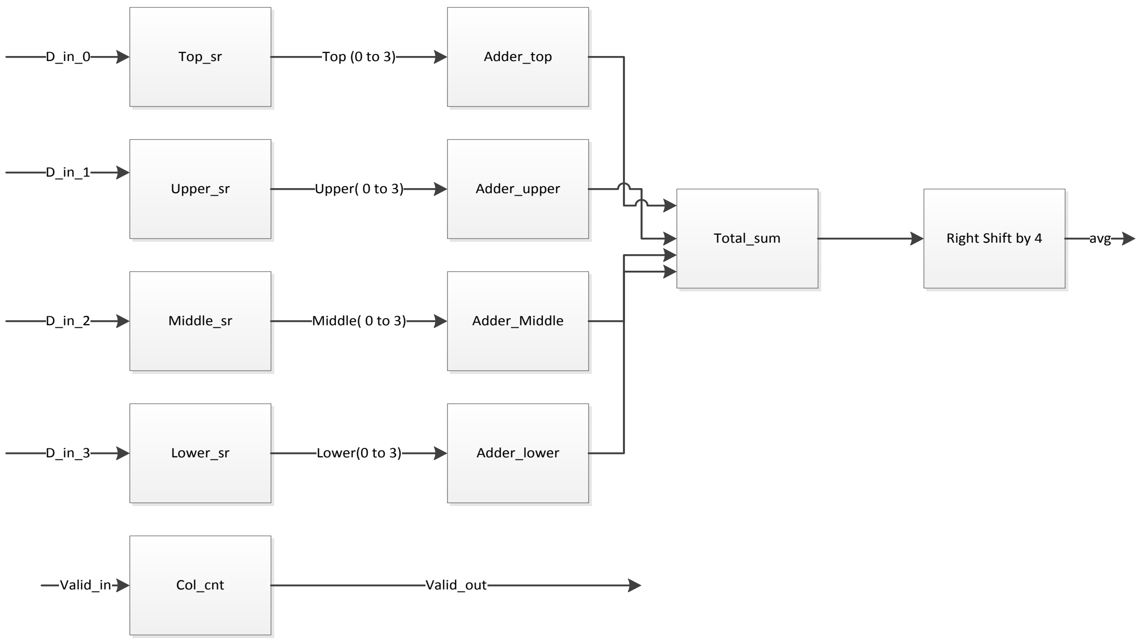


FIGURE 39: BLOCK DIAGRAM FOR AVERAGING MODULE

The test bench waveforms for this module can be seen below.

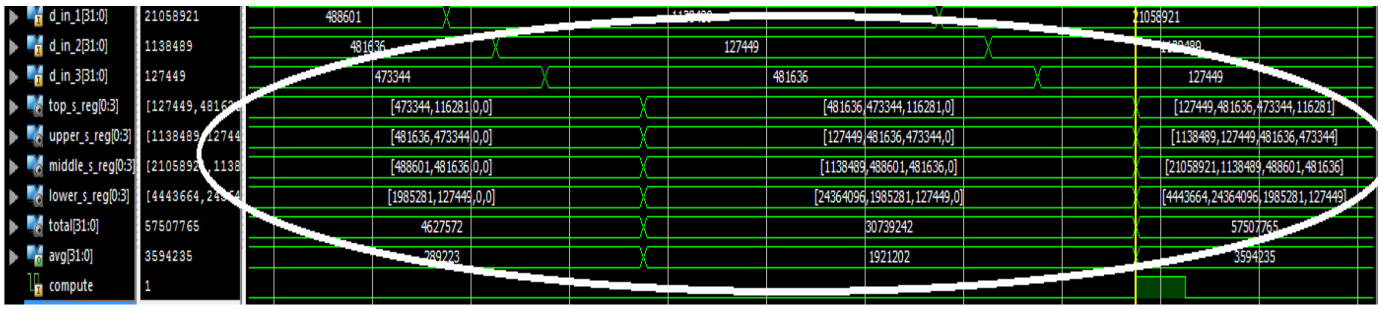


FIGURE 40: TEST BENCH FOR AVERAGING MODULE

The test bench above displays the behavior of the shift registers in the averaging module. The input data in the figure above was in an integer format to verify the operation of the averaging. As displayed by the test bench, it latched in the values and shifted every time new values were acquired, but does not output until all of the shift registers were full. When all of the registers were full, it output the average (as displayed above). The test bench verified the operation of the averaging module.

The next step in the process was to interface the averaging module with the rest of the design. The averaging modules were inserted after the derivatives were squared. The averages were fed into the Harris calculation module. The top-level block diagram can be seen below.

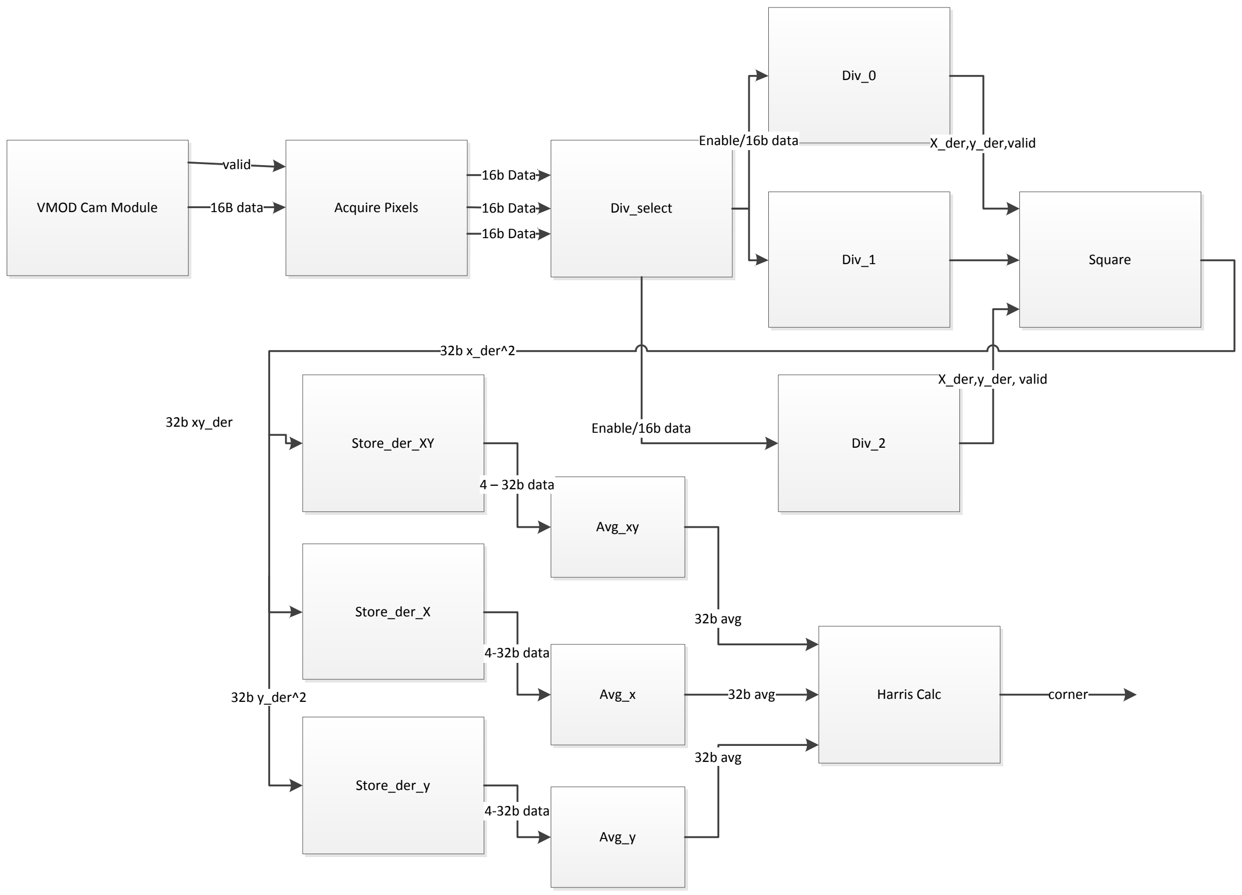


FIGURE 41: TOP LEVEL BLOCK DIAGRAM

The top level block diagram above shows the flow of the complete design. The pixels were acquired from the VmodCAM module and stored in block RAM. The values were then output to a component that calculated the required numerators for the derivatives. It also selected one of three dividers to increase performance. The dividers then divided the input value by 6 and output the image derivative to the next module that squared the value. The module squared the x and y derivative and multiplied them together to form the xy derivative. These values were stored again in a block RAM component for averaging. Four values were output from the block RAM to perform an average. The averages of all three of the aforementioned derivatives were taken in 16 pixel neighborhoods and output to the Harris Calculator to determine a corner.

The implementation described above calculated correct values based on the 16-bit inputs. The input is in RGB565 format. The difference in values was too large to get valid Harris values. There were too many values that were registering as corners. The problem with the implementation was all of the 16-bit RGB565 values were being used rather than the pixel intensity values. In order to get the correct output, the data was modified to add the R, G and B components in order to get an intensity value between 0 and 128. The next step in the design was to modify the modules in order to use the smaller values. After some experimenting was done in the test bench, it was noticed that when the values were added to each other (in the Calculate Derivative phase), some numbers overflowed and appeared to be smaller. In order to combat this error, an extra bit was added to compensate for the overflow. When this format was tested, it was determined that when two values were subtracted and the result was negative, the result was incorrect. In order to combat the sign problem, another bit was added to signify the sign bit. This was compensated for in the final stage by checking the MSB when comparing it to the threshold value. The entire block diagram for the corner detection algorithm (with appropriate bits) can be seen below.

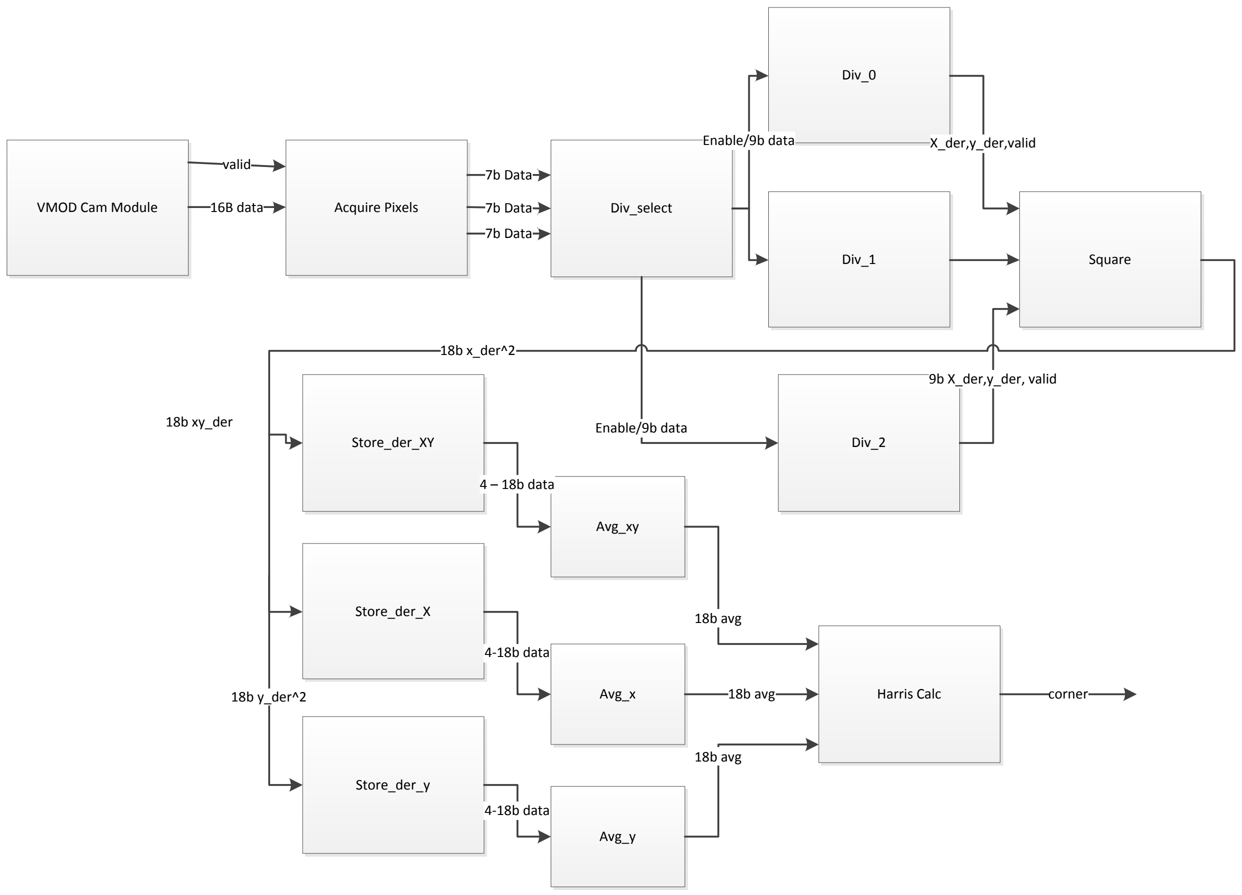


FIGURE 42: TOP-LEVEL BLOCK DIAGRAM WITH APPROPRIATE BITS

This top-level block diagram displays the new bit values for the inputs and outputs of each of the modules in the corner detection process. The first module was responsible for storing the pixel intensities (adding the RGB565 numbers described above) and producing the three 7-bit values necessary for the derivative calculation. The next module was responsible for calculating the differences in these values that is used for the image derivatives. This value was a 9-bit value because one additional bit is used to an overflow and a sign bit. This number was used in the Calculate Derivative module in which it is divided by 6. The derivative calculated was sent to the square module where an 18-bit value was produced. These values were then stored again in order to perform the averaging. Once the averaging was completed (using a 16 pixel averaging box), the values were sent into the Harris Calculator in which it was determined if the current pixel was a corner. If the pixel was a corner, then the coordinates were output and the corner strobe pulses high. The next step in the design was to test using an entire image and compare the output to the output from MATLAB using an identical input image. The test bench was set up to output to a text file whenever a new corner was detected. The output to the text file would be the x and y coordinates of the corner that was detected. These corners were then mapped to an image using a C# program. The outputs of both the MATLAB implementation and the VHDL test implementation can be seen below.

