Real-Time Remote Mapping

A Major Qualifying Project Report Submitted to the Faculty of

Worcester Polytechnic Institute

In partial fulfillment of the requirements for the

Degree of Bachelor of Science in Electrical & Computer Engineering

By:

Georges Gauthier

John DeCusati

January 2, 2017

Advisor:

Professor R. James Duckworth

Contents

List of Figures

Abstract

The overall goal of this project is to create a device capable of generating detailed maps and imagery of an area in real-time. This device will rely on an FPGA, and will use image processing algorithms capable of detecting, localizing, and tracking human beings. The device will gather imagery from the visual light and infrared-spectrums, as well as localization data and distance measurements from an IMU and a rangefinder. Along with gathering and processing data, the device will also serve as a long-range wireless access point, and will be able to transmit all generated maps and imagery in real-time. A major deliverable of this project is that the transmitted data will be fully processed, allowing it to be viewed remotely on less-powerful, mobile devices.

This device will be especially useful for first responders. It is intended to be mounted on a small remote control vehicle, allowing any connected user to wirelessly traverse dangerous and remote locations in search of people in need. Since this device transmits data in real-time, it will be able to provide first responders with an accurate representation of not only a 2-D floor plan of an area such as a building, but also where any people are located. An anticipated use of this device would be in the event of a building in danger of collapsing. Since it would be dangerous to physically enter the building, first responders could locate any people trapped inside and find the fastest route to them using the wirelessly transmitted floorplan. The first responders would also be aware of any dangers in their way by making use of the real-time augmented video stream. This video stream will consist of image data with overlaid with object indicators and location information on any human beings detected by the image processing algorithms.

1 Introduction

Currently there are many applications that rely on a simple video camera setup in order to gather information on remote and inaccessible locations. Although this is an effective strategy for simple surveillance, it is limited in many ways. Using current imaging and sensor technology, it is possible to gather camera images and 3D depth information on a given area as both a cost-effective and information-rich alternative to using a camera module on a device. If a product were to be created that gathers this information as a replacement to using a standalone camera module, it would be possible to use high speed data processing techniques in both hardware and firmware that would allow for the creation of an augmented real-time video feed.

This type of technology is known as Simultaneous Localization And Mapping, or SLAM. The purpose of SLAM is to compute the location of an agent within its environment, and allows for the creation of self-aware robot systems that are able to respond to their surroundings. SLAM is a common area of research in the image processing and high-speed computing field, and has been applied mainly to autonomous vehicles. We would like to propose the creation of a SLAM-like system that is capable of monitoring and mapping its environment in real-time, as well as detecting and localizing objects, such as human beings. For the purposes of our project, we would like to define our desired objective as Simultaneous Personnel Localization And Mapping, or SPLAM.

A device that is capable of both mapping its surroundings using SLAM and performing human detection in real-time would be applicable to many different fields. We are especially interested in creating a proof of concept sensor suite capable of performing these tasks that can be added to existing robotic systems as a stand-in replacement for a video camera. This type of technology would allow for people such as firefighters or first responders to wirelessly traverse dangerous and remote locations in search of people in need. We envision our sensor suite being able to process data so that its users would be provided with a 2D "floorplan" of the area being traversed by the sensor suite, as well as an augmented video feed with

imagery containing indicators for any detected human beings in the area.

One type of technology that would be useful for performing the high speed data processing necessary for SPLAM is a Field Programmable Gate Array , or FPGA. FPGAs pose several advantages over using standard computing or microcontroller technology for real-time data processing, as they have the ability to manipulate digital information in parallel using hardware only. This allows for extremely high-speed performance, as calculations can be run in parallel and are only dependent on their data inputs as opposed to waiting for specific tasks or scripts to run on a microcontroller or computer software interface.

Although FPGA technology is highly applicable to performing SLAM-like tasks due to its high speed, there are currently few existing commercial products that use FPGAs for the purpose of performing SLAM. Most current SLAM implementations rely on the use of a sensor suite connected to a computer or system on chip (SoC) computing device that performs data analysis using software or a real-time operating system (RTOS). This means that data must first be collected by a sensor suite, and then transferred to an external computing device that is only capable of processing it serially based on its arrival time. Although this type of setup is acceptable for performing real-time situational awareness analysis, we propose that an embedded, FPGA-based SPLAM device would be a much more elegant and higher-speed solution.

2 Background

A major concern with real-time image processing, especially in first responder situations, is speed. Because FPGAs have the ability to process data in parallel, they are ideal for this type of application. Using an FPGA for this system will enable all data inputs to be processed at the same time, thereby dramatically increasing throughput speed. Dealing with each input separately makes it easier to combine everything together, especially because each component functions at a different clock speed. Also, since everything is running in parallel, more cameras can be added to the system to increase the field of vision of the device without introducing any latency in the system, as long as enough memory is available.

SLAM is a widely expanding field with much potential for improvement. One application of such a system is a proof of concept of camera-based SLAM systems, presented by Andrew Davison of Oxford University in a research paper entitled "Real-Time SLAM with a Single Camera" [6]. This system is handheld and relies on a computer using a 2.2 GHz Pentium processor connected to a single camera and laser rangefinder. The system requires prior knowledge of the area being analyzed before it can successfully localize and map. It implements edge detection, but is limited to the narrow field of vision of the rangefinder, so it is only able to map an object directly in front of it. This system carries a latency of around 33 milliseconds. An output frame of the device is shown in Figure 1.

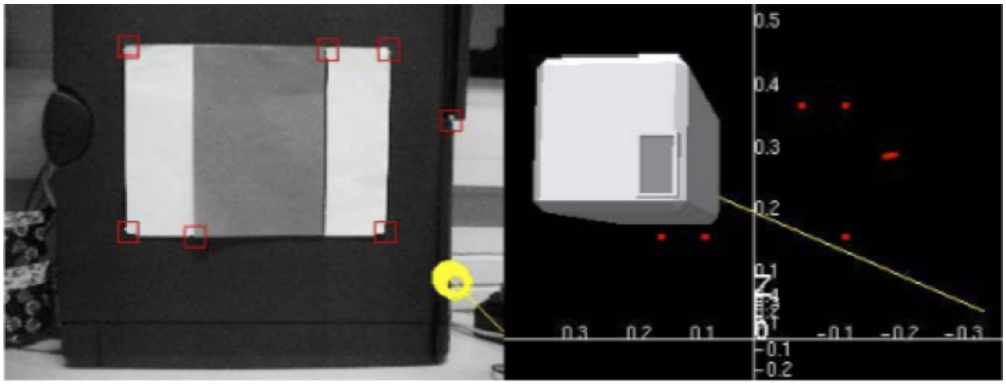


Figure 1: Real-Time SLAM with a Single Camera [6]

The frame on the left in Figure 1 is the video feed with 6 points of a paper target input as

prior knowledge, along with successfully marked identifying features (marked as red squares), and another identifying feature that is not marked for measurement (marked by a yellow circle). The frame on the right is a localization graph displaying the positions of all red squares.

A more commercial device similar to our concept is Serveball?s Squito^{TM} [12]. Squito is a wireless, throwable, 360° panoramic camera that implements target detection to stabilize the video feed from its many cameras. It is shown in Figure 2 below.

Figure 2: Serveball's Squito [12]

Squito utilizes a microprocessor receiving input from cameras, as well as orientation and position sensors in order to transmit a real-time stabilized video of its adventure. The device is still in the prototype stage and is receiving interest from first responders. The image in Figure 3 shows the input from the Squito's four cameras on the left, and the corresponding stitched output on the right.

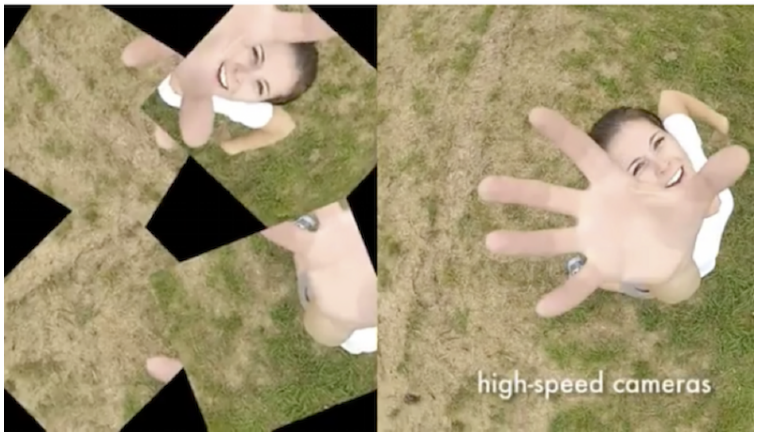


Figure 3: Serveball's Squito Input and Output [12]

By using multiple camera sensors in a sensor suite, it is also possible to determine depth information from corresponding images of an area. This technique is known as stereo imaging, and the process of gathering depth information from a pair of stereo images is known as disparity mapping. University of Bologna researchers Stefano Mattoccia and Matteo Poggi have worked to implement a real-time disparity mapping algorithm on an FPGA, and an example of a stereo image disparity is shown in Figure 4 below [13]. Using their stereo vision algorithm, the researchers are able to generate real-time image data showing the relative locations of objects within an image frame using color gradients. Based on this depth information, it is also possible to detect objects located within the field of view of the stereo imaging system, as shown in Figure 4. An implementation similar to this would be extremely useful in a SLAM-like system, as it would allow for the localization of objects and creation of 2D "floorplans" of an area in real-time using only two camera sensors.

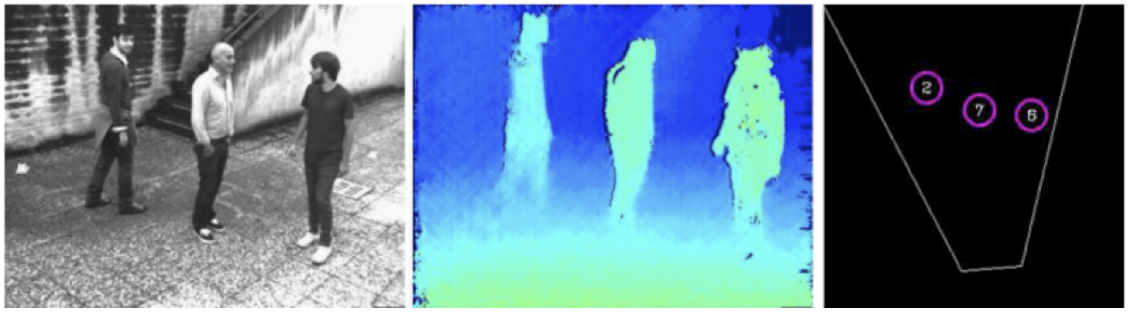


Figure 4: From Left to Right: Original Image, Disparity Map, Object Detection Results [13]

Many security systems implement human detection and human body tracking in order to increase their effectiveness. These devices process real-time images in order to identify human characteristics, and are limited to the field of vision of a stationary or rotating camera. An example of this type of system is explored by the Mitsubishi corporation in a research paper entitled "Human Body Tracking by Adaptive Background Models and Mean-Shift Analysis" [19]. The paper explores a stationary image processing system implemented on a PC platform with a 1.8GHz processor that yields a maximum processing time of 100 milliseconds. An output frame of the system is shown in Figure 5 below.

Figure 5: Human Body Tracking by Adaptive Background Models and Mean-Shift Analysis [19]

Our proposed device will combine the ideas of the four systems examined. It will be able to simultaneously localize and map an area, as well as implement human detection algorithms. The device will be capable of generating real-time 2D maps of any area it has traversed with humans' locations labeled, and an augmented video feed of what the device is recording with any humans' positions marked.

In order to successfully implement this system, we propose the creation of a device that will rely on two stereo cameras, a laser rangefinder, and an inertial measurement unit (IMU) as its sensor suite, as shown in Appendix Item 2. Limitations of previous art have been in their ability to combine human detection with real-time localization and mapping of a

large field of vision. Little to no existing commercial products are also capable of processing their gathered data locally and in real-time, with their gathered data usually requiring post-processing on external computing devices. Stereo cameras will allow our device to calculate disparity, just as human eyes do. Although disparity is useful for localization, it is not enough for accurate mapping because it only accurately provides the relative distance between objects. The inclusion of a rangefinder will allow for precise base distance readings, and an IMU will be used to spatially reference all gathered data. All of this data will be combined with the disparity maps and image data in order to create flawless localization and mapping. All time-dependent processing required for the device will be mainly done in parallel using hardware on an FPGA. An overall functional block diagram of our intended implementation is shown in Figure 6 below.

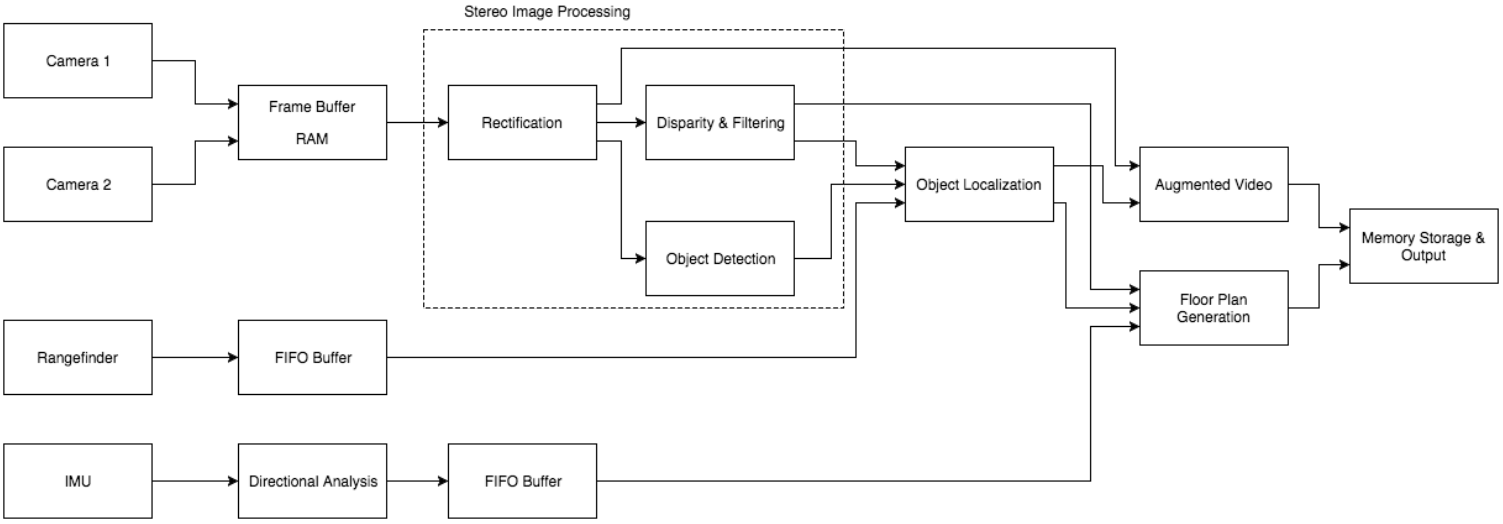


Figure 6: Functional Block Diagram

Most applicable previous camera-based systems have also focused on object detection from a stationary point, or edge detection from a mobile platform. Our project aims to combine these concepts, by creating a mobile device that detects people which will be especially of use in many first responder situations. In addition, this device will receive data from the visible light and infrared spectrums in order to identify people quickly and accurately in a way that has not been previously implemented.

As our research has progressed over time, our project objectives have continually evolved. We originally envisioned the creation of a device that used laser rangefinders to create 3D maps of its surroundings, similar to that of a Carnegie Mellon University device created in order to volumetrically map abandoned mines [20].