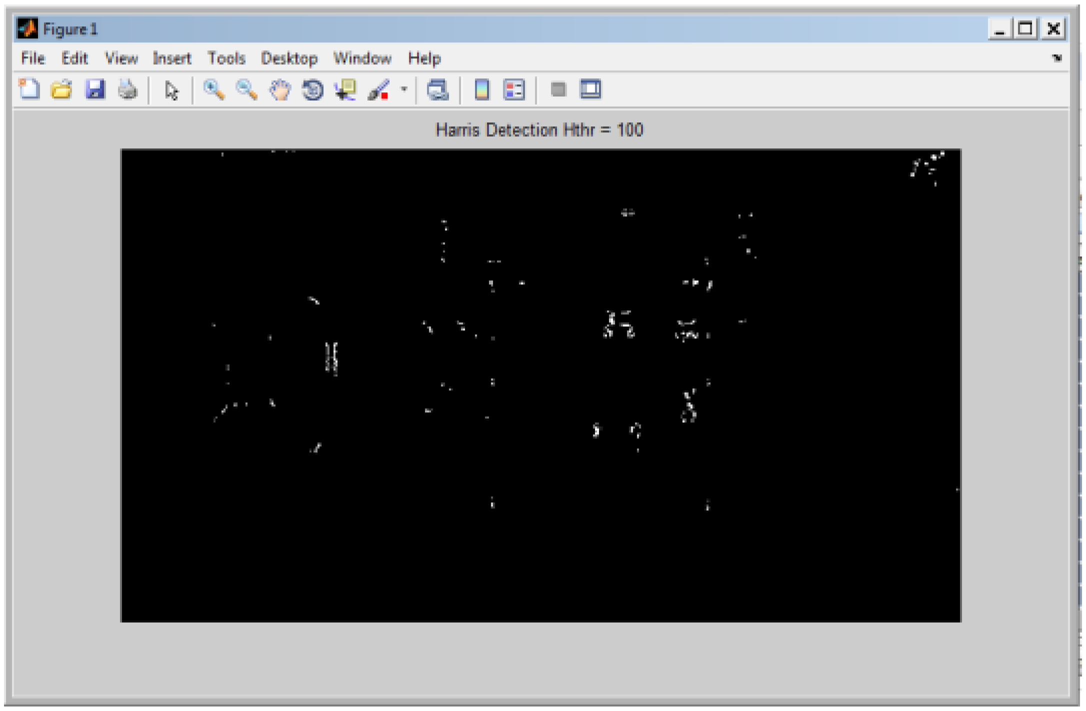


Figure 1
File Edit View Insert Tools Desktop Window Help
Harris Detection Hthr = 100

FIGURE 43: CORNER DETECTION USING HARRIS IN MATLAB

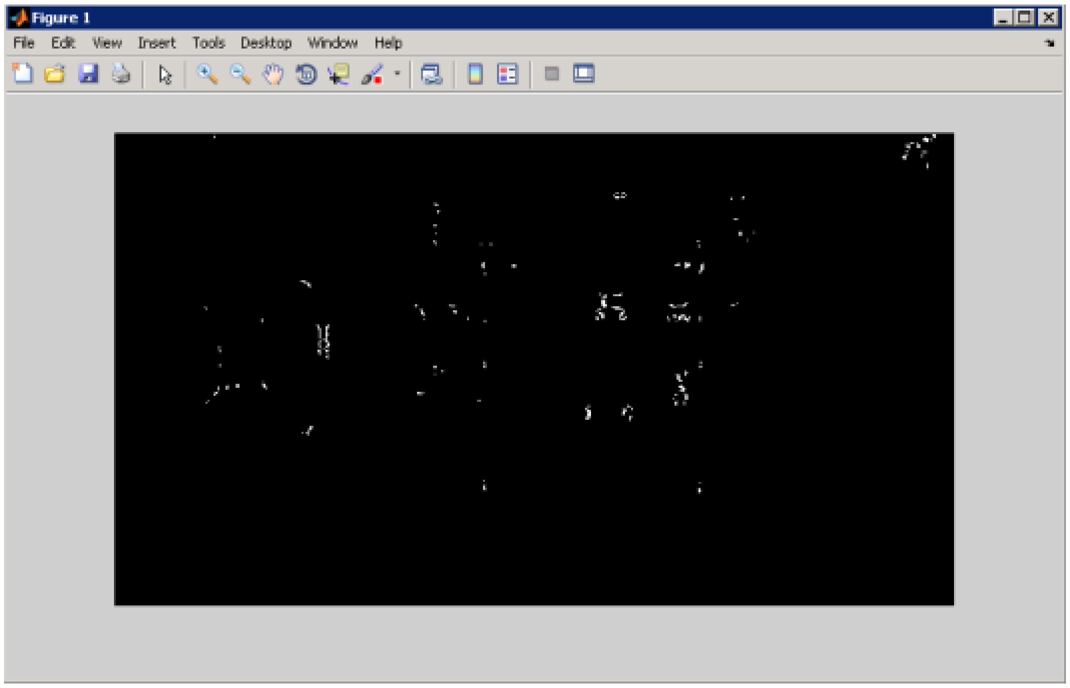


FIGURE 44: CORNER DETECTION USING HARRIS ON FPGA

The MATLAB code detected 583 corners, while the FPGA detected 590 corners. This was most likely due to round off error on the FPGA.

The next step in the design was to implement the UDP communications method to send the corners to the base station.

5.2.3 UDP Implementation

In order to implement the UDP communications module, an external module was used. This module was found online [18] and written to utilize the Ethernet port on the Atlys. The initial implementation with this module allowed for communications between the Atlys board and a computer running Wireshark. Once the initial communication was set up, experimentation began to determine the speed at which the data could be sent. In an attempt to simulate the data transmission from the corner detection module, data was sent every 10 clock cycles. This was chosen because the pipelined output of the corner detection could output two consecutive corners very quickly. In the initial implementation, a simple counter was sent over Ethernet to verify the output. When this output was tested, it was observed that not all of the packets were being received. Because of this problem, it was decided that a FIFO needed to be added to the design in order to send more corners in one packet. The calculation can be seen below:

$$\approx 1000\frac{corners}{frame}x \approx 30\frac{frames}{secnd}x\,1\frac{packet}{corner}\;x\,32\frac{bytes}{packet}\;x\,8\frac{bits}{byte} \approx 7.68\,Mbps$$

A better approach was determined to be to send more than one corner per packet to avoid the 20 bytes of packet overhead per packet. The maximum UDP packet size 1500 bytes. Using the 20 bytes of packet overhead, it was determined that we could send 360 corners per packet. The resulting calculation is seen below:

$$\approx 1000\frac{corners}{frame}x \approx 30\frac{frames}{secnd}x\,1\frac{packet}{360\,corners}\;x\,1500\frac{bytes}{packet}\;x\,8\frac{bits}{byte} \approx 1\,Mbps$$

The calculations above show that increasing the packet size to accommodate more corners would drastically decrease our data rate. In order to implement this in VHDL, a buffer was implemented.

In the Ethernet "Packet_Sender" module, there was a parameter adjusting the packet size. Because we were only sending 32 bits of data, this was not implemented and the data was just sent with the header information. In the packet sender module a loop was implemented to send the rest of the packet. In order to send data to the Ethernet module at every clock edge a FIFO was implemented inside the "Packet_Sender" module. The FIFO filled until a maximum read count was reached. At this point, a state machine advanced states to send the UDP data. The state machine sent data until a counter reached the maximum read count. The maximum read count was specified as the amount of coordinates to be sent in a single packet (360). A block diagram for the process can be seen below.

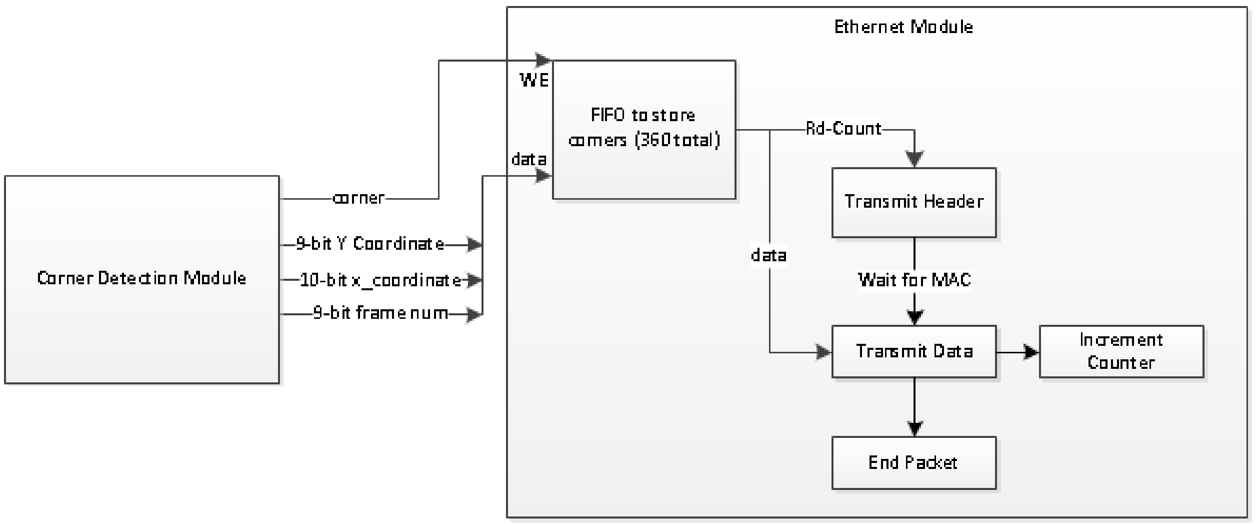


FIGURE 45: BLOCK DIAGRAM FOR PACKET FIFO

The write clock of the FIFO was wired to the system clock of the corner detection module and the read count of the FIFO was wired to the system clock of the "Packet_Sender" module. Because of the difficulty in simulating the Ethernet signals, the Ethernet module of the board was programmed to the board independently of the rest of the design for testing. In order to test, a new module was designed to simulate the corner detection output. This module consisted of sending a "corner" pulse followed by 32 bits of data. When this module was tested, it was determined that the code was working more efficiently than sending a single corner in each packet. A counter was used to determine if all of the data was being received. The data was observed in Wireshark and it was determined all of the counter values were being received. When the testing of this module was completed, the only design component remaining was a program to be written for the base station to store the packets being received.

5.2.4 UDP Receiver

The receiver software was implemented using C#, and consisted of two separate entities. The host computer listened on the UDP port 4660 (statically programmed in to each UDP packet sent by the FPGA). We utilized the UDPClient.Receive() function to block until a UDP datagram was received. The program then saved those received bytes in a text file, and returned to listening for the next packet. This program was essentially an infinite loop and was designed this way in order to not let the processing of each byte interfere with the receiving of packets.

A second program was then run which parsed the text file. It read in the bytes, 4 at a time, and parsed them using a bit mask. The program determined whether or not a new frame was started, and then plotted the received coordinates on that frame. Each frame was saved to the hard drive, in order to view the results.

This multi-process system of obtaining images was not efficient, but did solve quite a few issues we had with packet receive issues, especially at high data rates.