As our research progressed, we believed that we could use a visual light and thermal imaging camera set to gather information on an area, and supplement that data with IMU and rangefinder readings in order to produce detailed maps of our sensor suites? surroundings. Eventually we came upon the concept of disparity mapping and generating depth information from image data, and decided that we would again like to shift the overall setup of our device to rely mainly on stereo image data. Due to our overall budget and the resources that have been made available to us, in the coming terms we plan to use an electronic rangefinder, IMU, and stereo camera pair to generate real-time SPLAM video and floorplan information. Although we were also originally planning on including a thermal camera in our sensor suite as well, we have decided to eliminate the module in favor of higher quality cameras due to its prohibitive cost, low resolution, slow sampling rate, and small field of view. More information on this decision can be found in Appendix item 4.7.1.

3 System Design

4 System Implementation

4.1 Rangefinder Operation

Because this device is intended to traverse unknown locations and create a 2-dimensional map, data accuracy, precision, and reliability are vital. As such, proper equipment is needed to suffice these needs.

4.1.1 Selection

The project's rangefinder selection depended on the following criteria: field of vision, depth of sense, accuracy, precision, and cost. Many of the rangefinders limited by our budgetary restrictions were only strong in one of our project's vital criteria. However Professor Duckworth, and WPI's Electrical and Computer Engineering and Robotics Engineering Departments generously donated the URG-04LX Scanning Laser Rangefinder for the purpose of this project. The URG-04LX is a very sensitive piece of equipment that has a field of view of 240 degrees, a depth of data of 4 meters, and accuracy to within 10 millimeters, which is perfect for our application [8]. Figure 7 below shows the URG-04LX rangefinder that we will be using.

Figure 7: URG-04LX Scanning Laser Rangefinder

4.1.2 Communication

The URG-04LX rangefinder uses the RS-232C communication protocol over UART. RS-232 is a form of differential serial data transmission which recognizes a logic high from -3V to -25V, and a logic low from +3V to +25V [1].

The rangefinder can be connected to one of the ZedBoard's Pmod connectors because they support UART communication. The Pmod connectors use TTL communication, which is a form of non-differential serial data transmission that recognizes a logic high of +3V to +5V and a logic low of 0V [14]. Figure 8 shows a timing diagram of both RS-232 and TTL communication protocols.

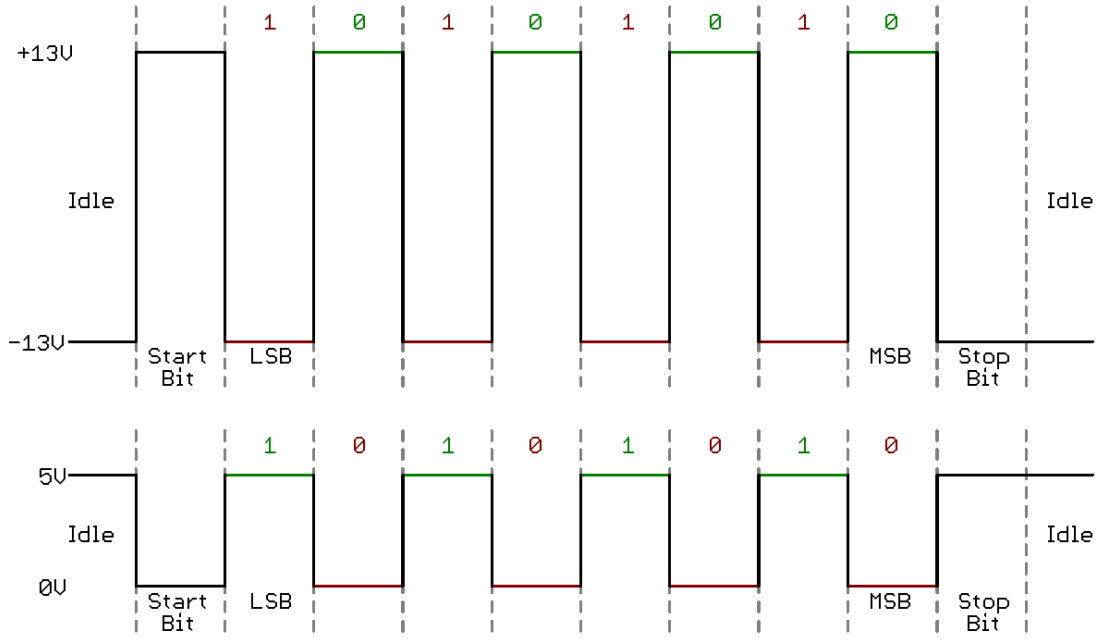


Figure 8: Timing Diagram of RS-232 (top) and TTL Communication Protocols [14]

Since these two serial communication formats have incompatible logic levels, an RS-232 to TTL converter is needed so that the rangefinder can communicate with the ZedBoard. The converter's TTL side will be connected to the ZedBoard's Pmod connector, and the RS-232 side will be connected to the rangefinder. However, for ease of connection and testing, the 9-pin DSUB RS-232 connector will be connected to an RS-232 breakout so that the pins

can be easily accessed. Figure 9 shows the RS-232 to TTL converter attached to the RS-232 breakout board.

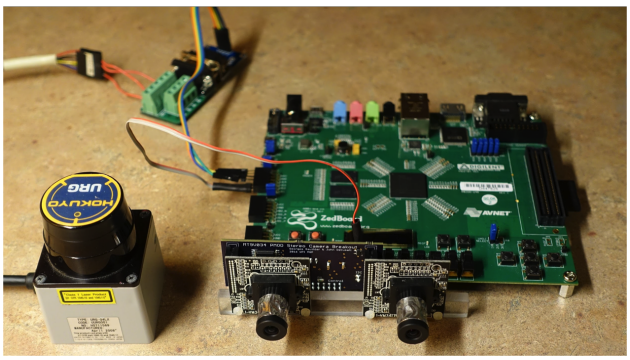


Figure 9: RS-232 to TTL Converter with RS-232 Breakout Board

Although the ZedBoard's Pmod connectors are sufficient for UART communication with the URG-04LX, the power specifications are not compatible; the Pmod connectors output 3.3V but the rangefinder requires 5V [4, 8]. Thus, the rangefinder was powered externally by a lab bench power supply.

4.1.3 Commands

The rangefinder defaults to a communication speed of 19.2 kbps, or 19200 baud. Using that baud rate over UART, the rangefinder recognizes four different commands: the version command, the laser illumination command, the communication speed setting command, and the distance data acquisition command. The version command is used as a test; as soon as it is received, the rangefinder transmits the device specific information. The laser illumination command is used to turn the laser on and off. The communication speed setting command is used to change the baud rate. The distance data acquisition command is the main command is used to request the distance data from the rangefinder [9].

The distance data acquisition command is the primary command that will be used for the purpose of this project. This command consists of five different parts that control the data output: 'G', the data starting point, the data end point, the cluster count, and either a line feed or a carriage return. The start point is the step of the area from where the data

reading starts, and the end point is the step of the area where the data reading stops. The data reading starts at the start point and traverses counterclockwise until the end point. Changing these steps changes the field of vision of the device. Figure 10 below shows a top-down view of the device field of vision with the steps labeled.

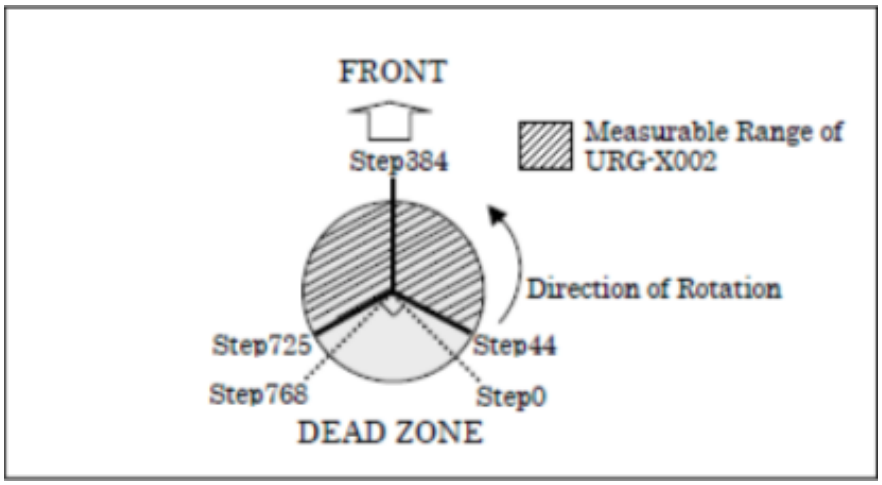


Figure 10: Top-Down View of Rangefinder Field of Vision [9]

For this project, we set the beginning point to '000' and the end point to '768' to obtain the device's maximum coverage of 240 degrees. Note that the device's angular rotation per step can be calculated by using Equation 1 below.

$$360^\circ/1024 \text{ steps} = 0.3515625^\circ \text{ per step} \tag{1}$$

Each step around the rangefinder's field of view corresponds to a change of 0.3515625°.

The cluster count is the number of neighboring points that are grouped together as a cluster. The cluster count was set to '01' in order to have a cluster count of one data point.

Putting all of these settings together, we get the data acquisition command 'G00076801\n' which was transmitted from the ZedBoard to the rangefinder to request one cycle of data.

4.1.4 Testing

We were able to test the rangefinder by connecting it to a laptop via its USB port. We used PuTTy, a serial console application, to communicate with the rangefinder. Figure 11 shows the data transfer via PuTTy between a laptop and the rangefinder. Note that PuTTy only shows data received, and that the rangefinder always echoes back the command that it receives.

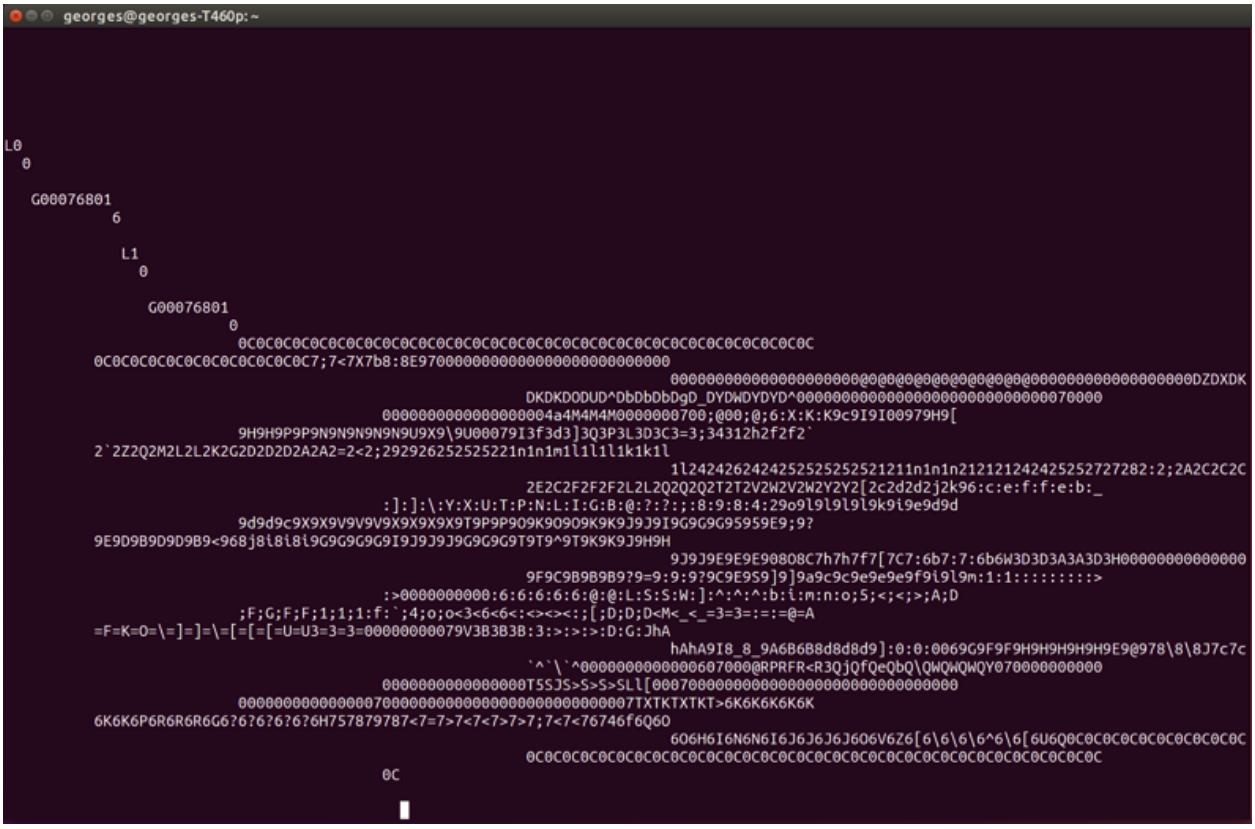


Figure 11: Rangefinder Communication Test via PuTTy

The figure above shows four communication sequences. The first is the laser illumination command 'L0\n'. Since the laser defaults on, this command turned the laser off. The rangefinder responded to this command first with the echo 'L0\n', and then with '0', indicating success. The second command is our data acquisition command 'G00076801\n'. The rangefinder responded with '6', indicating an error code, which was caused by the laser being off. The third command is the laser illumination command again, which turns on the laser.

The rangefinder's response was '0' again, indicating success. The last command shown is the data acquisition command again. The rangefinder's response begins with '0', indicating success, followed by the distance data block. The data block consists of 768 points, specified by the data acquisition command. Each data point consists of two characters [9]. By communicating with the rangefinder via PuTTy, we were able to observe the rangefinder's behavior and confirm the data acquisition command functions properly.

In addition to communication via PuTTy, the URG-04LX has a data viewing tool which is a useful application by Hokuyo Automatic that can be used to view, record, and replay the device's data. Again, to use this tool the device must be plugged into a computer via its USB port. Figure 12 below shows a screen capture of the application recording data captured by the rangefinder.