Upon completion, the entire design was ready to be programmed to the board and observed.

5.3 Complete Design Implementation

Once each component was designed and individually tested, the entire design was synthesized together. A portion of the top-level VHDL module used to can be seen in the Appendix. The block diagram of the final design can be seen below.

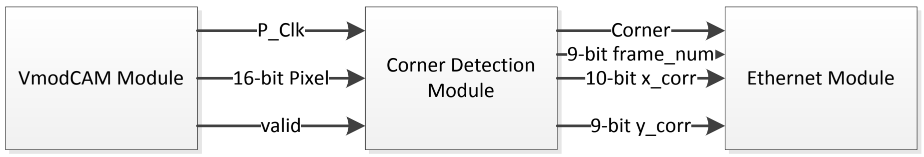


FIGURE 46: TOP-LEVEL BLOCK DIAGRAM

When all of the modules were combined and programmed to the FPGA, a number of problems were observed. The output from the corner detection algorithm did not resemble the expected output. A sample output image can be seen below.

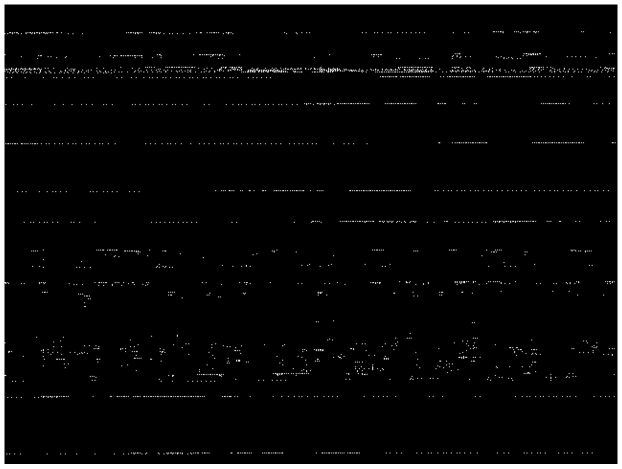


FIGURE 47: INITIAL SYSTEM OUTPUT

As shown in Figure 47 above, the corner detection algorithm did not behave as expected. The lines also appeared to be “scrolling” from frame to frame. In order to gain a better understanding of the signals coming from the camera and other internal signals, the I/O ports of the board were used. These ports were used to examine some signals that were not used in simulation. The first was the “DV_O” signal, which signifies when there is a new pixel acquired. Another signal was the pixel clock. Two additional camera signals that were observed were “FV_I” and “LV_I”, which correspond to frame valid and line valid. The valid signal generated from the first pipeline stage of the corner detection module was also observed. The following figure shows the oscilloscope output from these signals.

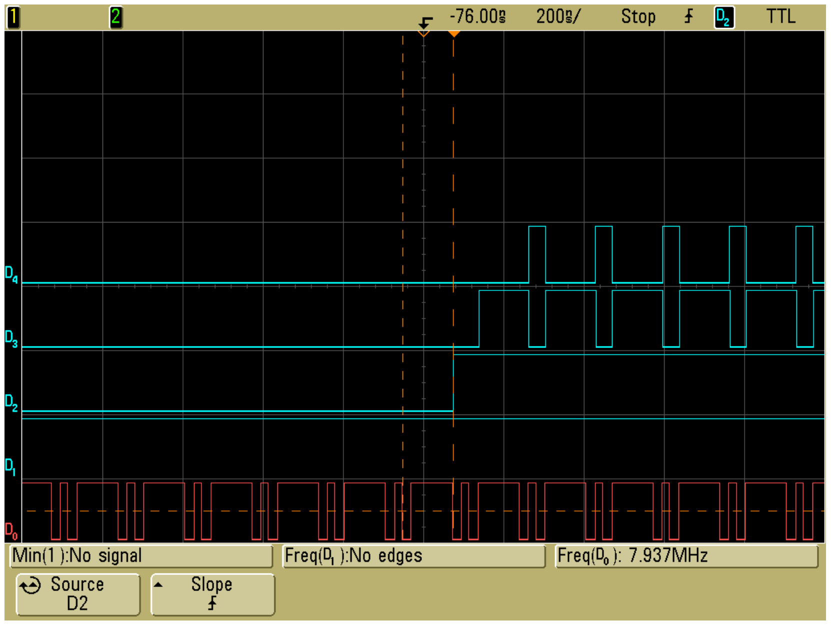


FIGURE 48: OSCILLOSCOPE CAPTURE FROM VMODCAM SIGNALS

In Figure 48 above, the bottom (red) signal represents the pixel clock, the signal above represents the frame valid signal, the signal above represents the line valid signal, and the top signal represents the valid input of the corner detection module. The alarming signal in this module was the pixel clock signal. The pixel clock was not a constant clock. Once this behavior was observed, the data sheet for the camera was consulted to determine if it was the expected behavior for the camera. The timing diagram from the data sheet can be seen below.

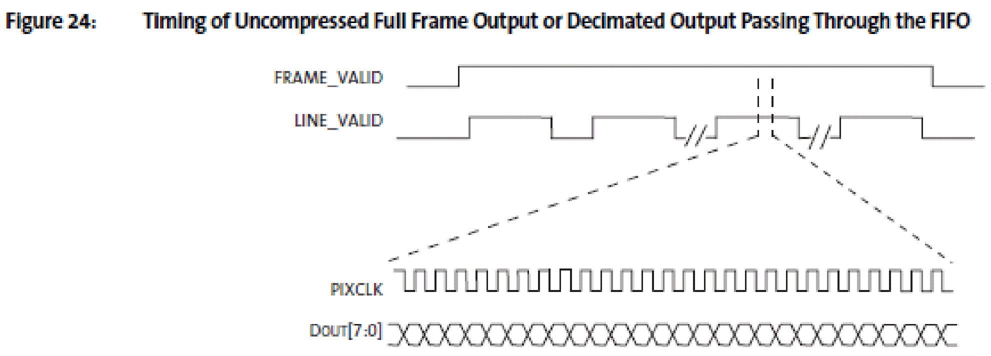


Figure 24: Timing of Uncompressed Full Frame Output or Decimated Output Passing Through the FIFO
FRAME_VALID
LINE_VALID
PIXCLK
DOUT[7:0]

FIGURE 49: DATA SHEET TIMING DIAGRAM [19]

As seen in the figure above, the “PIXCLK” appears constant, unlike the one seen in our scope reading. After some thought was put into results, it was determined that this behavior was due to our application not using the full resolution of the camera. In order to generate a lower resolution, the camera would not output every pixel that was acquired. In order to test if this was the case, the frequency of the valid input to the corner detection was divided by the length of the line valid signal to determine the number of pixels that were being read in per line. When this division was computed, it was determine that the camera was acquiring 640 pixels per line which was exactly what was expected. The same calculation was done with the frequency of the line valid signal and the length of the frame valid signal to determine the number of lines per frame. This calculation again resulted in the expected output (480). Once it was determined that the amount of pixels per frame was correct, the synchronization of the camera and the corner detection was examined.

The major component in the synchronization was the used of the "FV_I" signal. This signal indicates when the pixels from the camera were in a valid frame. The rising edge of the frame valid signal would indicate a when a new frame was starting. This signal was interfaced with the rest of the design setting the reset signal of the module that captures the initial pixels to "NOT FV_I". By setting the reset signal, all of the counters would reset to process a new frame. This ensured that the corner detection algorithm would start processing each frame as they arrived. This modification didn't remove the "scrolling" from the output images as hoped. This led to the determination that the interface with the camera was not exactly behaving as it was simulated in the test bench. In order to observe the camera behavior, the I/O ports on the Atlys were again used. The camera behavior was observed on the oscilloscope. The scope capture below shows the valid input to the corner detection module D_6 (top) and the clock for the corner detection module D_0 (bottom).

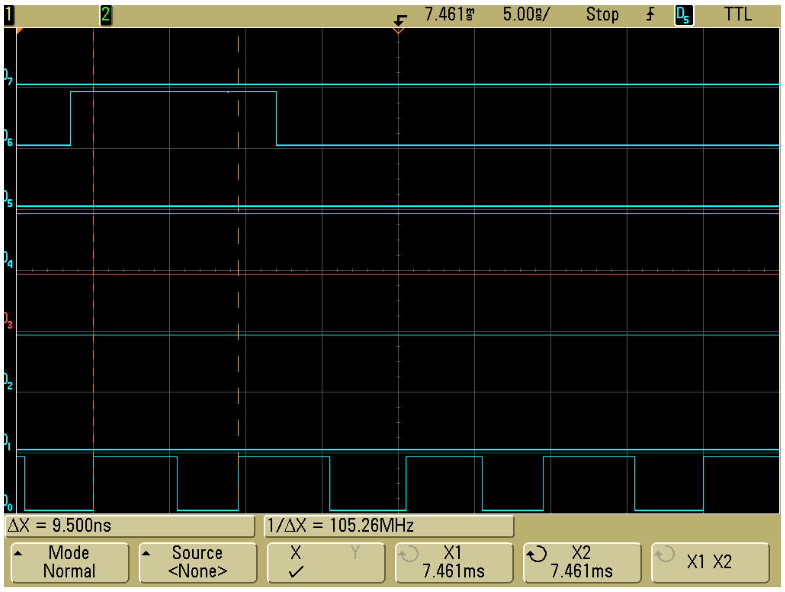


FIGURE 50: OSCILLOSCOPE CAPTURE FOR VALID SIGNALS

It can be observed from the cursors that the valid signal lasted for longer than one clock cycle. This lead to unexpected behavior. If the valid signal was high for too long, the same pixel would be read more than once. This problem was corrected by synchronizing the clock domains. In order to provide synchronization, three flip flops were used. The first flip flop was connected to the valid input of the design. The second flip flop was connected to the output of the first flip flop and the third flip flop was connected to the output of the second flip flop. The new "valid" signal was the output of the second flip flop AND the inverted output of the third flip flop. The block diagram for the synchronization can be seen below.

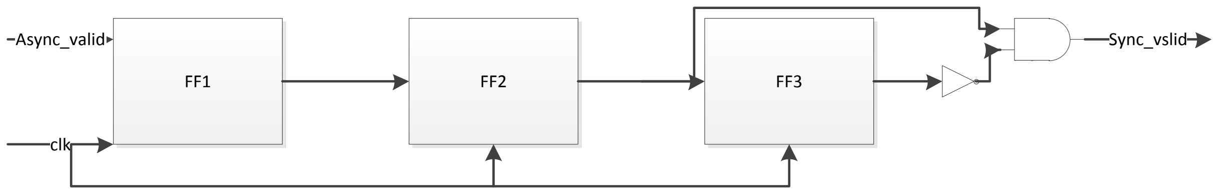


FIGURE 51: SYNCRONIZATION OF VALID SIGNALS BLOCK DIAGRAM

The above process was tested in a test bench to ensure correct operation. The test bench output can be seen below.

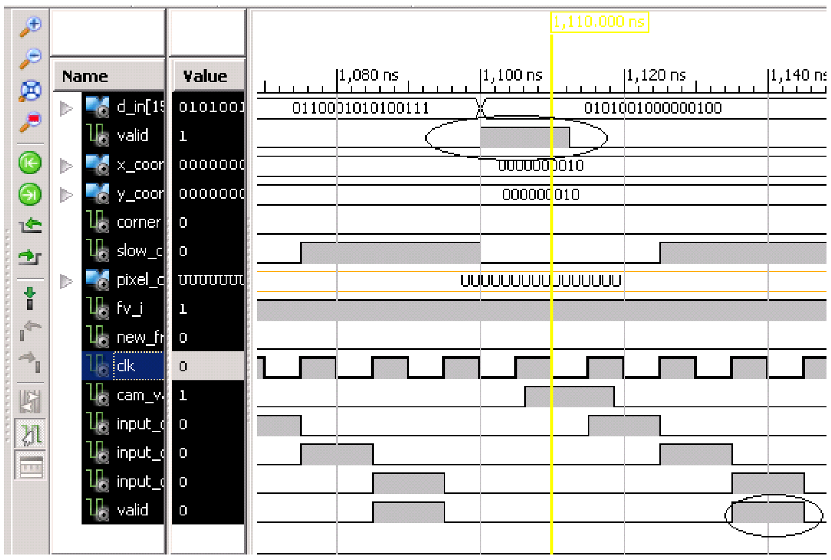


FIGURE 52: VALID SYNCHRONIZATION TEST BENCH

Figure 52 above shows the output of the test bench for a valid signal longer than 1 clock cycle. The upper circle showed that asynchronous valid and the lower circle shows the synchronous valid. The synchronous valid lasted exactly one clock cycle as expected. The last modification to the existing code met the timing of the pixels coming in.

It was observed by measuring the frequency of the valid input signal that the signals were arriving faster than was simulated in the test bench. When the speed of the test bench simulation was increased, it was determined that the design was not operating fast enough to meet the pixel requirements. The test bench below shows the output of the module.

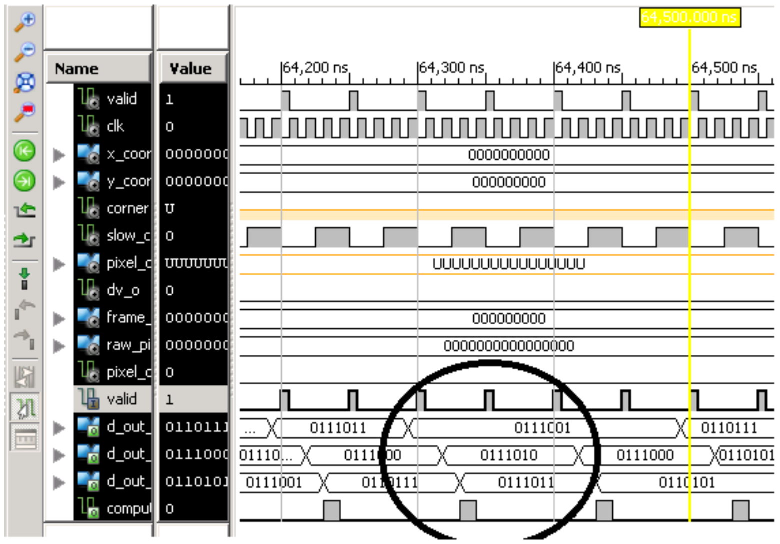


FIGURE 53: TIMING TEST BENCH OUTPUT

As seen in Figure 53 above, the valid inputs were arriving faster than the module could output the data. This occurred because the pixels were stored in 1 Block RAM element specified to be 1920 pixels (three rows). When one pixel was read, two surrounding pixels were also read for derivative calculation. The two other pixels were read by specifying to different addresses and waiting for each read to complete. This process took longer than it took for the pixels to arrive. To correct this problem, three 640 pixel Block RAMs were used instead of one 1920 pixel Block RAM. This enabled all of the Block RAMs to read in parallel and accomplish exactly the same output in one clock cycle. The block diagrams for the two different outputs can be seen in Figure 54 below. The initial implementation is shown on the left and the new implementation is shown on the right.

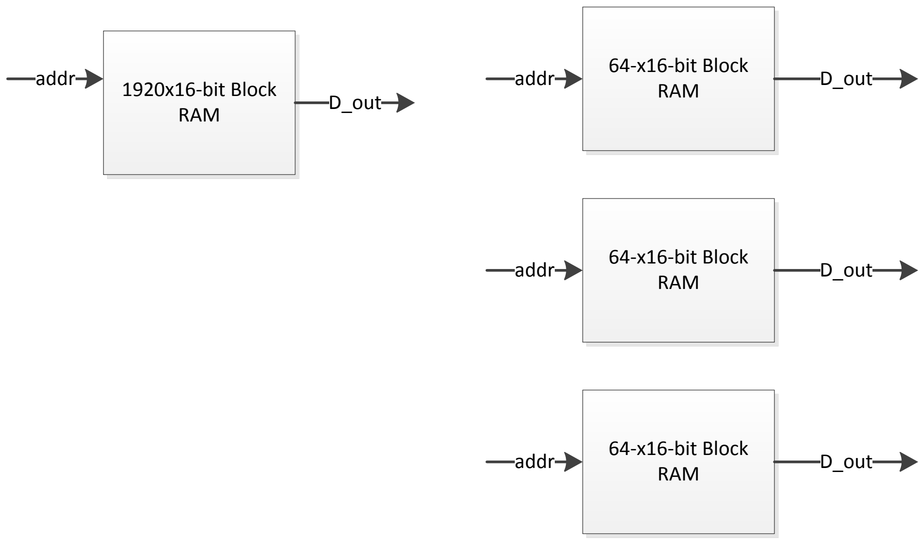


FIGURE 54: IMPLEMENTATIONS OF BLOCK RAMS

As seen in the block diagram above, the new implementation allowed for three pixels to be output at once rather than one pixel at a time. The test bench output for the new implementation can be seen below.

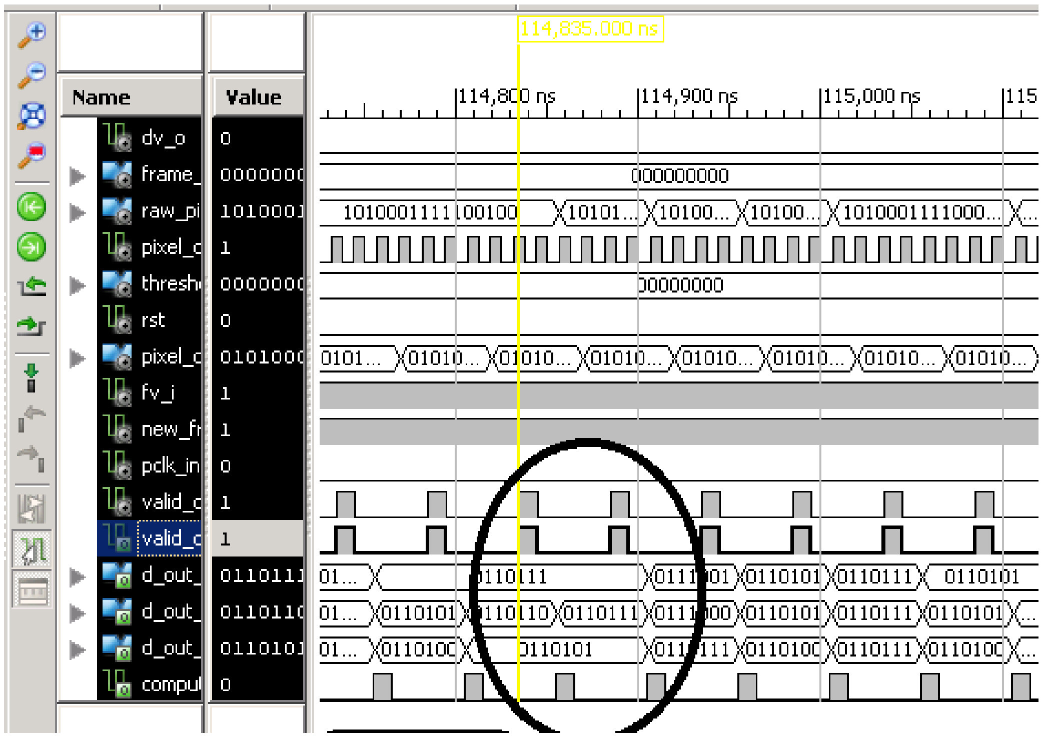


FIGURE 55: MODIFIED RAM IMPLEMENTATION

As seen in Figure 55 above, the new implementation produced one "compute" output to every valid output. This was the expected behavior. Timing hazards occurred in two more places in the pipeline. The second area was the divider module. In order to correct this problem, another divider was added in order to process the data coming in at the appropriate speed. The final location was the module to store the derivatives before the averaging takes place. This module was corrected using the exact same method described above. The module featured a RAM large enough to store four rows of the image. This module was modified to contain 4 one row block RAMs to meet the timing specifications. Once the aforementioned modifications were made, the entire system behaved as designed. Once the final implementation was constructed, the testing of the module began.

CHAPTER 6 VERIFICATION AND TESTING

The previous chapter detailed the process that was taken to implement the complete system. The complete system captures video, performs corner detection on the individual images, and sends coordinates of detected corners to the base station. The base station creates images based on the coordinates of the detected corners and runs EKFMonoSLAM on those images. This chapter contains the information regarding the testing and verification of the complete system. The chapter is broken down into two sections: still capture testing and scenario based testing. The still capture testing was performed by holding the camera still and verifying the output. The scenario testing consisted of two scenarios: walking in a straight line down the hallway and walking in a straight line down the hallway and making a 90-degree right hand turn. The corner-only images gathered from the scenario testing were processed using EKFMonoSLAM to test the output of the entire system.

6.1 STILL CAPTURE TESTING

The still capture testing consisted of pointing the camera at certain locations inside the Precision Personnel Locator (PPL) lab. In the first still capture test, the camera was pointed at the location shown in Figure 56 below.

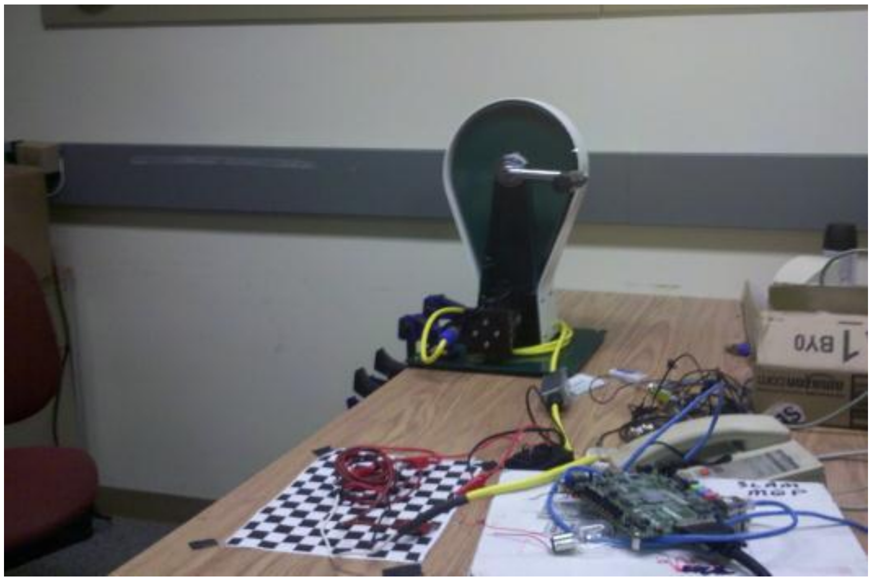


FIGURE 56: TEST LOCATION 1 FOR STILL IMAGE CAPTURE

After the camera has held in the same position for a short period of time (about 5 seconds) and output was captured. After the packets were sent to the base station and corner-only images were created, Figure 57 was observed.

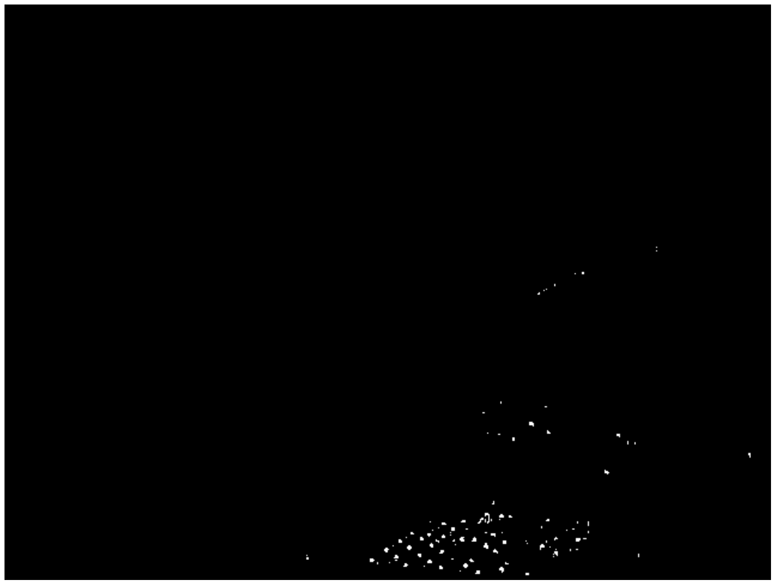


FIGURE 57: CORNER OUTPUT FOR STILL CAPTURE TESTING

As displayed above, the output from the corner detection looks to resemble the test image. A checkerboard image can be seen in the lower portion of the image. This resulted from a paper containing a checkerboard image lying on a desk. The checkerboard pattern of the paper of the original image can be seen in the corner detected image. Based on the appearance of the checkerboard pattern on the corner detected image, the next logical test was to point the camera directly at the paper with the checkerboard and verify the output. The results of the corner detection algorithm when the camera was pointed directly at the checkerboard pattern can be seen in Figure 58 below.