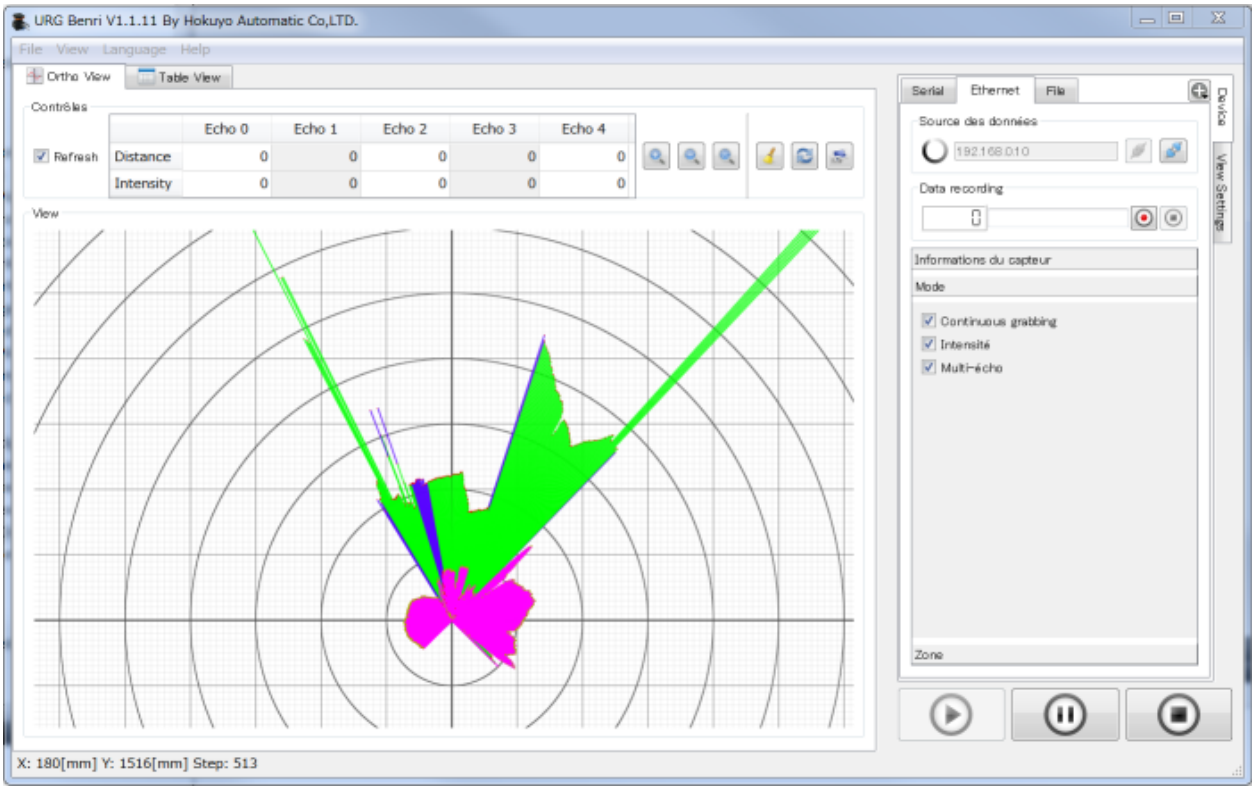


Figure 12: Screen Capture of the URG-04LX Data Viewing Tool [10]

Note that the start point of 0, end point of 768, and dead zone align to that shown in Figure 10. For this project the data viewing tool was used to verify our project's data

processing functionality.

4.2 Rangefinder Implementation

With the data acquisition command tested and functioning properly, the rangefinder needs to be connected to the ZedBoard. Since the URG-04LX uses UART communication, there are a few reasonable options to create a UART controller on the ZedBoard.

4.2.1 UART Options

As the ZedBoard is such a powerful device, it has a few different options for controlling UART. A few of them are by controlling UART through linux, through a MicroBlaze soft-core processor, or through the Zynq-7000 Processing System. Running linux on the ZedBoard would use much of the board's valuable resources, and we would only be using a fraction of the capability provided by linux. The MicroBlaze soft-core processor would be a better alternative, but it runs in the programmable logic in the FPGA and is unnecessary when the ARM processor on the ZedBoard is unused [22]. Because of this, we decided to utilize the ARM processor on the ZedBoard by using the Zynq7 Processing System via Xilinx's Zynq-7000 Processing System Intellectual Property (IP) core.

4.2.2 Zynq7 Processing System

The ZedBoard SoC features a dual-core ARM Cortex-A9 MPCore processing system and Xilinx Programmable Logic. The Zynq7 Processing System IP core acts as a logic interface that integrates the Programmable Software (PS) with the Programmable Logic (PL), which allows access to both on-chip and external memory interfaces, to PL clocks, to many I/O peripherals, and even to extended I/O peripherals [23]. With all of this overwhelming functionality, the processing system is easy to customize, featuring a simple user interface, once it is added into a project's block design. The user interface can be used to change the processing system's activated features. Figure 13 below shows the processing system

customization window.

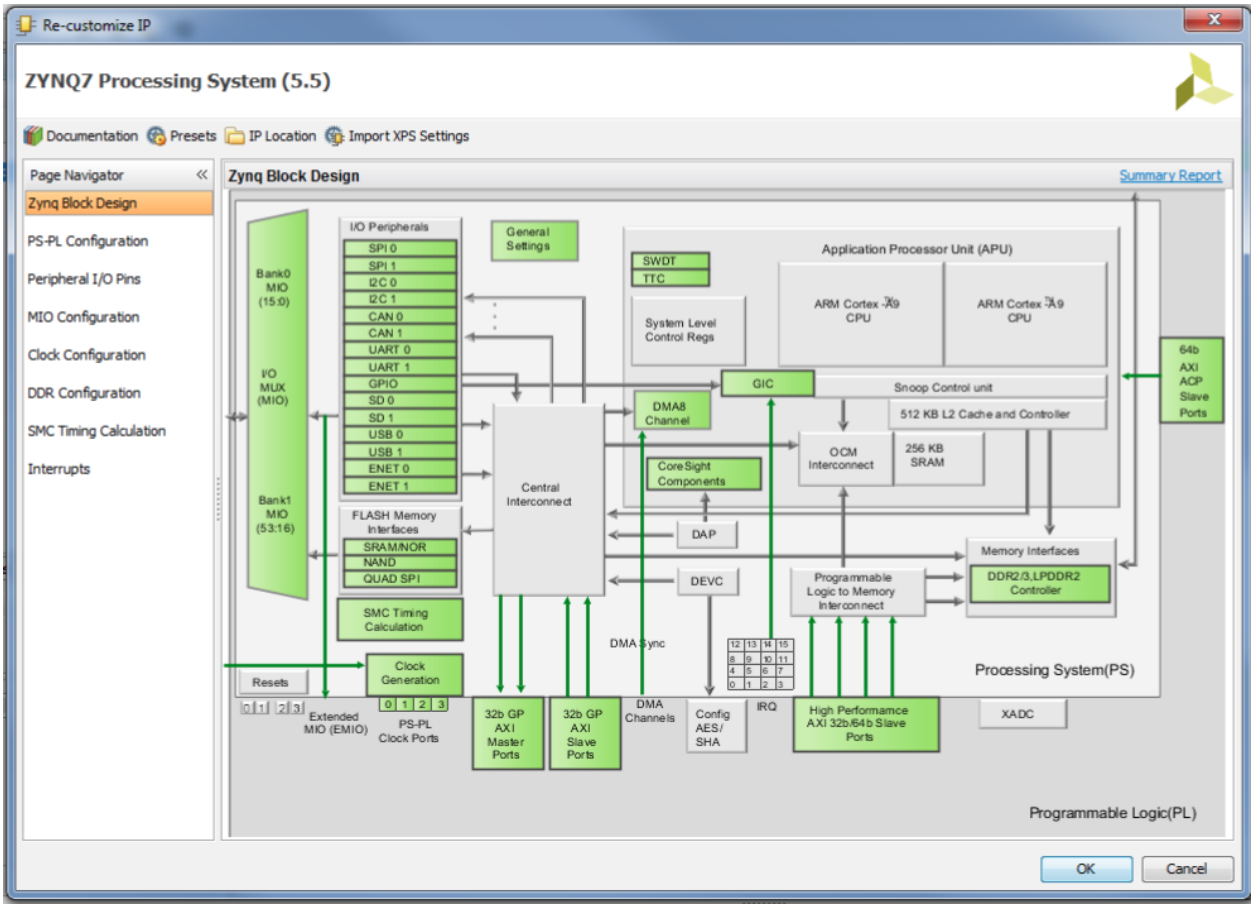


Figure 13: Zynq7 Processing System Customization Window [23]

In the figure above there are two options for UART shown: UART0 and UART1. The functionality of UART0 and UART1 are nearly identical, except that UART1 has the capability of being routed to the ZedBoard's USB UART port, which is not compatible with the rangefinder [4]. So, we arbitrarily chose UART0 and routed the signals to MIO10 and MIO11, which correspond to the ZedBoard's PS Pmod, JE.

After choosing UART0 and configuring the MIO pins, the Baud Rate needs to be configured such that it corresponds with the rangefinder's default communication speed, 19200 Baud [9]. This can be done in the processing system's customization window under PS-PL Configuration on the sidebar in Figure 13. Figure 14 below shows the PS-PL Configuration window.

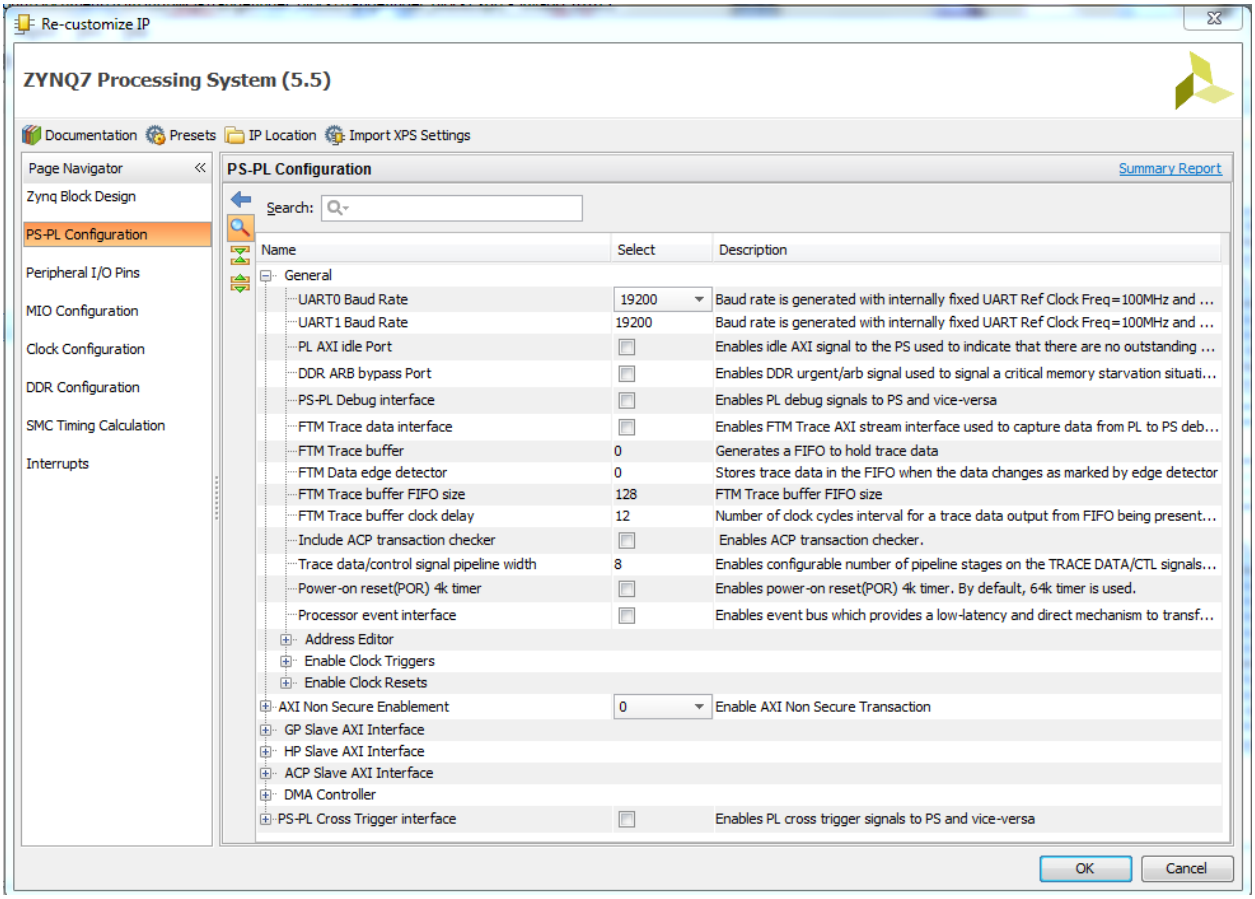


Figure 14: Zynq7 Processing System PS-PL Configuration Window

With the processing system customized in this fashion, the programmable logic's configuration is complete.

4.2.3 PS-PL Communication / Creating Custom IP

maybe break into two sections? one for ps-pl and talk about what AXI is and why we need to use it, and then in the testing and results section make a second one about how we needed to create custom axi peripherals? whatever works best I HAVE NO IDEA WHAT TO WRITE FOR THIS

4.3 Rangefinder Data Processing

With the communication between the ZedBoard and rangefinder, and between the PS and PL functioning properly the data processing can begin.

4.3.1 Programmable Software

The Programmable Software (PS) is responsible for all of the communication with the rangefinder, in addition to formatting the data before it is sent to the Programmable Logic.

The distance data acquisition command is transmitted by the PS when it receives a signal from the PL. Once the command is sent, there are two different pieces of information that the software needs to capture: the distance data and the step count. The rangefinder transmits the distance data half a data point (one character) at a time, but it does not transmit the step count. As such, the software receives the distance data one character at a time, so the first character is stored in a buffer until the second character is received. Once the second character is received the distance data buffer is updated to hold both characters, the step count is incremented, and then both are sent to the PL by writing them to the memory location referenced by it, as discussed in Section 4.2.3.

4.3.2 Programmable Logic

The Programmable Logic (PL) is responsible for all data processing, block memory manipulation, and outputting to VGA.

The rangefinder encodes each data point before transmitting it. The uncoded data point is expressed with 12 bits, in order to cover the device's maximum distance of 4095 millimeters. The 12-bit data is separated into two 6-bit data points, and 30_{16} is added to each. The resultant data point is comprised of two ASCII characters [2]. The decoding process takes place in the PL, and is the inverse of encoding where 30_{16} is subtracted from each character and then they are merged together [9].

Since the rangefinder provides the distance away from an object and the angle at which

it was detected, the data is essentially expressed in the polar coordinate system [21]. We will be outputting our data on a VGA screen which expresses data in the rectangular (Cartesian) coordinate system, so we must convert from polar to rectangular coordinates. This is accomplished by using the step number, which corresponds to an angle around a circle as shown in Figure 10.

Converting from polar to rectangular coordinates requires basic trigonometry. Luckily, Xilinx supports a few options for performing complex math operations in the PL: a Coordinate Rotational Digital Computer (CORDIC) function, multiplier IP blocks, or Lookup Tables (LUTs). Due to latency concerns and ease of integration, we decided to implement a LUT with a multiplier instead of a CORIC function. The values in the LUT will be used to extract the horizontal and vertical components from the polar coordinate. Taking advantage of a circle's symmetry, the LUT only needs to hold values 256 values, which correspond to the amount of rangefinder data steps in one quadrant of a circle. Each step value is manipulated such that it corresponds to two addresses in the LUT: one for the horizontal scale factor, and one for the vertical scale factor.

The LUT was set up in Vivado using a read-only BRAM IP core with a depth of 256, and was initialized by importing a coefficient file (.coe). The 256 values in the coefficient file were calculated by using Equation 2 for step values between 128 and 384, corresponding to one quadrant of a circle. Note that multiplying by 4096 equates to a 12-bit left shift, and is used to decrease error due to rounding in later data manipulation.

$$\mathrm{LUT}[step] = \sin((384 - step) \times \frac{\pi}{180}) \times 4096 \tag{2}$$

Only one address in the BRAM can be read from at a time, but we require both a horizontal and vertical scale factor. To avoid read conflicts, the 256-address LUT was split into two 129-address LUTs, where one LUT corresponds to $0° \leq \theta \leq 45°$, and the other corresponds to $45° \leq \theta \leq 90°$. Separating the LUT to solve this problem takes advantage of

the property that sine and cosine are 90° out of phase with each other[1]. The code for the coefficient files can be found in Appendix 7.8.1.

Once the LUT was customized, the BRAM IP Wizard window specifies the BRAM's latency. In this case, once the addresses are calculated there is a latency of 2 clock cycles before the data from the LUT is valid. Once the horizontal and vertical data is valid, each is multiplied by the decoded polar coordinate data point by being input to a Multiplier IP block which, in this case, has a latency of (LOOK UP LATENCY FOR MULTIPLIER). This transformation changes the data from polar to rectangular coordinates. After this step, the data is shifted back in a manner such that the data points will be able to fit on a VGA screen with resolution 640×480 pixels. Next the data needs to be localized to the device's location, so the x- and y-coordinates are shifted by the device's location. After this step, the x- and y-location accurately reflect the distance data localized to the device.