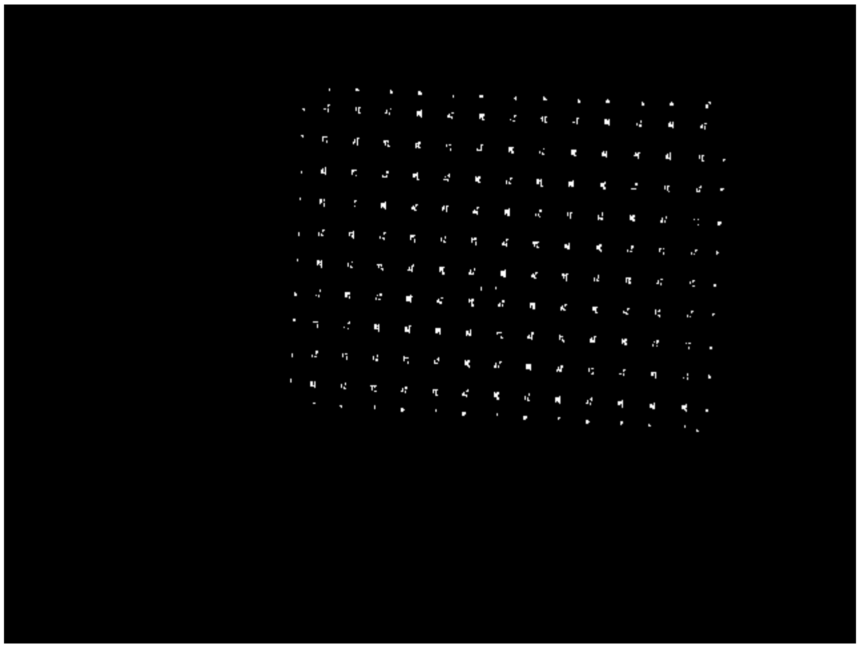
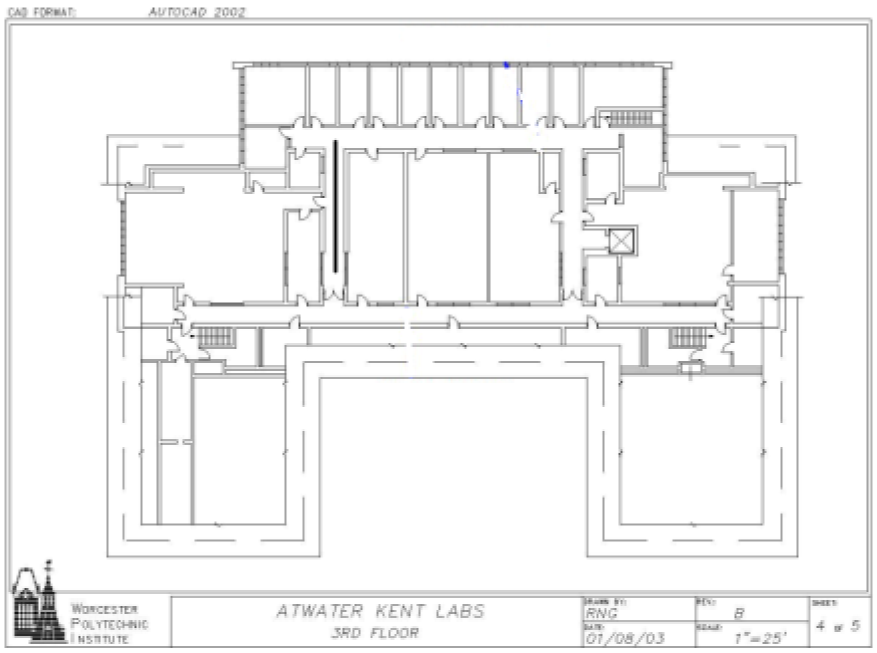


FIGURE 58: CORNER DETECTED CHECKERBOARD IMAGE

The corner detected output from the checkerboard image displays accurate results. The output from the corner detection displays accurate enough results to proceed with scenario testing; the scenario testing consisted of two tests. The first test is walking in a straight line and the second is taking a 90-degree turn.

6.2 Scenario Testing

As described above, the scenario testing consists of two tests. The walks performed for the two tests can be seen below.



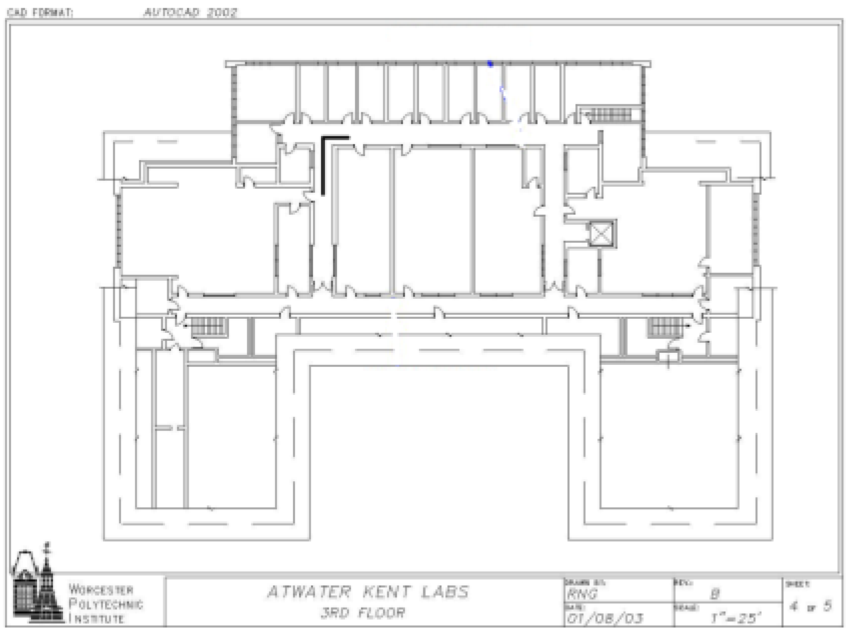


FIGURE 59: SCENARIO TEST PATHS

The two scenario tests paths were walked and corner detected images were captured and run through EKFMonoSLAM. A sample corner detected image from the hallway can be seen below.

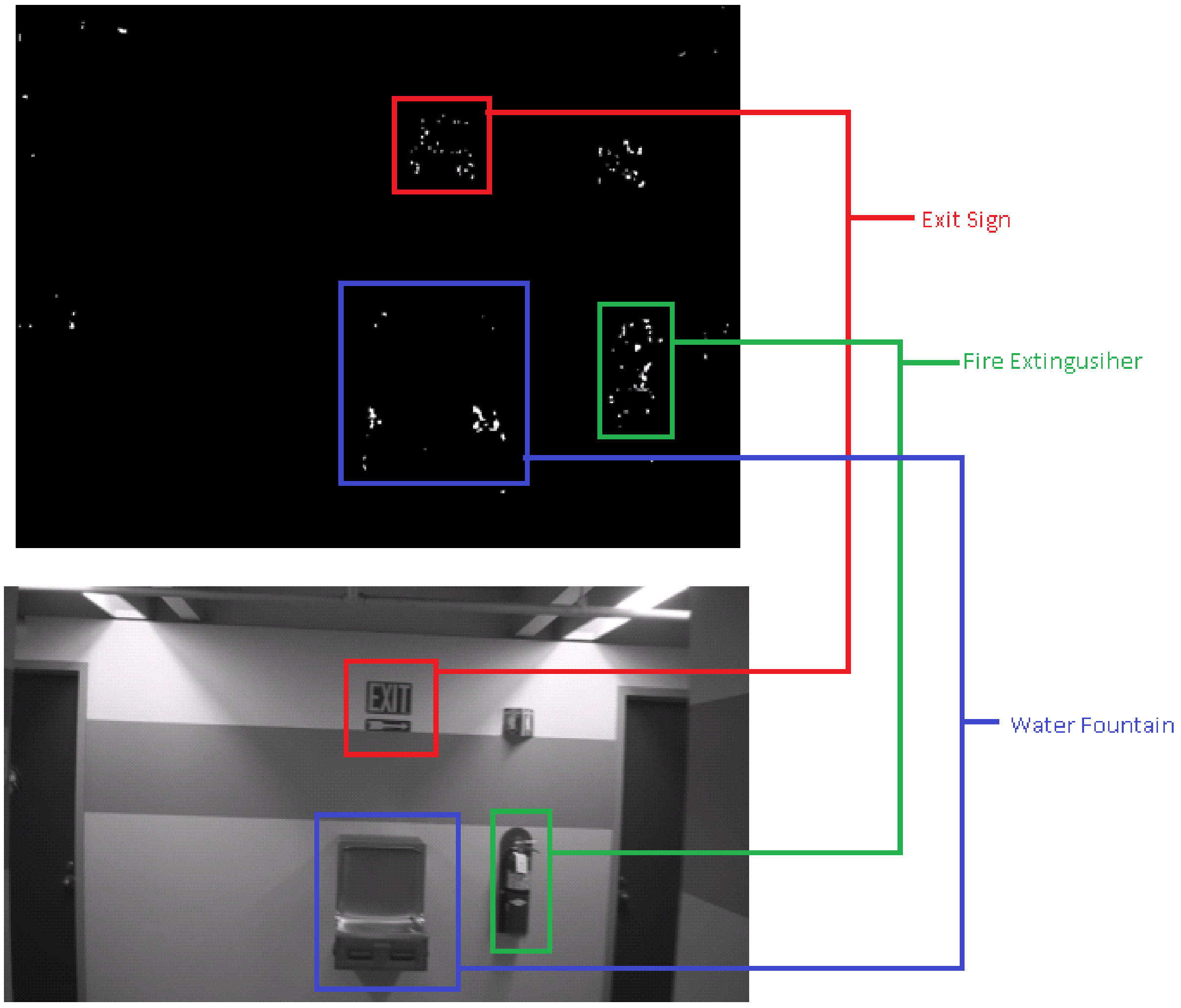


FIGURE 60: IMAGE COMPARISON FROM HALLWAY

Both images above were taken in the same hallway of Atwater Kent as and the same features can be seen in the two images. The test down the hallway was duplicated a number of times, experimenting with different thresholds and camera positions. In some of the tests, the camera was placed on a cart. In other tests, the camera was held by one of the group members. The camera was also taped to a chair with wheels in some of the tests. The chair with wheels testing apparatus seemed to work the best because it both held the camera still and was easier to push straight than the cart. A picture of our test apparatus can be seen below.

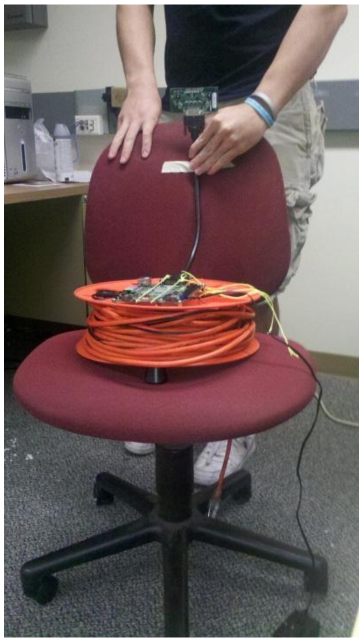


FIGURE 61: TEST APPARATUS IMAGE

Once all of the test sequences were run through EKFMonoSLAM, some promising results were observed. The first test we attempted was a straight-line test. This test was run by walking down the hallway as straight as possible for approximately 20 feet.

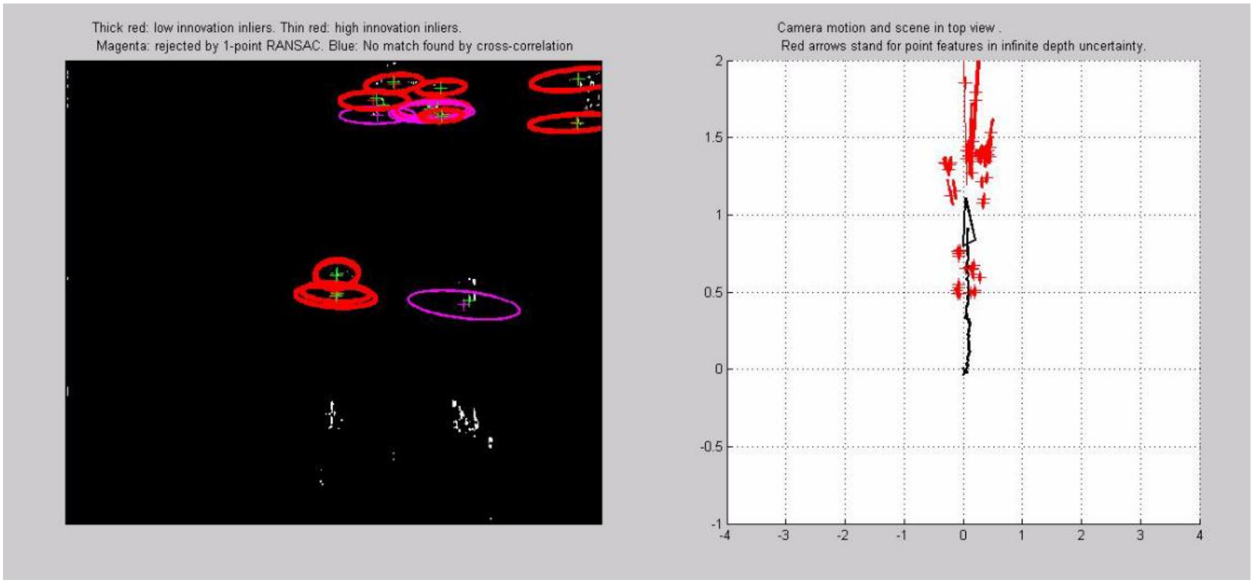

Thick red: low innovation inliers. Thin red: high innovation inliers.
Magenta: rejected by 1-point RANSAC. Blue: No match found by cross-correlation
Camera motion and scene in top view .
Red arrows stand for point features in infinite depth uncertainty.
2
1.5
1
0.5
0
-0.5
-1
-4
-3
-2
-1
0
1
2
3
4

FIGURE 62: FINAL IMAGE OUTPUT FOR STRAIGHT LINE TEST

As seen above, the path displayed by EKFMonoSLAM is nearly identical to that shown in Figure 59.