With proper x- and y-coordinates, the data is ready to be stored in memory. Another BRAM IP was created for the VGA control. Our VGA resolution is 640×480, so our VGA BRAM requires 307,200 addresses. This BRAM IP was customized to function in the write-first, dual port configuration. We implemented this BRAM module such that one port is a write-only port using our 100 MHz clock, and the other is a read-only port using our 60 Hz VGA clock. This BRAM IP avoids memory access conflicts by writing to memory before attempting to read. Our write address was calculated by using another Multiplier IP with Equation 3.

$$\text{write address} = (640 \times y_{\text{location}}) + x_{\text{location}} \tag{3}$$

With the data stored in memory, it is ready to be output to a VGA screen. We implemented a VHDL 640×480 VGA controller model by Digilent, found in Appendix **??** ADD ANOTHER APPENDIX FOR CODE, to control our VGA logic. In addition, the

[1]This process could have been avoided by setting up the BRAM in a True Dual Port RAM configuration, so that there are two separate, individually addressable address busses for the same BRAM block.

ZedBoard's VGA pins were configured in a constraints file (.xdc) to support 12-bit color resolution [4]. Similar to Equation 3, Equation 4 was used to calculate the read address of the VGA BRAM IP.

$$\text{read address} = (640 \times v_{\text{count}}) + h_{\text{count}} \tag{4}$$

4.4 Inertial Measurement Unit (IMU) Operation

talk about need/motivation for imu

4.4.1 Selection

something simple, easy to use, sensitivity, pmod connector

4.4.2 Communication

spi vs i2c

4.4.3 Settings

setting the registers, turning on the magnetometer

4.4.4 Commands

idk if this is necessary

4.5 IMU Implementation

talk about what it takes to insert the imu into the design

4.5.1 Recustomizing the Zynq7 Processing System

talk about re-customizing the processing system, rerouting the SPI pins to the EMIO PMOD, differential signals.

4.6 IMU Data Processing

The IMU's data processing was implemented in the Programmable Software because it involves complex mathematical equations, and can be easily integrated with the rangefinder's data in the Software Development Kit.

4.6.1 Programmable Software

Talk about the math, integration into the rangefinder's code, and how the IMU data rotates the rangefinder's data.

4.7 Camera Operation

4.7.1 Camera Selection

In order to determine the proper camera modules for this project, several steps needed to be taken. Due to our limited project budget and time constraints, we focused on finding camera modules that were low-cost and simple to communicate with. This ruled out many low-cost camera modules that rely on complicated communications protocols, as well as all commercially available stereo image sensor suites. One other important factor that we sought to satisfy in our camera setup was the use of global shutter cameras, which acquire image data from the entire image sensor at once, rather than sequentially by pixel. The use of global shutter camera modules makes it so that our setup is not susceptible to lens artifacts, or distorted imagery due to moving objects or a moving camera setup. With these factors kept in mind, the decision matrix shown below was created for selecting a proper camera module. Each module evaluated was given a ranking from 1-10, with 10 representing the ideal camera module for our project.

Camera Module	Max Frame Rate (FPS)	Resolution at Max Frame Rate (px.)	Cost	Requires External Adapter	Data Transfer Interface	Shutter	Field of View (deg.)	Rank 1-10
OV7670	30	640x480	$10	No	Parallel	Rolling	25	5
Raspberry Pi Camera	90	640x480	$30	Yes, $53	MIPI (CSI2)	Rolling	49	6
PC1089K	60	720x480	$32	No	NSTC/ PAL	Rolling	Not Given	5
OV4682	330	640x480	$89	Yes, $50	MIPI	Rolling	Not Given	6
MT9V034	**60**	**752x480**	**$73**	**No**	**Parallel**	**Global**	**55**	**9**

Based on our decision matrix, we believed that the MT9V034 camera module would be ideal for our stereo camera interface. These camera modules are the only low-cost global shutter option we were able to find in our research, and are ideal for taking images in a sensor suite that is susceptible to motion. The MT9V034 also uses a parallel data interface and relies on an external clock and shutter trigger, making the module ideal for interfacing with an FPGA-based stereo imaging setup. After obtaining two of the MT9V034 cameras, the operation of the camera modules was then investigated. In order to gather working images from each camera module, we first needed to understand what circuitry our camera module breakouts contained so that we could interface with them. The MT9V034 camera breakouts used have been purchased through Leopard Imaging Inc. Although these camera module breakout boards are intended to be used with Leopard Imaging's LeopardBoard ARM development board, the breakouts were found to contain only the supporting circuitry recommended in the MT9V034 datasheet, and we decided that they would be ideal for our application [11, 18]. Once the schematics of each camera module breakout were known, it was then possible to design a basic control interface for each camera.

4.7.2 Camera Signaling

By default, the MT9V034 camera module will continuously gather image data at 60Hz as long as it is supplied with an external clock signal and output is enabled [18]. Several output signals from the camera module are then used to transmit image data. Each image,

or frame, is broken up into individual "lines" which correspond to a line of pixels that stretch the width of the frame. Since our camera module captures images at 752x480 pixel resolution, one frame will contain 480 lines of 752 pixels each. The camera module breaks up image data by frame and line, and camera data pins FRAME_VALID and LINE_VALID are toggled to indicate the transmission of a frame or line. The timing diagram shown in Figure 15 shows the operation of these pins while transmitting an image.

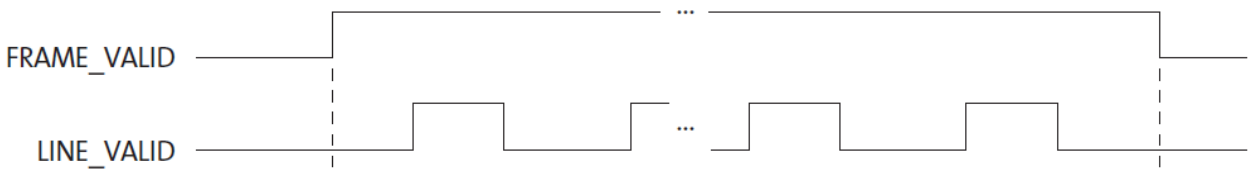


Figure 15: Frame and Line Valid [18]

Since the MT9V034 module transmits image data in parallel and each pixel contains 10 bits of resolution, 10 pins are used to transmit pixel values. Pixel data is transmitted in correspondence with LINE_VALID and output clock signal PIXCLK. When LINE_VALID is asserted, the pixel data pins are updated with values corresponding to pixels 0-751 of the given line. Values for each pixel are written out on the falling edge of the camera's PIXCLK pin, allowing for each pixel's value to be read on each rising PIXCLK edge. A full LINE_VALID data transmission sequence will therefore contain 752 PIXCLK cycles, corresponding to the 752 pixels that make up the given line. A timing diagram of this data transmission scheme is shown in Figure 16.

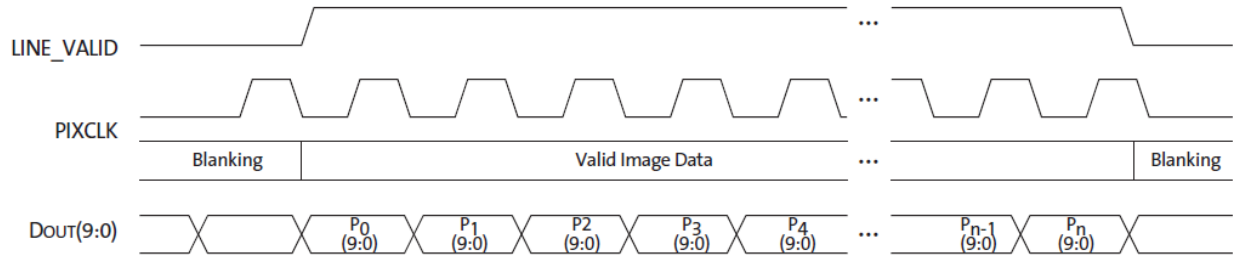


Figure 16: Line Data Transfer [18]

The default camera data transmission scheme was also examined using an oscilloscope,

as shown in Figure 17, with channels 1-4 corresponding to camera PCLK, FRAME_VALID, LINE_VALID, and Data[0], respectively. In the case of Figure 17, the camera is initially powered off, resulting in an inactive PCLK signal during the beginning of the recording.

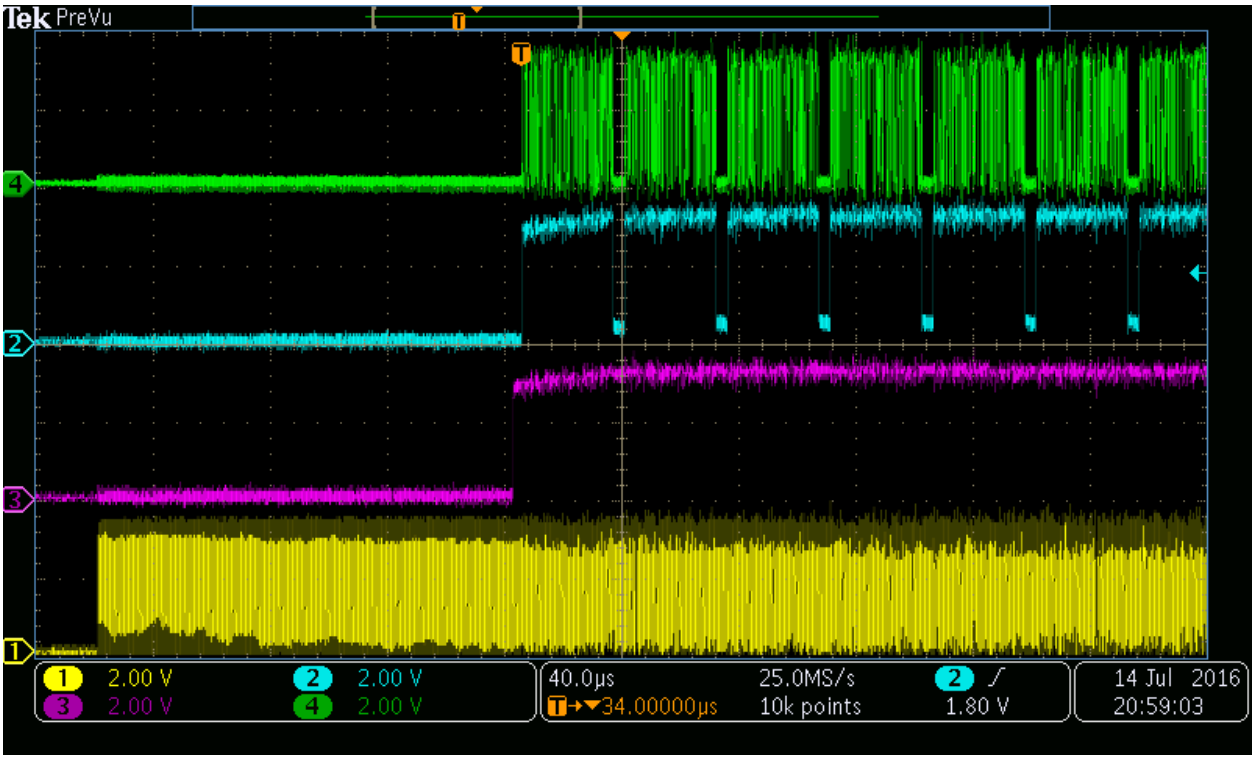


Figure 17: Camera Data Transfer

4.7.3 I²C Control

The MT9V034 Camera module's mode of operation can be configured using a standard I²C control interface. I²C, or Inter-Integrated Circuit, is a bidirectional serial interface that allows for a master device to read from and write to several slave devices sharing the same data bus. An I²C interface will use a Serial Data Line (SDA) and Serial Clock Line (SCL) that are normally pulled to 5V. When one connected I²C device wishes to communicate with another, it will pull the SDA line low while leaving the SCL line high. The master device will then begin clocking the SCL line, and SDA will be used to transfer 7 bits representing the address of the desired slave device, along with an 8th bit representing whether it would like to read from or write to the device. An example of this transfer is shown in Figure

18. A second 8 bit sequence representing a specific register within the slave device may also be transmitted following the device address. For example, if the master device wishes to write to slave device 0x40 at register 0x00, it will transmit 0x41 (address 0x40 and WRITE), followed by 0x00. If the slave device receives this transmission, it will acknowledge by pulling the SDA line low. At this point, the master can then transmit the value that it wishes to write to the given slave address and register. If the operation were a read rather than a write, the slave would transmit a value back to the master.

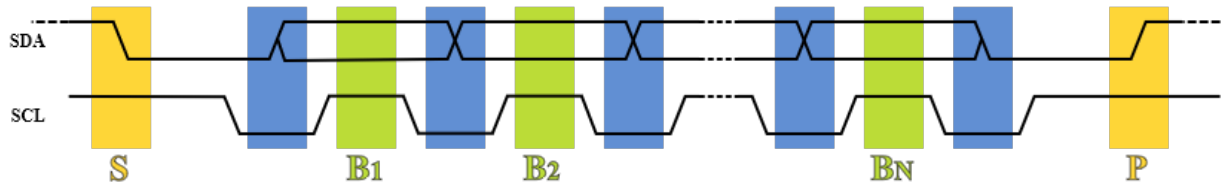


Figure 18: Example I^2C Data Transfer

Based on the LIVM34LP camera board schematic, each breakout board have been configured so that its camera is accessible at I^2C address 0x58 [11, 18]. Note that since both cameras come configured with the same I^2C bus address, a pullup resistor must be added to one of the cameras I^2C address lines so that both are individually accessible on a shared bus.

4.7.4 Image Buffering

Since each camera image contains 752x480 pixels with 10 bits of resolution per pixel, a full camera image will consume 3,609,600 bits, or 440.6kB, as shown in Equation 5.

$$Image\ Size = 752px * 480px * 10\frac{bits}{pixel} = 3609600\,bits * \frac{1\,byte}{8\,bits} * \frac{1kB}{1024\,bytes} = 440.625kB \quad (5)$$

In order to send a camera image to a computer or monitor for viewing, several steps need to be taken. Although it would be ideal to transfer the image directly from the camera to a computer or display, this would be difficult to achieve due to the high speeds of the camera's

data output. In order to properly synchronize camera data with a VGA display, both the camera and VGA display would have to run at exactly the same clock speed, and would need to have the same amount of vertical and horizontal blanking to display each pixel in its correct location. If the image were transferred to a computer, the act of packaging the information so that it may be interpreted by said computer would place severe limitations on the speed of the system. A proper solution to these timing issues would be to buffer the image between the camera and the desired output source, since this would allow for separate clock domains to be used for camera data transfer and data output. However, the act of locally buffering a camera image on an FPGA would also be difficult due to low memory resources.

Although 440kB may seem like a relatively small image size, creating a buffer object large enough for storing said image would consume an extremely large amount of logic. For reference, a standard Nexys3 FPGA evaluation board contains only 18kB of onboard Block RAM (internal memory), and would not be able to buffer an image of this size without the use of external memory[2]. This leaves the final option of using either external memory or a First-In First-Out (FIFO) memory array for transferring a captured image between clock domains. During initial development, an AL422B FIFO IC was used, since the IC has been created specifically for buffering VGA imagery similar to that of the MT9V034 camera module, and can be connected directly to the camera module outputs [3]. The AL422B FIFO module contains 3M-bits of RAM that can be written to and read from in parallel, and supports separate input and output clock speeds between 1-50MHz [3]. This means that the camera module can write pixel data to the FIFO as long as it operates at a speed between 1 and 50MHz, and the FPGA can independently read from the FIFO at any speed within the same range. Note that since this FIFO supports only 8-bit parallel data in and out, the lowest two bits of camera pixel data must be truncated. This isn't a major issue, since the truncation will correspond to a 4/1024 reduction in the range of values that each

[2]Xilinx, *Spartan-6 FPGA Block RAM Resources*, 11.
`http://www.xilinx.com/support/documentation/user_guides/ug383.pdf`

pixel can map to.

4.8 Disparity Algorithm

Some important properties of the stereo camera setup used in this project may be taken advantage of in order to extract 3D depth information from 2D image data. Since both cameras will capture imagery of the same scene from slightly different vantage points, depth information on the scene may be extracted by calcuating the pixel offsets, or disparity, between the same object's relative location in each image. Given this pixel offset, one may determine the distance from the camera pair to a given object using simple geometry based on the focal length and baseline of the stereo camera pair. Note that each camera must have the same focal length, or distance from the image sensor to the lens of the camera. The baseline of the stereo camera pair is the distance between the two image sensors, which is usually a similar length to the average distance between a pair of human eyes [5].

4.8.1 Image Rectification

One simple way of determining the disparity between objects in a stereo image pair is known as the Sum of Absolute Differences. The Sum of Absolute Differences algorithm operates under the assumption that objects in both camera images lie on the same horizontal line between both images, known as an epipolar line. An example of shared epipolar lines between camera imagery is shown in Figure 19 below. Although an ideal stereo camera setup would contain shared epipolar lines between camera images, raw image data from each camera will contain slight differences in object location based on the physical position of the camera modules, as well as minor differences in the lenses of each camera. Both input images can be adjusted to share the same epipolar lines through a post-processing step known as image rectification.

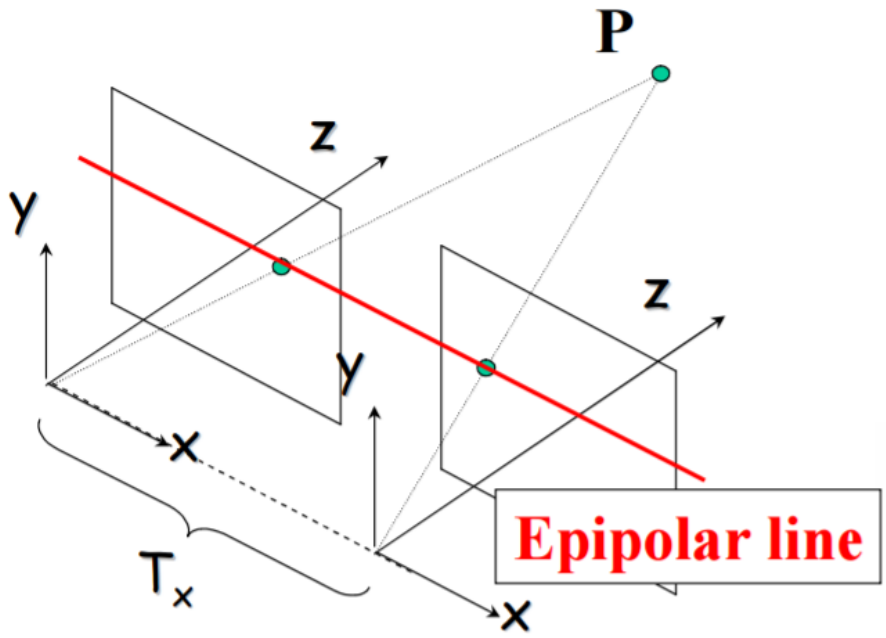


Figure 19: Horizontal Epipolar Lines [5]

A pictorial representation of the process of stereo image rectification is shown in Figure 20 below [15]. This process is achieved using a 3x3 matrix coordinate transform based on parameters obtained from the external calibration process.

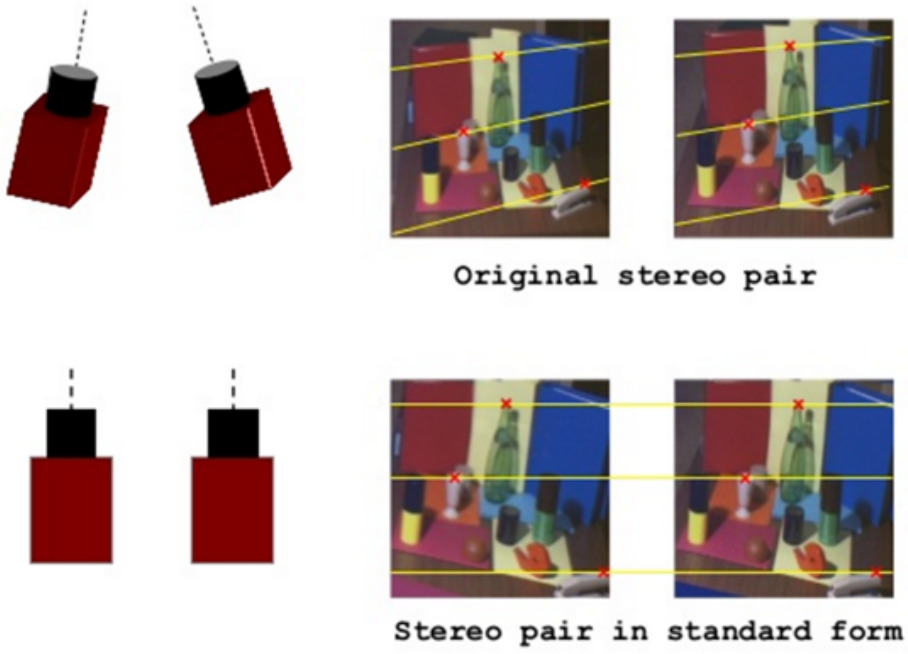


Figure 20: Stereo Image Rectification [15]

After a given pair of images has been rectified, it is then possible to perform the Sum of Absolute Differences on the given image pair in order to extract depth information.

4.8.2 Sum of Absolute Differences

The method used in our disparity algorithm implementation is known as the Sum of Absolute Differences, or SAD. SAD is a common digital image processing technique used to measure the similarity between blocks of image data. In the case of our stereo camera interface, a SAD algorithm is used to search along epioplar lines in the right image for pixel blocks that match a template block selected from the left camera image. This process is performed using 7x7 pixel search blocks over 20 pixel horizontal ranges, and is repeated throughout the image. The expression for the sum of absolute differences is shown in Equation 6 below.

$$SAD = \sum_{x} \sum_{y} |template - block| \tag{6}$$

A visual representation of the Sum of Absolute differences is shown in Figure 21, with the

top image showing the left image template block, and the middle image showing the right image search window in relation to the location of the template block. Below both images is a visual representation of the Sum of Absolute Differences between the template block and the current search block, outlined in white. In the case of the current search, the template and search blocks are relatively different, resulting in a high SAD value.

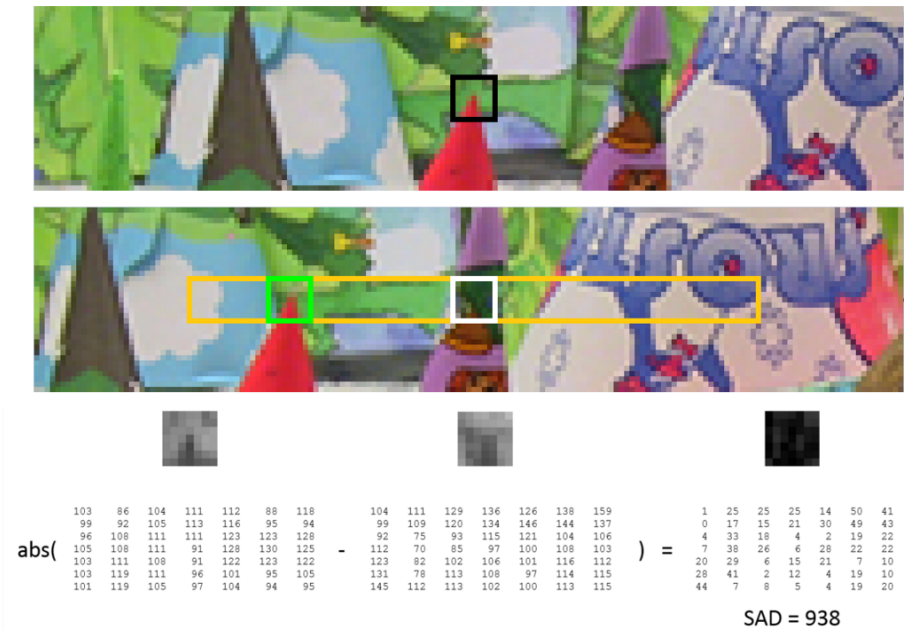


Figure 21: Sum of Absolute Differences [16]

Since the disparity algorithm used in this implementation calculates the sum of absolute differences for multiple search blocks, the resulting SAD values for each search block can be compared to find the location of the most similar matching block in the search image. Due to the nature of the SAD algorithm, lower SAD values indicate higher similarity between the template and search blocks. This comparison is demonstrated in Figure 22 below. In the case of Figure 22, higher match score values for each search block indicate lower SAD values.

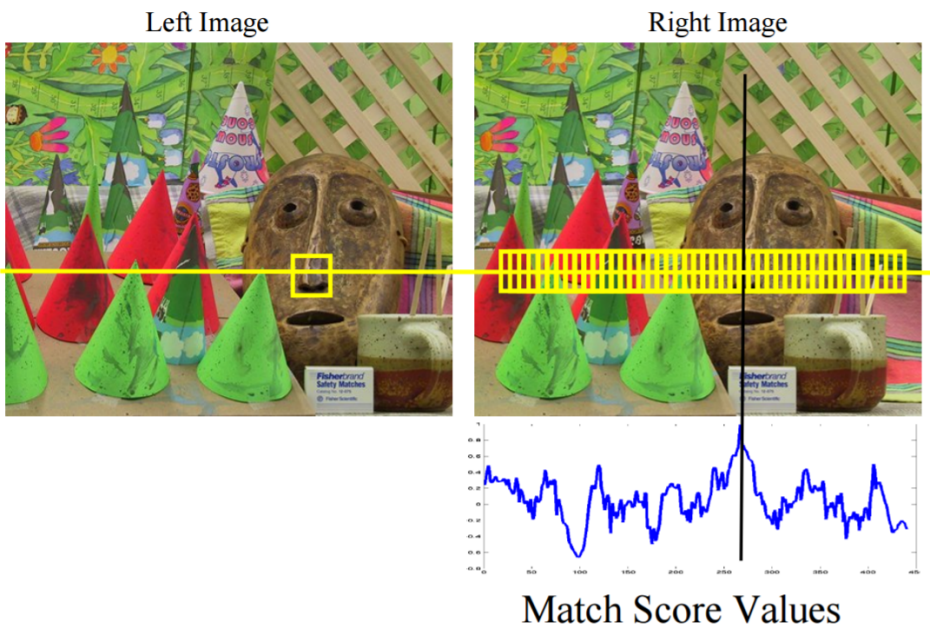


Figure 22: Block Matching Overview [5]

The SAD at multiple search points can be used to estimate the pixel offset between the template block and matching search block based on array index locations, since all SAD values for a single search are stored in a vector. This pixel offset is known as the disparity value for a given template and search block. The disparity d at a given point can be transformed into a units of distance using the focal point f and baseline distance T_x between image sensors as shown in Equation 7 below.

$$depth = Z = \frac{fT_x}{d} \tag{7}$$

Pixel coloration values in a disparity image are based on the distance calculation shown in Equation 7, where each pixel is referenced to the disparity at a given template block's location. An example disparity image created using a given pair of test images is shown in Figure 23 below.

Figure 23: Disparity Algorithm Output

5 Testing and Results

5.1 Single Camera Testing

After obtaining two of the MT9V034 cameras chosen through the process described in Section 4.7.1, several steps were taken to obtain test images from each camera. These steps are outlined in the following sections.

According to the MT9V034 datasheet, each camera module needs to be supplied with an external Master Clock and Output Enable signal in order to operate [18]. A simple Verilog module for the Nexys3 Spartan-6 FPGA board was created in order to supply the camera module with a 24MHz master clock signal, and a switch was used to toggle output enable. With this module implemented, the camera module's default outputs could then be observed. In order to interface the camera module with an FPGA, the breakout board shown in Figure 24 was also created to make the module's pins more easily accessible.

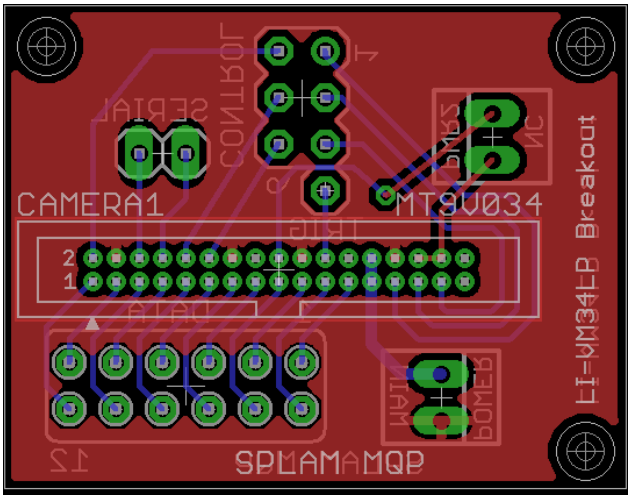


Figure 24: LI-VM34LP Breakout Board

5.1.1 I^{2}C Control

Although the MT9V034 camera control registers are closed source, the previous model's registers are available in the camera module datasheet, and have been found to work with the current model thus far [17]. As a baseline, the camera module was sent a read request at

address 0x00, which should return 0x1324 for the MT9V034 camera module. An oscilloscope screenshot of this request is shown in Figure 25, with the first packet consisting of a request to address 0x00 of device 0x058, and the second packet consisting of the camera's response of 0x1324.