We also ran a scenario where we walked straight down the hallway and subsequently turned 90 degrees to the right, and walked another few feet. The image shown below is the last figure output from EKFMonoSLAM for the 90 degree turn test.

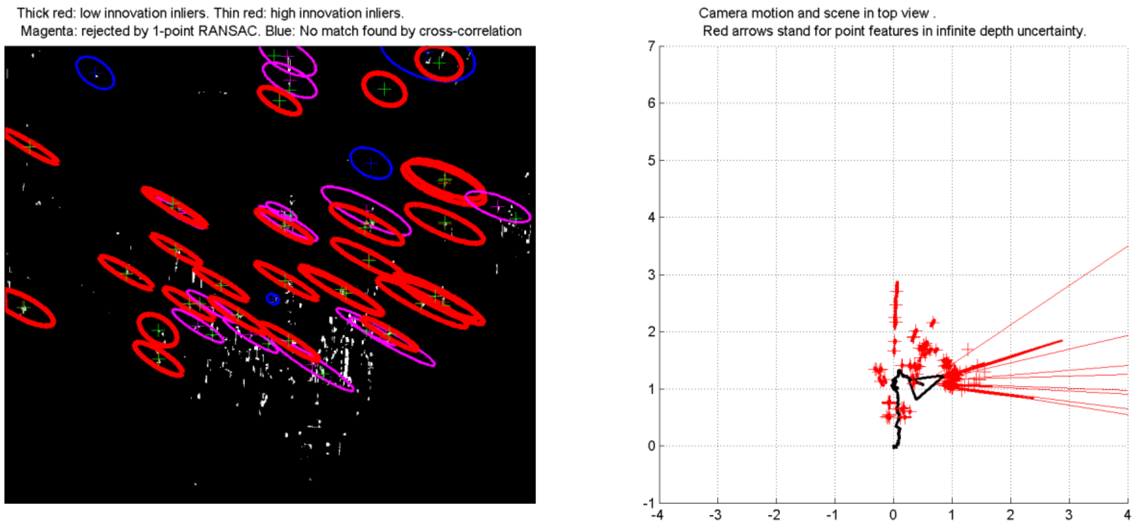

Thick red: low innovation inliers. Thin red: high innovation inliers.
Magenta: rejected by 1-point RANSAC. Blue: No match found by cross-correlation
Camera motion and scene in top view .
Red arrows stand for point features in infinite depth uncertainty.
7
6
5
4
3
2
1
0
-1
-4
-3
-2
-1
0
1
2
3
4

FIGURE 63: FINAL OUTPUT FOR 90-DEGREE TEST

As seen in Figure 63 above, the final output of the 90-degree test seems to be consistent with the path that was walked, as plotted in Figure 59.

Overall, our results were very accurate. We realize that our tests were somewhat simple; however, these tests prove the feasibility of using external corner detection and EKFMonoSLAM to track a user's position. The next chapter describes the overall conclusions made regarding the project.

CHAPTER 7 CONCLUSION

This project provided a way of performing indoor navigation and tracking for first responders using SLAM. Upon completion of this project, all of the goals set forth in were successfully accomplished. Table 4 below shows the project goals and the implementation used to complete the goals.

Project Goal	**Implementation**
Capture and process images in real time	VmodCAM Stereo Camera Module with FPGA Processing
Send Resulting Data to Base Station	Ethernet Module on FPGA with base station receiver
Develop method to provide SLAM algorithm with input	Corner Detection on FPGA to base station receiver with black and white images
Configure SLAM algorithm to accurately track motion using corner-only input	Changed settings in EKFMonoSLAM to reflect differences with corner-only input

TABLE 4: DESIGN GOALS AND IMPLEMENTATION

The successful completion of the goals was verified by performing two scenario tests: walking in a straight line and walking in a straight line and performing a 90-degree turn. By accomplishing all of the project goals, the constraints for the project were also met. The VmodCAM and Atlys FPGA provide the firefighter with a light-weight portable unit that would not hinder and movement. Although our design was not implemented using a battery powered device, it would be possible to run the FPGA and camera on a battery. The complete design featured a data transmission rate of around 1 Mbps which is achievable using WiFi. The UDP implementation also allows for easy Wi-Fi expansion if desired.

7.1 SIGNIFICANCE

The major problem with the SLAM algorithm was the processing time required to provide the base station with output. This project helped to cut down on the processing time by performing feature detection remotely on the FPGA. This brings the SLAM algorithm one step closer to becoming a system that can be implemented and provide faster output to the base station regarding the firefighter's location.

The successful output from the scenario testing provides a basis for using SLAM for indoor firefighter location. The SLAM indoor location method may remove or lessen some of the challenges presented in previous indoor localization methods such as RF or inertial based tracking. SLAM does not need to be used independently, but rather it can be used in conjunction with one or more of the existing technologies to possibly provide a more accurate location.

7.2 CHALLENGES IN IMPLEMENTATION

The goals set forth were accomplished, but with some difficulty in implementation. One of the problems in accomplishing the goal to capture data and process images in real time was interfacing with the VmodCAM module. Although Digilent provided a sample code, the implementation still remained difficult. The datasheet for the VmodCAM module was not very informative and did not give many specifics about the camera. For instance, the Digilent module set up the camera in "Context A". Upon reading the data sheet, "Context A" was said to be a "preview" mode for the camera. The data sheet also specified a "Context B" that could be set up for both "video capture" and "snapshot" mode. The datasheet did not distinguish between the modes. After some experimentation, it was determined that "Context B" could not be configured perform the image capturing at a low enough resolution for our application (640x480) so "Context A" was used.

Problems were also faced in implementing the UDP receiver. The main problem in the UDP receiver implementation was the data was coming in faster than the UDP receiver could receive and process the data. The original implementation was to draw the black and white images as they were being received (i.e. every time the frame number changed), but it was quickly determined that too many packets were being missed. Instead, it was decided that the packets would be dumped into a .txt file and processed into images once all of the data was received. This problem was somewhat expected due to the inherent nature of the UDP implementation.

The rest of the problems in implementation had to do with timing synchronization. The camera was running on a different clock than the corner detection which was running on a different clock than the Ethernet module. The timing was synchronized between the camera and corner detection by using the flip-flop method described in Chapter 5. The Ethernet module was synchronized to the corner detection by using a FIFO with independent read and write clocks.

Perhaps the most difficult problem that was faced in the design implementation was getting correct output from EKFMonoSLAM using the corner-only images. Since EKFMonoSLAM was designed to work with full images, certain parameters had to be tweaked inside EKFMonoSLAM to get the correct output. The modification of these parameters required a long time to test because of the simulation time associated with EKFMonoSLAM. Although many problems were faced in implementing our design, all of the complications were overcome and the design was completed.

Based on the results obtained from testing, it was determined that this project provides a viable method for indoor navigation and tracking. The SLAM method provides an alternative way of indoor navigation and tracking that may avoid some of the problems that have been associated with the previous methods described above. The implementation of this project provided accurate tracking for two scenario tests: walking in a straight line and performing a 90-degree turn. This algorithm can be expanded to provide tracking for more comprehensive testing.

Also, all of the design goals set forth for this project were completed. We were able to successfully capture and process images on an FPGA. We were able to set up a communications link between the FPGA and a laptop running SLAM software. Additionally, we were able to modify the existing SLAM software to use the processed images from the FPGA. Finally, we were able to provide accurate results for the two scenario tests that were performed.

7.3 Suggestions for Further Development

Although the goals set forth for this project were completed, there are also areas for further improvement and future research. The first area of improvement would be to implement a stereo SLAM algorithm that uses both of the cameras contained on the VmodCAM module. Stereo SLAM would provide a different approach to the SLAM algorithm and give different and perhaps more accurate results with two sources of data. Another approach might be to use a thermal camera instead of a traditional camera to enable the system to be used in smoke-filled environments.

Another area for improvement would be to modify the division module of the corner detection pipeline to allow for faster frame rates. The modification of this module would be to implement a RAM lookup table for the division. This lookup table could be implemented because the algorithm is dividing by 6 every time. The maximum value that could be the input to the division module is 256. This allows for about 40 possibilities for a quotient. The input to the module could be specified as the address to a pre-initialized block RAM component and the output data of the RAM module would be the quotient. The sign (most significant) bit of the input data would be concatenated to the output of the RAM to form the output of the module. This implementation would reduce the amount of clock cycles for the division from about 20 to 1.

Another improvement would be to improve the EKFMonoSLAM implementation. The EKFMonoSLAM module processing time was a major bottleneck of the project. The amount of time required to verify simulations made it impossible to make quick changes and test the difference. EKFMonoSLAM also does not allow for real-time results of the corner detection algorithm. Although corners are tracked from image to image, EKFMonoSLAM is responsible for creating a map and track based on the corners. If this process takes too long to complete, real-time tracking can never be achieved.

For further development, more of the EKFMonoSLAM algorithm could be moved to the FPGA. Perhaps, some of the corner correlation could be done on the FPGA because of its parallelizability. This would make EKFMonoSLAM faster and provide closer to real-time output.

References

[1] Braddee, Richard . "Fire Fighter Fatality Investigation Report F99-47 | CDC/NIOSH." CDC - The National Institute for Occupational Safety and Health (NIOSH). Centers for Disease Control and Prevention, 27 SEP 2000. Web. 20 Mar 2012. <http://www.cdc.gov/niosh/fire/reports/face9947.html>.

[2] University, Drexel. *Simultaneous Localization and Mapping (SLAM).* http://prism2.mem.drexel.edu/~billgreen/slam/lecture01/slam_lecture01.pdf (accessed April 30, 2011)

[3] Ribeiro, Maria Isabel. *Kalman and Extended Kalman Filters: Concept Derivation and Properties.* February 2004. http://users.isr.ist.utl.pt/~mir/pub/kalman.pdf (accessed April 28, 2011).

[4] Montemerlo, Michael. *FastSLAM: A Factored Solution to the Simultaneous Localization and Mapping Problem With Unknown Data Association.* Pittsburgh, PA , June 18, 2003.

[5] *OpenCV 2.1 C++ Reference.* 2010. http://opencv.willowgarage.com/documentation/cpp/index.html (accessed April 30, 2011).

[6] Javier Civera, Oscar G. Grasa, Andrew J. Davison, J. M. M. Montiel, 1-Point RANSAC for EKF Filtering: Application to Real-Time Structure from Motion and Visual dometry, to appear in Journal of Field Robotics, October 2010.

[7] Csetverikov, Dmitrij. *"Basic Algorithms for Digital Image Analysis: acourse"* Institute of Informatics Budapest, Hungary. http://ssip2003.info.uvt.ro/lectures/chetverikov/corner_detection.pdf (accessed October 5, 2011)

[8] Digilent Inc,. *Digilent Nexys2 Board Reference Manual.* July 11, 2011. http://digilentinc.com/Data/Products/NEXYS2/Nexys2_rm.pdf (accessed September 25, 2011)

[9] Digilent Inc,. *Digilent Video Decoder Board (VDEC1) Reference Manual.* April 12, 2005. http://www.digilentinc.com/Data/Products/VDEC1/VDEC1-rm.pdf (accessed September 28, 2011)

[10] Analog Devices. *Multiformat SDTV Video Decoder* 2005. http://www.analog.com/static/imported-files/data_sheets/ADV7183B.pdfpdf (accessed September 28, 2011)

[11] Digilent Inc,. *Atlys™ Board Reference Manual.* May 2, 2011. http://digilentinc.com/Data/Products/ATLYS/Atlys_rm.pdf (accessed September 30, 2011)

[12] Digilent Inc,. *VmodCAM™ Reference Manual.* July 19, 2011. http://digilentinc.com/Data/Products/VMOD-CAM/VmodCAM_rm.pdf (accessed September 30, 2011)

[13] Digilent Inc,. *PmodWiFi™ Reference Manual.* January 19, 2011. http://digilentinc.com/Data/Products/PMOD-WIFI/PmodWiFi_rm.pdf (accessed October 3, 2011)

[14] Digilent Inc,. *PmodRF1™ Reference Manual.* December 18, 2009. http://digilentinc.com/Data/Documents/Product%20Documentation/PmodRF1_rm.pdf (accessed October 12, 2011)

[15] Tammy Noergaard . UDP disgram. 2010. Graphic. Electrical engineering, electronic engineering times, ee times, news, analysis, electronic design, products, education, learning ,eet,part search,tech papers, demos,fundamental courses,product search,components,part number | Electronics industry news, electronics design ideas & products for EEsWeb. 28 Mar 2012. <http://www.eetimes.com/design/embedded-internet-design/4019868/Guide-to-Embedded-Systems-Architecture--Part-3-Transport-layer-UDP-and-embedded-Java-and-networking-middleware-examples>.