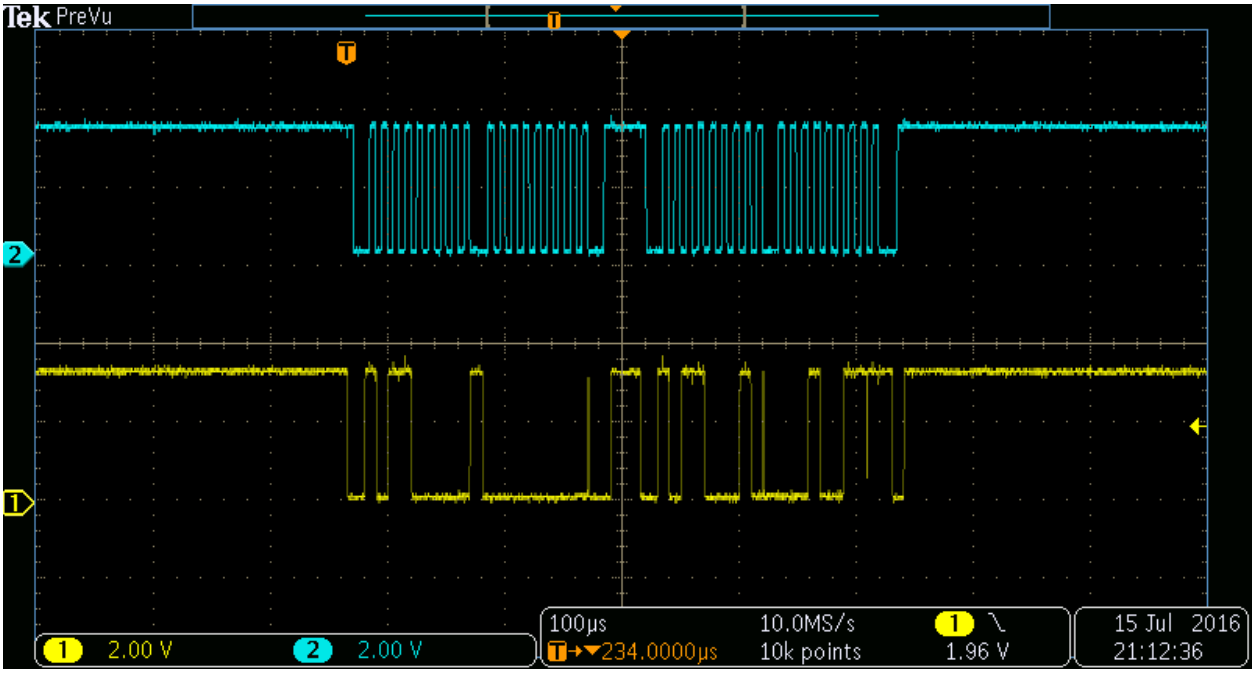


Figure 25: Example I^2C Transfer with Camera

After the camera I^2C was deemed working, the camera control register needed to be modified to put the camera in "snapshot" mode. In this mode, the camera module will no longer continuously take pictures, and will only gather new images when an external trigger is activated. This is the mode that each camera will need to operate in in order to acquire stereo imagery, since a shared trigger line will allow for both cameras to be controlled simultaneously.

According to the previous camera iteration's datasheet, the camera module's operational mode can be set through control register 0x07. By default, this register will be set to a value of 0x0388, which corresponds to master mode with parallel output and simultaneous readout of pixel data enabled [17]. In order to put the camera in trigger mode, the control register needs to be written with value 0x0198, which allows for the same functionality as before

with the exception of having continuous shutter mode replaced with an external trigger. For reference, a table with bit descriptions for the camera control register can be found in Appendix item 7.3 [17].

A button input was then attached to the camera's TRIGGER input line, and the TRIGGER and FRAME_VALID lines were observed on channels one and two of the oscilloscope, as shown in Figure 26. This oscilloscope screenshot can be seen as an example of how the camera is no longer in continuous operation, since FRAME_VALID only asserts itself in response to a TRIGGER input.

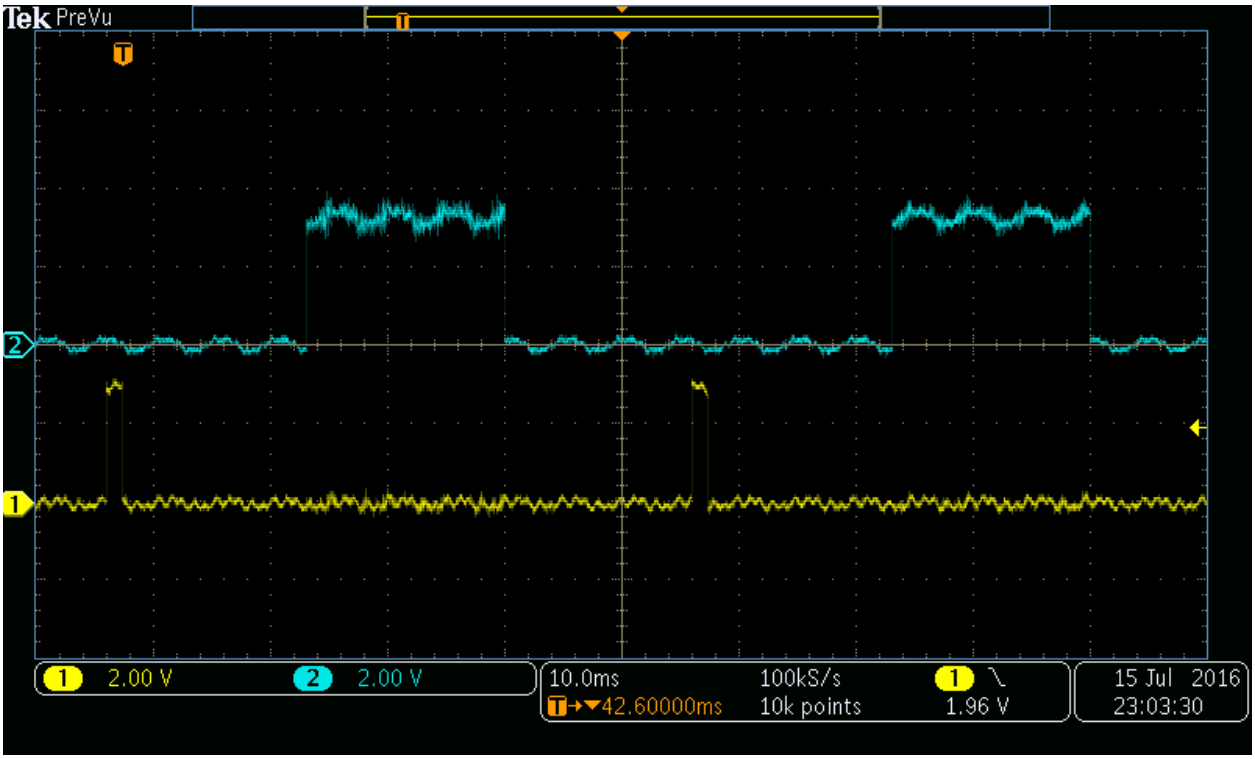


Figure 26: Camera Trigger and FV in Trigger Mode

In order to prevent accidental modification of the camera module's configuration registers, the register lock feature of the camera I^2C bus is also used. By writing 0xDEAD to register 0xFE, it is possible to disable the I^2C bus from being written to. This feature is disabled when the power of the camera module is cycled, or by writing 0xBEEF to the register lock register.

5.1.2 Data Management

After successfully creating a camera control interface and placing the MT9V034 camera module in trigger mode, it was then possible to begin viewing images from the module. With the inclusion of the external FIFO module, it is possible to capture and store an image for future reading, and to read out image data in chunks. Keeping this in mind, the system shown in Figure 27 was created for capturing, storing, and transmitting camera images to a computer for external analysis. In order to reduce development time, an external microcontroller was used for controlling the camera module's I^2C interface and placing the module in trigger mode. Various buttons and switches on the FPGA were then used for controlling the camera output and trigger, allowing for a user to trigger an image for storage on the AL422B FIFO. Once the image has been stored on the FIFO, the FPGA is capable of reading the image line-by-line into an internal buffer. An internal System on Chip (SoC) is used to control FPGA reads from the FIFO into this internal buffer. An image dump will begin when the SoC microcontroller signals to the FPGA to read a new line of pixels into its internal 8 bit by 752 address pixel buffer. The FPGA will then signal to the microcontroller when this buffer has been filled, and the microcontroller will print out the value of each pixel in the buffer to a connected computer over a Universal Asynchronous Reciever/Transmitter (UART) port. When the microcontroller finishes printing out the value of each pixel in the line buffer, it will signal to the FPGA to read in a new line of pixels. This process will repeat for each of the 480 lines of pixels in the image, allowing for the transmission of an entire image's worth of data from FIFO to computer. The Verilog implementation of the top module and line buffer for this interface can be found in Appendix item 7.6.1.

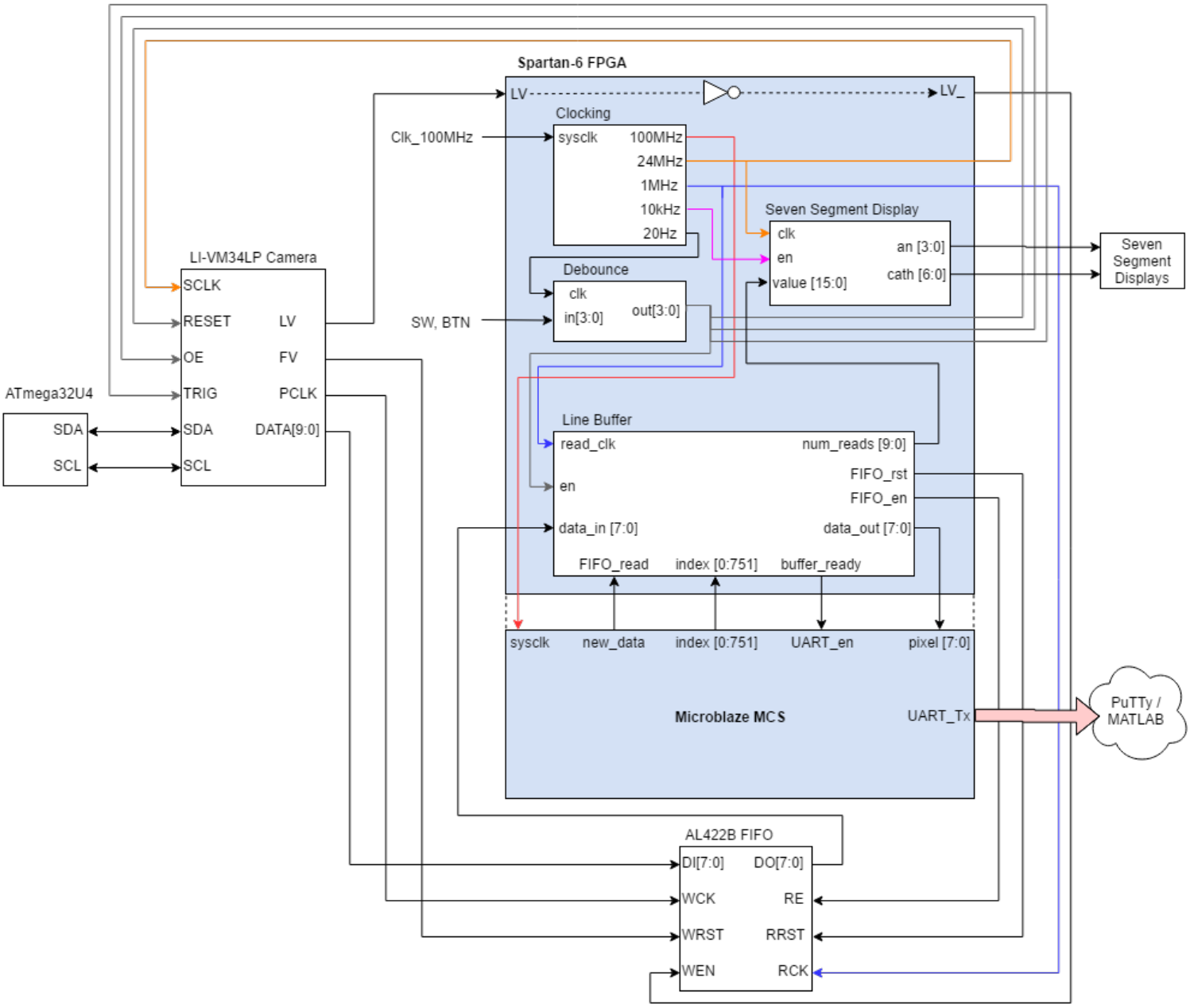


Figure 27: Camera Test System Block Diagram

An example of the transmission of one line of pixel data from the FIFO to the FPGA is shown in Figure 28. The green, purple, blue, and yellow lines in this image represent pixel data, FIFO read enable, read reset, and read clock, respectively. Since the FPGA reads in one line of pixel data at a time, this process will take 752 read clock cycles, as measured in Figure 28. In order to simplify debugging, an internal counter and seven-segment display

controller have also been implemented on the FPGA, and will display a running count of the number of pixel lines that have been read into the FPGA's internal buffer, ranging from 0x0000-0x01E0 (0-480).

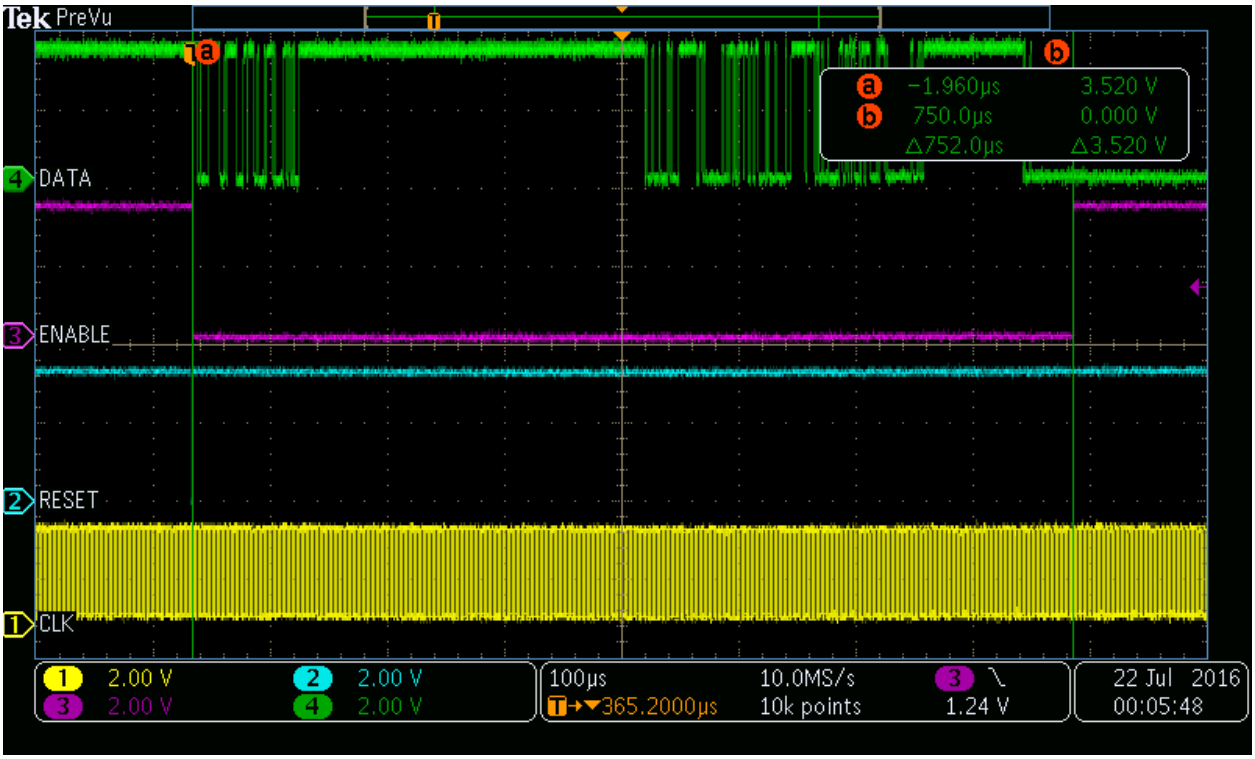


Figure 28: Transferring Line Data from FIFO to FPGA

5.1.3 Transmitting Images Over UART for Analysis

Once the FIFO and FPGA line buffer interfaces were created, the source code found in Appendix item 7.7.1 was implemented on a Microblaze SoC in order to transmit camera line data from the FPGA's internal line buffer over UART. An example of the microcontroller's UART output is shown in Figure 29. The microcontroller will print the value of each pixel followed by a newline and carriage return, starting with the top left pixel in the acquired image.