[16] The IP datagram header format . N.d. Graphic. Netanya Academic College - Students Linux ServerWeb. 28 Mar 2012. <http://mars.netanya.ac.il/~unesco/cdrom/booklet/HTML/NETWORKING/node020.html>.

[17] Gyorgy, Elod (2011) VmodCAM Reference Design 1 [Computer program]. Available at http://www.digilentinc.com/Products/Detail.cfm?NavPath=2,648,931&Prod=VMOD-CAM (accessed October 20 2011)

[18] Williams, Joel (2011) Simple test framework for the Atlys' 88E1111 chip [Computer Program]. Available at http://tristesse.org/FPGA/DigilentAtlysResources (accessed February 9 2012)

[19] Aptina Imaging Coorporation. *MT9D112: 1/4-Inch 2Mp SOC Digital Image Sensor.* May 3, 2011. http://www.aptina.com/support/documentation.jsp?t=0&q=116&x=28&y=11# (accessed March 15, 2012)

APPENDIX

APPENDIX A – ALTERNATE CORNER DETECTION ALGORITHMS

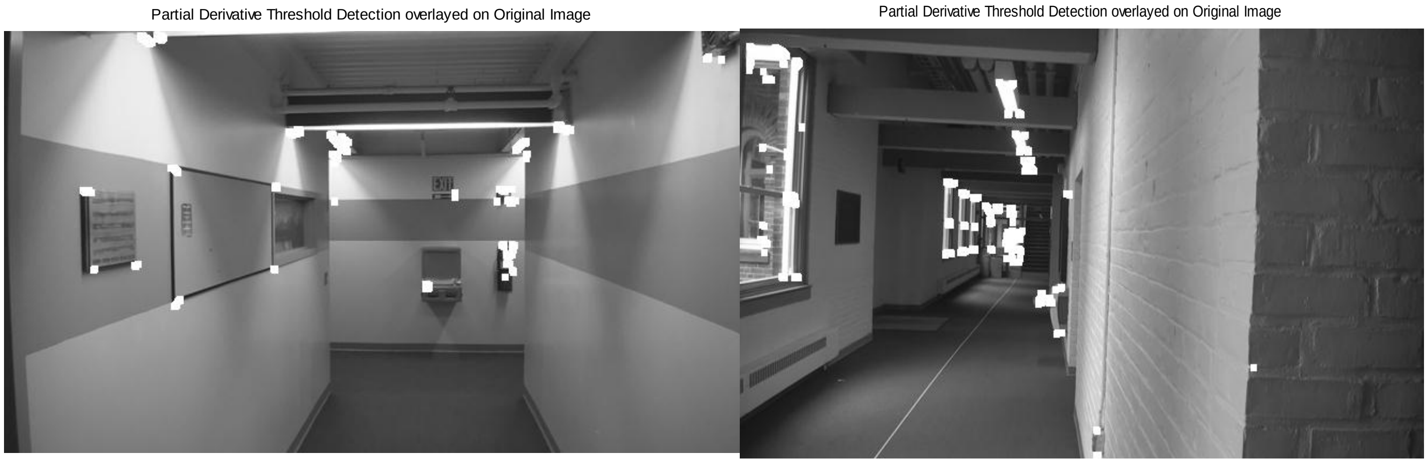

Partial Derivative Threshold Detection overlayed on Original Image
Partial Derivative Threshold Detection overlayed on Original Image

FIGURE 64: USING FIRST DERIVATIVES AND THRESHOLDS

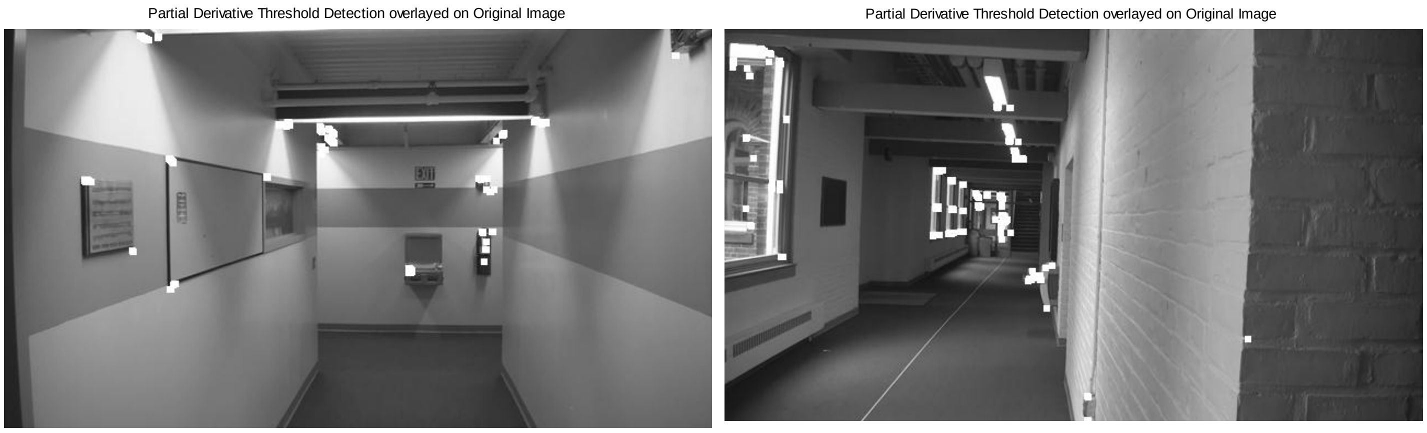

Partial Derivative Threshold Detection overlayed on Original Image
Partial Derivative Threshold Detection overlayed on Original Image

FIGURE 65: USING SECOND DERIVATIVES

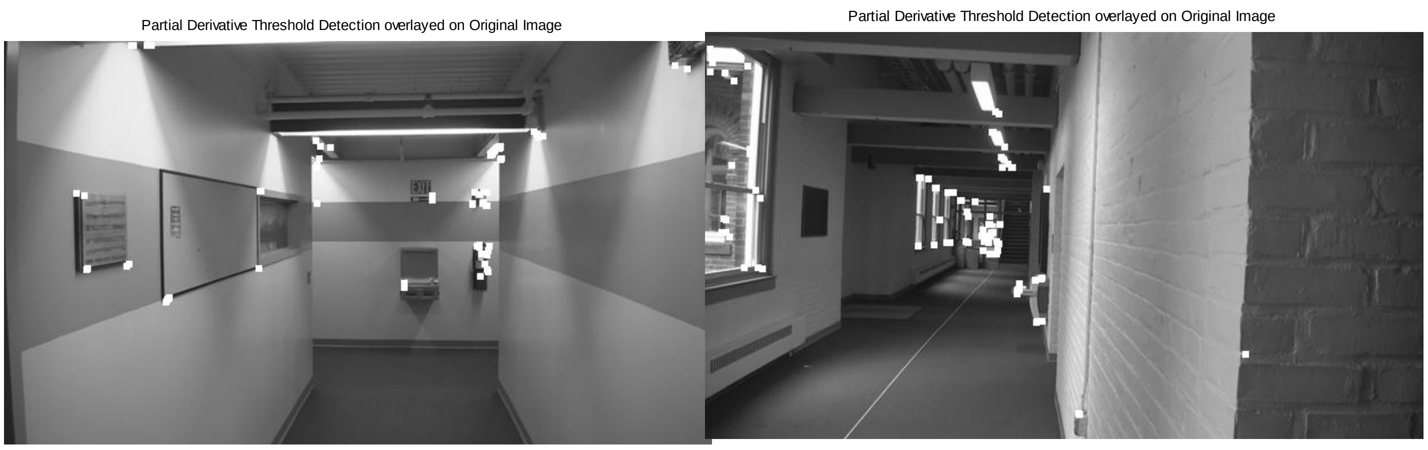

Partial Derivative Threshold Detection overlayed on Original Image
Partial Derivative Threshold Detection overlayed on Original Image

FIGURE 66: DERIVATIVE NEIGHBORHOOD APPROACH

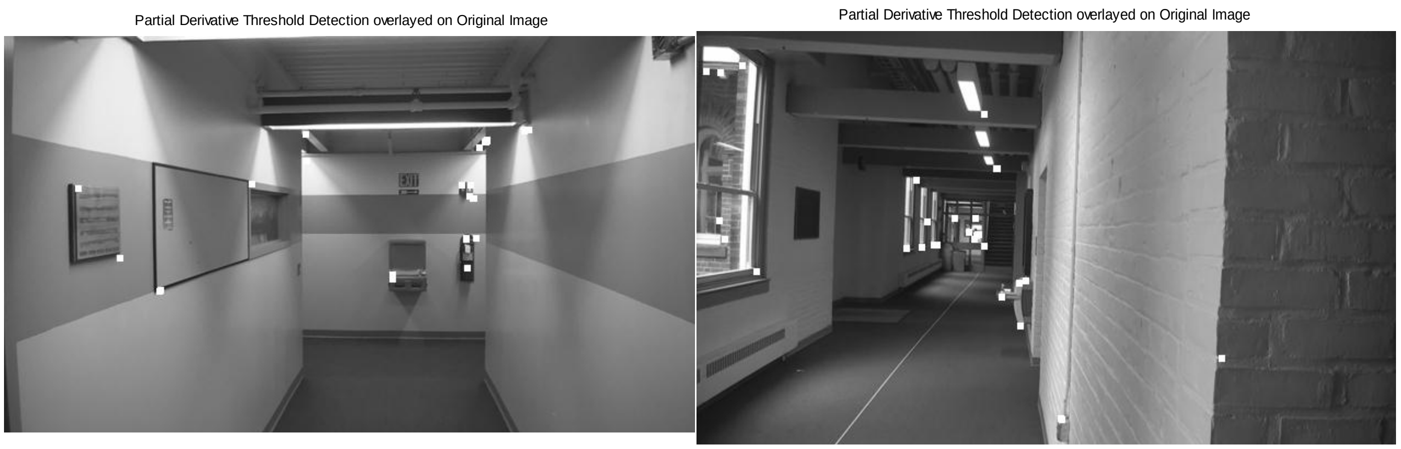

Partial Derivative Threshold Detection overlayed on Original Image
Partial Derivative Threshold Detection overlayed on Original Image