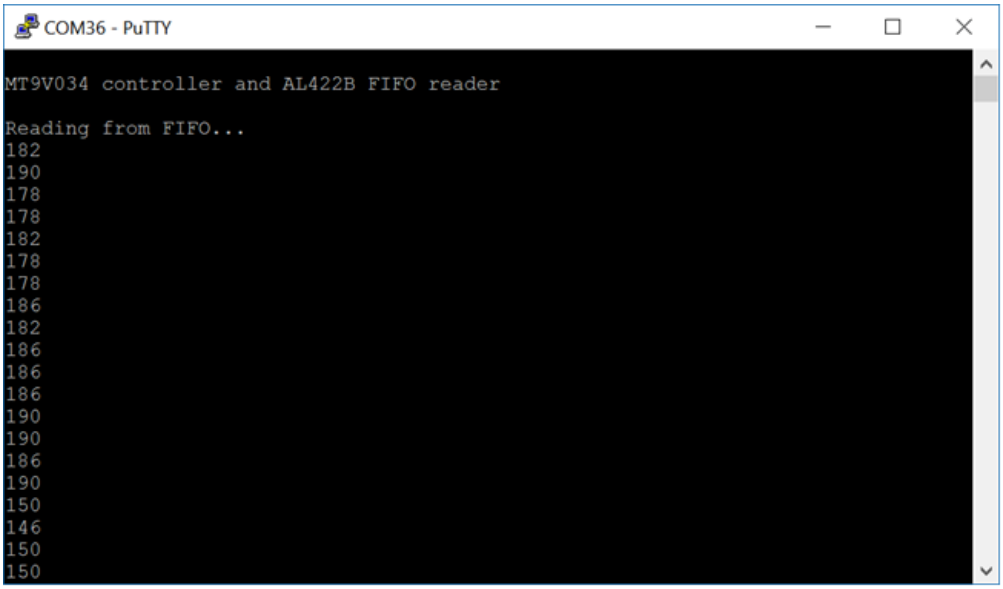


Figure 29: Reading FIFO Data

After the image is received through PuTTy, the MATLAB script found in Appendix item 7.5.1 is used to parse the corresponding logfile into a greyscale image. An example image created through this process is shown in Figure 30. Note that the sub-optimal quality of this image is due to signal interference and degradation in the test setup's long wiring, as shown in Figure 31.

Figure 30: Notebook With Grid and Oscilloscope Leads

Although this system was tested using the Nexys3 (Spartan-6) FPGA board, the use of an external FIFO and little to no platform-specific hardware make it so that it can easily be implemented on any system, including the Zynq family of processors that are used in the final system implementation.

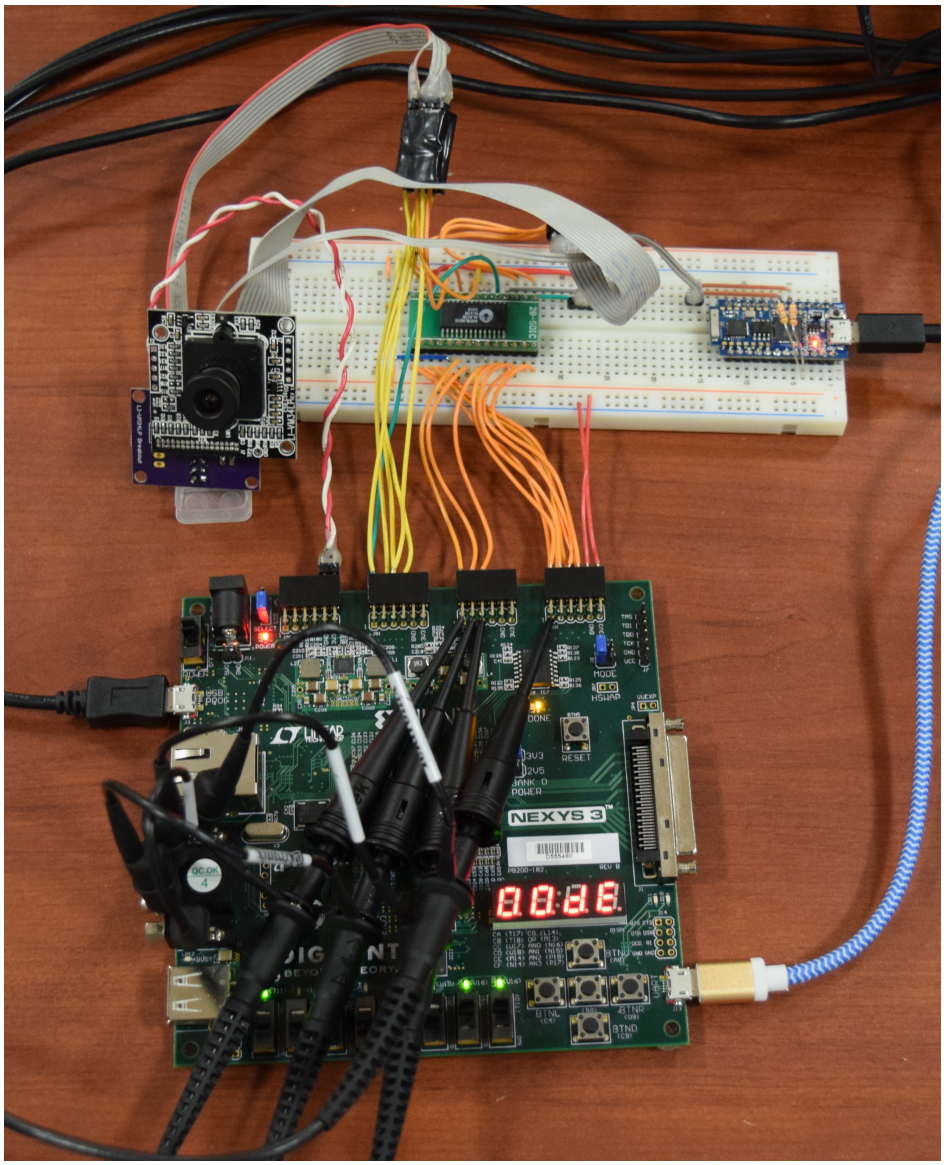


Figure 31: Camera Test Setup

5.2 Final Camera Hardware Implementation

After successfully gathering image data from a single camera module, an interface needed to be created for controlling both cameras at once using the ZedBoard. Interfacing each camera module directly to the ZedBoard's GPIO is not feasible, since the pair would consume every available PMOD pin on the board, leaving no additional pins for the IMU or rangefinder[3]. One solution originally investigated was the use of the ZedBoard's FPGA Mezzanine Card (FMC) connector, since it contains 68 available GPIO pins and would be more than adequate for interfacing the stereo cameras with the board. However, the FMC connector has been configured to provide logic voltage levels of only 1.8 or 2.5 volts without modification to the ZedBoard. Since each camera module is only compatible with 3.3 volt logic, the FMC connector is therefore not feasible for our designs.

This leaves the final option of reducing the overall pin count required by the cameras and interfacing the combined camera setup with the board's PMOD pins. One significant method of reducing the necessary pins required is to include an individual AL422B FIFO per camera. Based on the testing described in the previous section, it has already been determined that these FIFO modules are compatible with the MT9V034 cameras, and are capable of significantly reducing memory requirements on the FPGA. A second major advantage of including these FIFO modules in the camera interface is that their data output lines may be placed in a high-impedance state. This means that the individual data output lines of each FIFO module can be connected in parallel, with a single FIFO driving the lines at a time. Since the bulk of each camera module's required pin count lies in its data lines, the ability to connect these lines in parallel reduces the overall camera GPIO requirements by 8 pins. Since each AL422B FIFO module is capable of being read from at a clock speed of up to 50MHz and the maximum master clock rate of each MT9V034 camera module is 27MHz, the inclusion of the FIFO modules also won't cause a significant decrease in the overall speed of the stereo camera system [3, 18].

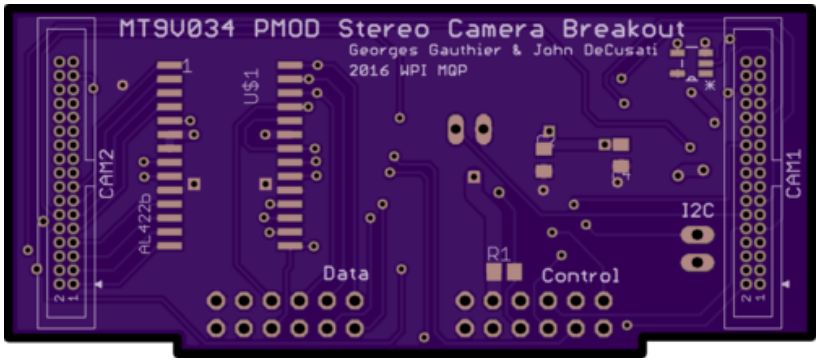
[3] $[2 * (D[9:0] + TRIGGER + OE + RST + SCLK + PCLK + FV + LV)] + SDA + SCL = 36\ pins$

Along with the shared camera data lines between each AL422B FIFO module, it is also possible to connect several other signals in parallel. Since each camera image capture should be triggered at approximately the same time in a stereo imaging setup, it is already desirable to connect both camera TRIGGER lines together. The RST, OE, SDL, SCA, and SCLK lines of each camera module can also be tied together in pairs of two, and the OE lines can simply be held at 3.3 volts. Lastly, since each camera LV signal must be inverted for use with the AL422B FIFOs, a discrete inverter IC may be used to save on FPGA GPIO. Overall, these modifications will save a total of 25 pins, as shown in Equation 8.

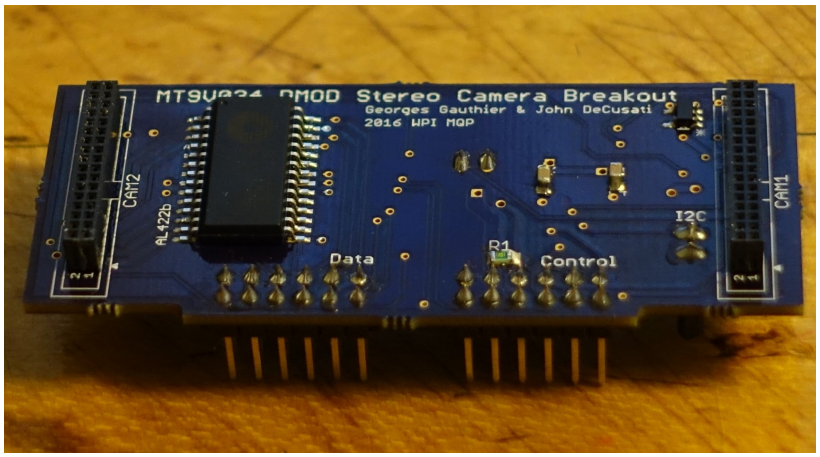
$$\begin{aligned} 36\ Pins - (8\ Data + 4\ truncated\ bits) - (TRIGGER + SCLK + RST) \\ -2 * (OE + PCLK + FV + LV) = 13\ pins\,(!) \end{aligned} \tag{8}$$

Note each FIFO must be controlled individually, requiring an additional Read Reset (RRST) and Read Enable (RE) pin per camera, as well as a shared Read Clock (RCK) line. This brings the total pin count required by the stereo camera setup to 16 pins plus two I2C pins, which is conveniently the number of GPIO available in two PMOD headers. This setup was implemented as shown in Appendix item 7.4, and the final stereo camera breakout board shown in Figure 32 was then created.

A Verilog module was created using a modified version of the MT9V034 camera test code found in Appendix item 7.6.1 and a VGA controller in order to test the stereo camera breakout board. A switch input is used to select one of the two camera modules for image aquisition, and a binned 60x92 pixel set from the center of the camera's image is buffered locally for VGA display. The image is then independently written to the display according to internal VGA timing. This process is repeated at a high rate of speed, allowing for a realtime video stream from the selected camera to be displayed. The assembled stereo breakout board used in this test is shown in Figure 33.



(a) PCB Top



(b) Assembled PCB

Figure 32: Stereo Camera PMOD PCB

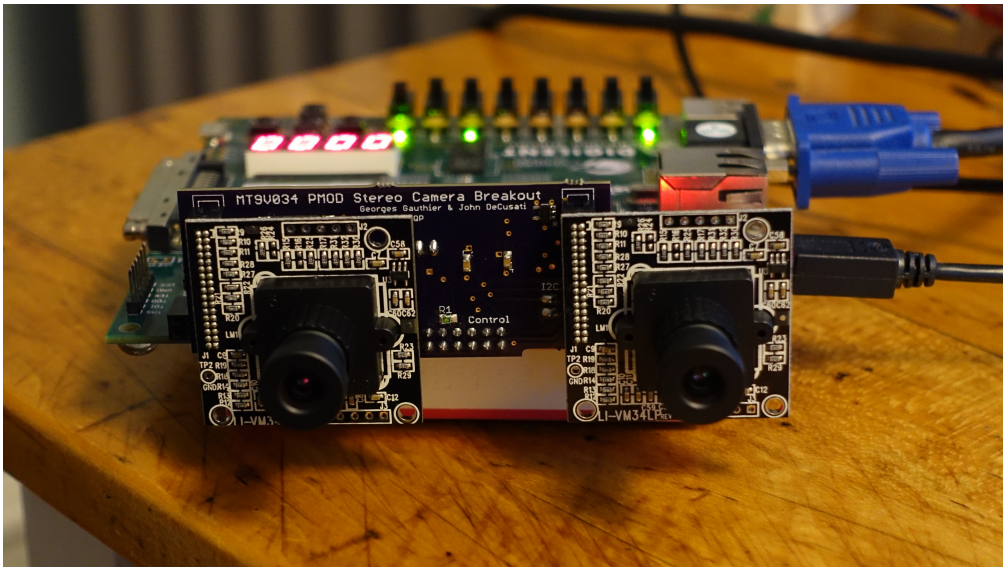


Figure 33: Stereo Camera Breakout Under Test

After attempting to manually focus each camera using the VGA module described above, the code used in Section 5.1.3 was used to transmit image data from the stereo cameras to a computer for further analysis. As you can see from the example image in Figure 34 below, the new stereo camera setup is far less susceptible to data loss in comparison to the previous version.

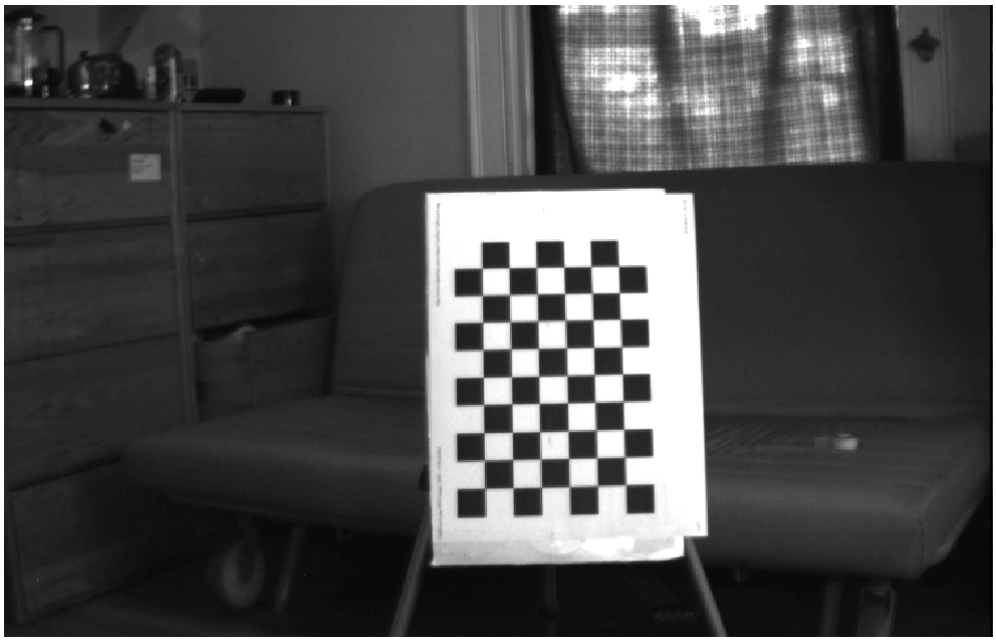


Figure 34: Stereo Camera Breakout Sample Image

For further comparison, please refer back to the test image acquired using the original camera test setup, as shown in Figure 30.