FIGURE 67: RESULTS FOR COMBINATION OF APPROACHES

Appendix B-Differences Between 9 and 16 Pixel Averaging

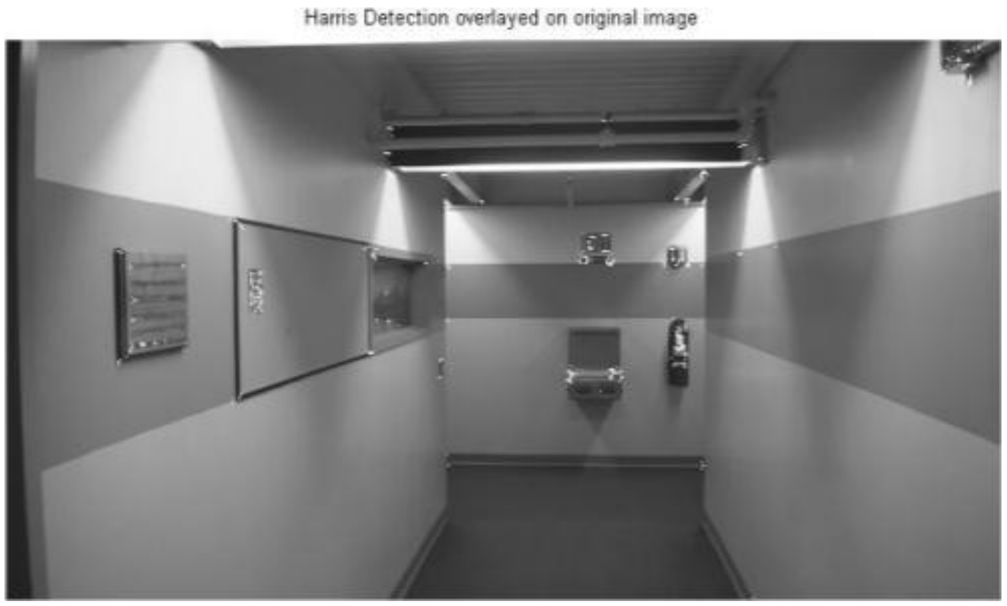


FIGURE 68: HARRIS OUTPUT USING 9 PIXEL AVERAGING BOX

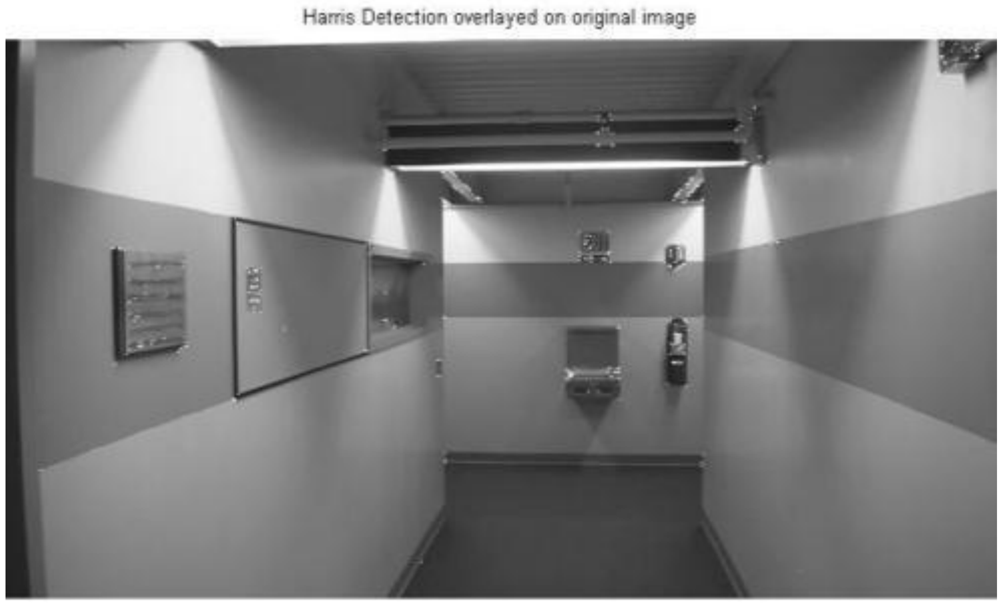


FIGURE 69: HARRIS OUTPUT USING 16 PIXEL AVERAGING BOX

APPENDIX C- TOP-LEVEL VHDL CODE

```
----------------------------------------------------------------------------
-----
-- Company:
-- Engineer: Nick Long Matt Zubiel
--
-- Create Date:    17:34:03 02/14/2012
-- Design Name:
-- Module Name:    top_level - Behavioral
-- Project Name:   Firefighter Indoor Navigation using
-- Distributed SLAM
-- Target Devices:
-- Tool versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments: This module combines the corner
--    detection, image capture and ethernet modules
--
----------------------------------------------------------------------------
-----
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
library UNISIM;
use UNISIM.VComponents.all;

entity top_level is
   Port (
            SW_I : in STD_LOGIC_VECTOR(7 downto 0);
            CLK_I : in  STD_LOGIC;
            RESET_I : in  STD_LOGIC;
            LED_O : out STD_LOGIC_VECTOR(7 downto 0);
----------------------------------------------------------------------------
-----
-- Camera Board signals
----------------------------------------------------------------------------
-----
            CAMA_SDA : inout  STD_LOGIC;
            CAMA_SCL : inout  STD_LOGIC;
            CAMA_D_I : inout  STD_LOGIC_VECTOR (7 downto 0); -- inout
Workaround for IN_TERM bug AR#      40818
            CAMA_PCLK_I : inout  STD_LOGIC; -- inout Workaround for IN_TERM
bug AR#     40818
            CAMA_MCLK_O : out STD_LOGIC;
            CAMA_LV_I : inout STD_LOGIC; -- inout Workaround for IN_TERM bug
AR#   40818
```

```
            CAMA_FV_I : inout STD_LOGIC; -- inout Workaround for IN_TERM bug
AR#   40818
            CAMA_RST_O : out STD_LOGIC; --Reset active LOW
            CAMA_PWDN_O : out STD_LOGIC; --Power-down active HIGH

            CAMX_VDDEN_O : out STD_LOGIC; -- common power supply enable (can
do power cycle)

            CAMB_D_I : inout  STD_LOGIC_VECTOR (7 downto 0); -- inout
Workaround for IN_TERM bug AR#      40818
            CAMB_SDA : inout  STD_LOGIC;
            CAMB_SCL : inout  STD_LOGIC;
            CAMB_PCLK_I : inout  STD_LOGIC; -- inout Workaround for IN_TERM
bug AR#     40818
            CAMB_MCLK_O : out STD_LOGIC;
            CAMB_LV_I : inout STD_LOGIC; -- inout Workaround for IN_TERM bug
AR#   40818
            CAMB_FV_I : inout STD_LOGIC; -- inout Workaround for IN_TERM bug
AR#   40818
            CAMB_RST_O : out STD_LOGIC; --Reset active LOW
            CAMB_PWDN_O : out STD_LOGIC; --Power-down active HIGH

-----------------------------------------------------------------------------
-----
-- Ethernet signals
-----------------------------------------------------------------------------
-----
            PhyResetOut_pin : out STD_LOGIC;
            MII_TX_CLK_pin : in STD_LOGIC; --25 MHz clock for 100 Mbps - not
used here
            GMII_TXD_pin : out STD_LOGIC_VECTOR(7 downto 0);
            GMII_TX_EN_pin : out STD_LOGIC;
            GMII_TX_ER_pin : out STD_LOGIC;
            GMII_TX_CLK_pin : out STD_LOGIC;
            GMII_RXD_pin : in STD_LOGIC_VECTOR(7 downto 0);
            GMII_RX_DV_pin : in STD_LOGIC;
            GMII_RX_ER_pin : in STD_LOGIC;
            GMII_RX_CLK_pin : in STD_LOGIC;
            MDC_pin : out STD_LOGIC;
            MDIO_pin : inout STD_LOGIC;

            btn : in STD_LOGIC_VECTOR(4 downto 0);
-----------------------------------------------------------------------------
-----
-- Debug signals
-----------------------------------------------------------------------------
-----
      I_O : out STD_LOGIC_VECTOR(7 downto 0)


      --    rs232_tx : out STD_LOGIC
      );
end top_level;



architecture Behavioral of top_level is
```

```
Component ethernet_test_top IS
      PORT(
      clk_100 : in STD_LOGIC;
      udp_data : in STD_LOGIC_VECTOR(31 downto 0);
      sw_send_packet    : in std_logic;
      PhyResetOut_pin : out STD_LOGIC;
      MII_TX_CLK_pin : in STD_LOGIC; --25 MHz clock for 100 Mbps - not used
here
      GMII_TXD_pin : out STD_LOGIC_VECTOR(7 downto 0);
      GMII_TX_EN_pin : out STD_LOGIC;
      GMII_TX_ER_pin : out STD_LOGIC;
      GMII_TX_CLK_pin : out STD_LOGIC;
      GMII_RXD_pin : in STD_LOGIC_VECTOR(7 downto 0);
      GMII_RX_DV_pin : in STD_LOGIC;
      GMII_RX_ER_pin : in STD_LOGIC;
      GMII_RX_CLK_pin : in STD_LOGIC;
      MDC_pin : out STD_LOGIC;
      MDIO_pin : inout STD_LOGIC;
      pclk : in STD_LOGIC;
      btn : in STD_LOGIC_VECTOR(4 downto 0)
      --stop : in STD_LOGIC;
      --sending : out STD_LOGIC

      --rs232_tx : out STD_LOGIC
      );

END COMPONENT;

signal udp_data : STD_LOGIC_VECTOR(31 downto 0);
signal corner_out : STD_LOGIC;
signal x_coor_out : STD_LOGIC_VECTOR(9 downto 0);
signal y_coor_out : STD_LOGIC_VECTOR(8 downto 0);
signal frame_num_out : STD_LOGIC_VECTOR(8 downto 0);
signal clk_100 : std_logic;
signal udp_sending : std_logic;
signal raw_pix : STD_LOGIC_VECTOR(15 downto 0);
signal p_valid : STD_LOGIC;
signal pix_clk : STD_LOGIC;
signal cam_led : std_logic_vector(7 downto 0);

signal dummy_t, int_CAMA_PCLK_I, int_CAMA_FV_I, int_CAMA_LV_I,
int_CAMB_PCLK_I, int_CAMB_FV_I, int_CAMB_LV_I : std_logic;
signal int_CAMA_D_I, int_CAMB_D_I : std_logic_vector(7 downto 0);

attribute S: string;
attribute S of CAMA_PCLK_I: signal is "TRUE";
attribute S of CAMA_FV_I: signal is "TRUE";
attribute S of CAMA_LV_I: signal is "TRUE";
attribute S of CAMA_D_I: signal is "TRUE";
attribute S of CAMB_PCLK_I: signal is "TRUE";
attribute S of CAMB_FV_I: signal is "TRUE";
attribute S of CAMB_LV_I: signal is "TRUE";
attribute S of CAMB_D_I: signal is "TRUE";
attribute S of dummy_t: signal is "TRUE";

begin
```

```
IBUFG_inst : IBUFG
   port map (
      O => clk_100, -- Clock buffer output
      I => CLK_I  -- Clock buffer input (connect directly to top-level port)
   );

--  This module contains both the image capture and corner detection modules
inst_VmodCAM_Ref: entity work.VmodCAM_Ref
PORT MAP(
            SW_I => SW_I, --used to control threshold
            CLK_I => clk_100,
            RESET_I => RESET_I,
            corner => corner_out, --determine if the pixel is corner
            x_coor => x_coor_out,
            y_coor => y_coor_out,
            frame_num => frame_num_out,
            LED_O => LED_O, --used for debugging
-----------------------------------------------------------------------------
-----
-- Camera Board signals
-----------------------------------------------------------------------------
-----
            CAMA_SDA => CAMA_SDA,
            CAMA_SCL => CAMA_SCL,
            int_CAMA_D_I => int_CAMA_D_I,
            int_CAMA_PCLK_I => int_CAMA_PCLK_I,
            CAMA_MCLK_O => CAMA_MCLK_O,
            int_CAMA_LV_I => int_CAMA_LV_I,
            int_CAMA_FV_I => int_CAMA_FV_I,
            CAMA_RST_O => CAMA_RST_O,
            CAMA_PWDN_O => CAMA_PWDN_O,

            CAMX_VDDEN_O => CAMX_VDDEN_O,
            CAMAPClk_O => pix_clk,
            CAMB_SDA => CAMB_SDA,
            CAMB_SCL => CAMB_SCL,
            int_CAMB_D_I => int_CAMB_D_I,
            int_CAMB_PCLK_I => int_CAMB_PCLK_I,
            CAMB_MCLK_O => CAMB_MCLK_O,
            int_CAMB_LV_I => int_CAMB_LV_I,
            int_CAMB_FV_I => int_CAMB_FV_I,
            CAMB_RST_O => CAMB_RST_O,
            CAMB_PWDN_O => CAMB_PWDN_O,
            I_O => I_O
            );

-- data sent to base station valid, y-coordinate, x-coordinate, frame num
udp_data <= "1010" & y_coor_out & x_coor_out & frame_num_out;


inst_ethernet_test_top : ethernet_test_top
PORT MAP (
      clk_100 => clk_100,
      udp_data => udp_data,
      sw_send_packet    => corner_out, --sent based on corner
      pclk => pix_clk,
      PhyResetOut_pin =>PhyResetOut_pin,
      MII_TX_CLK_pin => MII_TX_CLK_pin,
      GMII_TXD_pin => GMII_TXD_pin,
      GMII_TX_EN_pin => GMII_TX_EN_pin,
      GMII_TX_ER_pin => GMII_TX_ER_pin,
      GMII_TX_CLK_pin => GMII_TX_CLK_pin,
      GMII_RXD_pin => GMII_RXD_pin,
      GMII_RX_DV_pin => GMII_RX_DV_pin,
      GMII_RX_ER_pin => GMII_RX_ER_pin,
      GMII_RX_CLK_pin => GMII_RX_CLK_pin,
      MDC_pin => MDC_pin,
      MDIO_pin => MDIO_pin,
      btn => btn

      );
end Behavioral;
```