5.2.1 Image Buffering

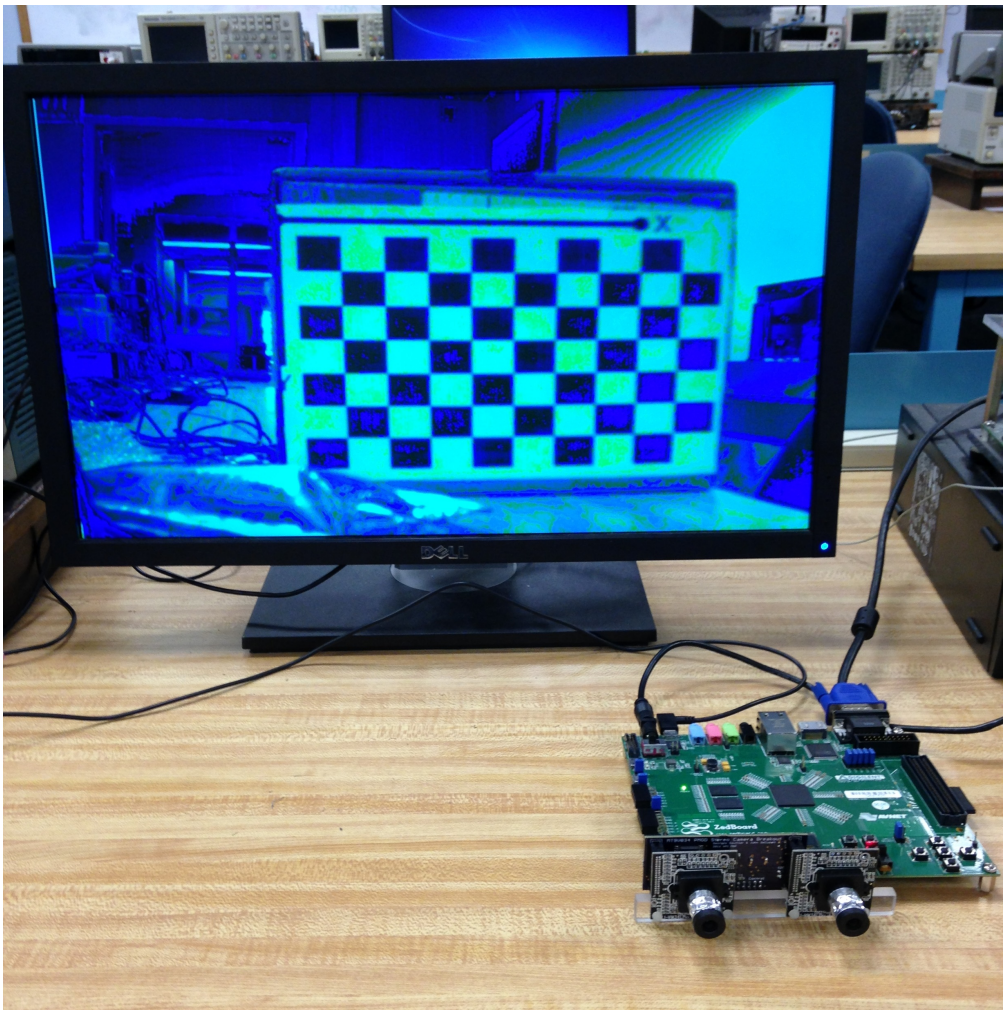


Figure 35: ZedBoard BRAM Camera Test

5.3 Disparity Testing

After verifying that the camera interface was functional, a large portion of time was spent implementing a disparity algorithm that would allow for the extraction of 3D depth information from stereo image data. This algorithm was first implemented in MATLAB, and was then transferred to programmable logic after the algorithm was verified working.

5.3.1 Image Rectification

In order to perform the most accurate block matching as possible on camera image data, it would be ideal to rectify the images as outlined in Section 4.8.1.

- ADD AN EXAMPLE OF THE MATLAB CALIBRATION STUFF (figures showing 3d anaglyph process)

- TALK ABOUT WHY WE DIDNT USE IN FINAL IMP

5.3.2 Matlab Impelmentation

5.3.3 Verilog Test Bench

The original disparity test implementation used closely follows the MATLAB disparity algorithm shown in Appendix item 7.5.2. This algorithm is implemented using a finite state machine with five states, as shown in Figure 42 below. In order to maintain simplicity, the test algorithm has been implemented to operate on 46x30 windowed portions of the input imagery. By default, the disparity module will remain in an idle state until an external enable signal is toggled high using a button input. This will cause the finite state machine to advance to its READ state, and image data for the left and right camera images will be read in from the stereo camera breakout board. After image data has been received, the state machine will then advance to a cyclical set of states used for iterating through each image and calculating disparity.

The disparity module will begin by isolating the template and search blocks from the right and left image data in the finite state machine's separation state. Next, the state machine will advance to its SAD state, and will calculate the sum of absolute differences between the template and search block. This value is placed in a vector that matches the length of the search range. If the vector hasn't been completely filled, indicating that there are more search blocks to compare to the template, the state machine will revert back to the separate state, isolating a new search block from the right camera image. When the SAD

vector is full, the state machine will advance to its finalization state.

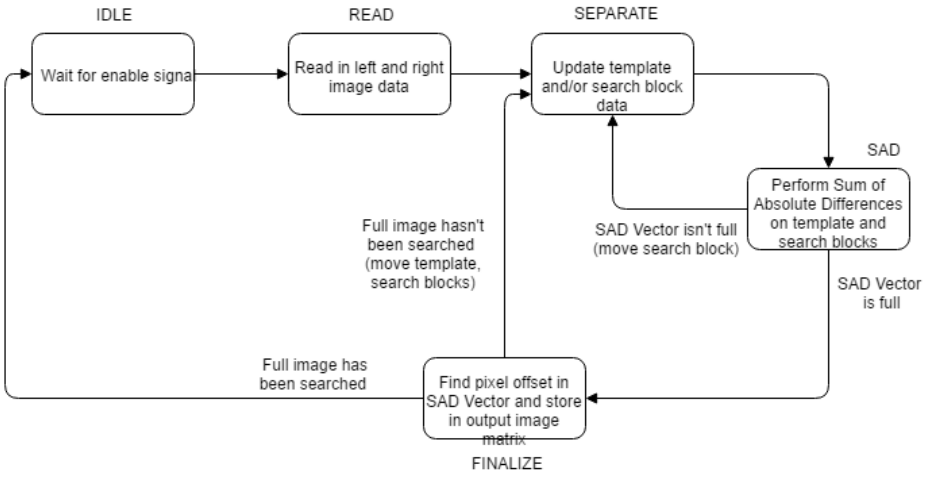


Figure 36: Disparity Test Implementation

The finalization state is used to search through the SAD vector for the lowest value. The index of this value within the SAD vector in reference to the template block location is used to create a disparity value for the given template block location. This value is then converted to a distance using Equation 7, and is stored in the output image location. If the output image hasn't been fully populated with distance values, the state machine will then revert back to the separate state. Otherwise, the state machine will advance to its idle state, and the resulting disparity image can be read for output.

This module was initially tested using a verilog test bench, and was then tested using camera image data and a VGA display controller module, allowing for real-time verification of the algorithm's effectiveness. After testing the initial disparity algorithm, several modifications were made to increase the overall speed and efficiency of the disparity module.

5.3.4 Test Bench Results

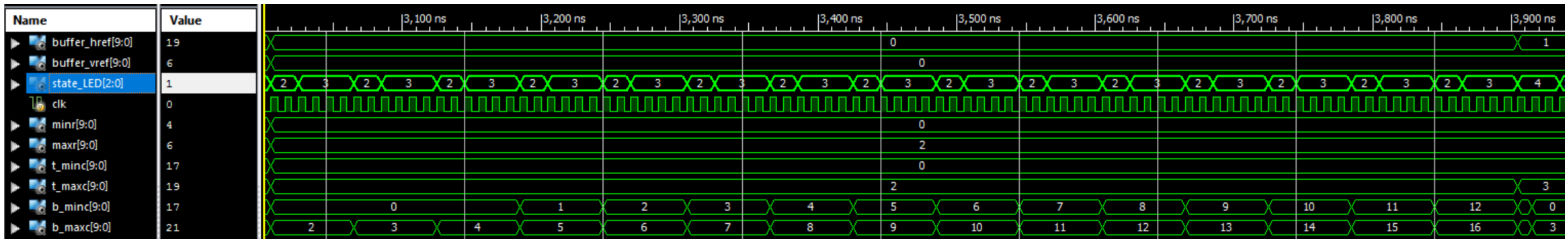


Figure 37: Disparity Search Vector

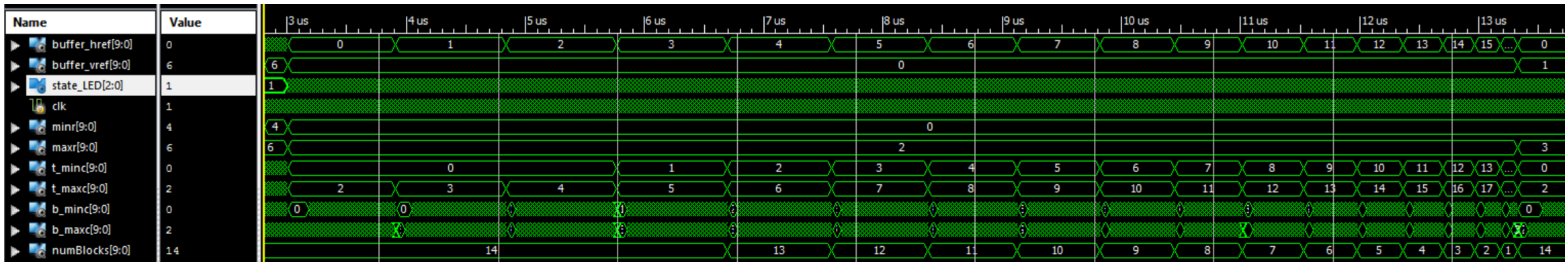


Figure 38: Horizontal Pixel Row Search

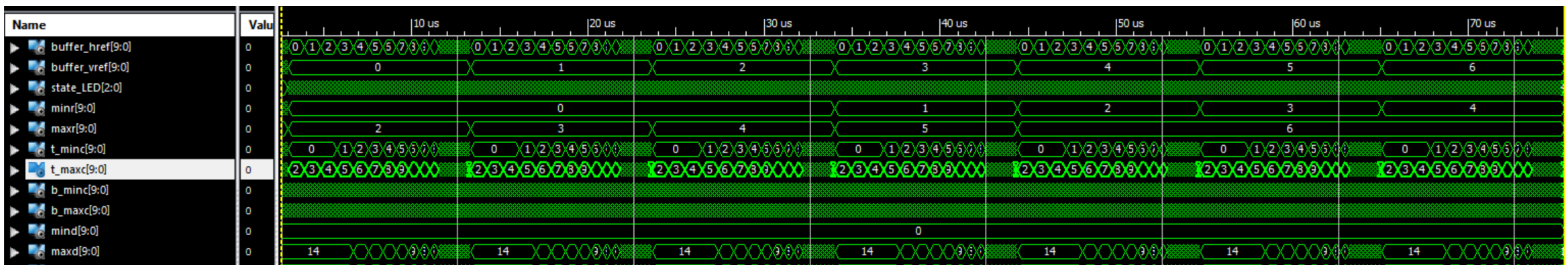


Figure 39: Full Image Search

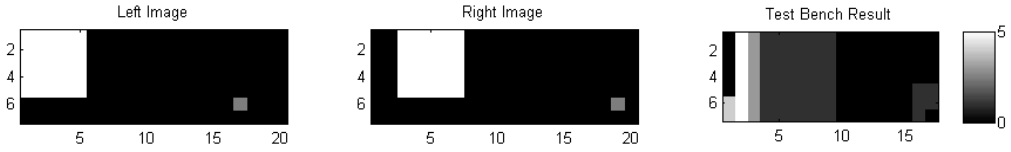
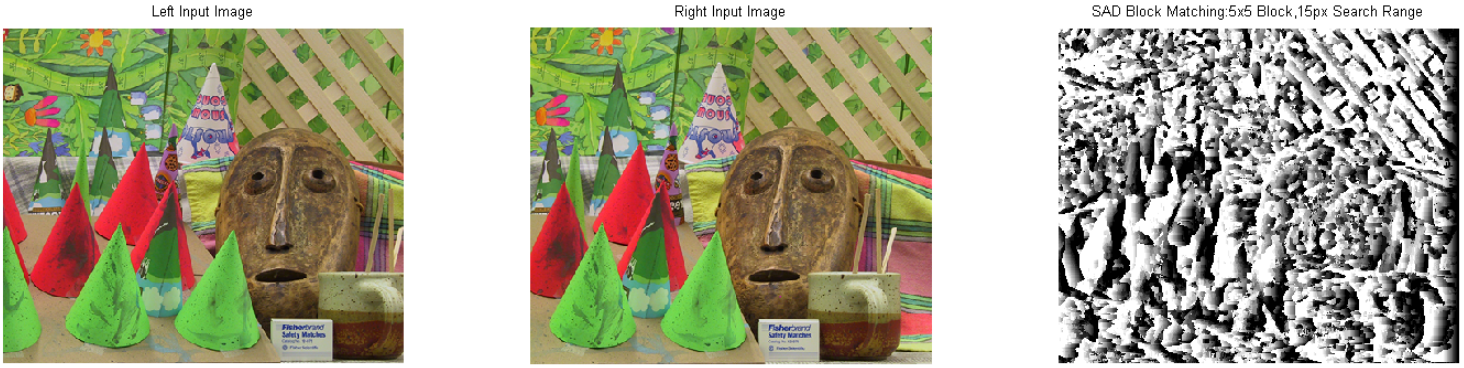
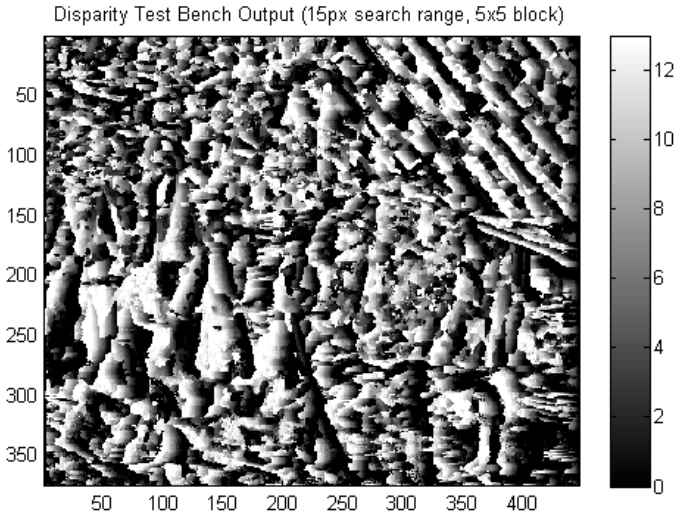


Figure 40: Disparity Test Results



(a) MATLAB Result



(b) Test Bench Result

Figure 41: MATLAB vs. Verilog Test Bench Results

5.3.5 Final Implementation

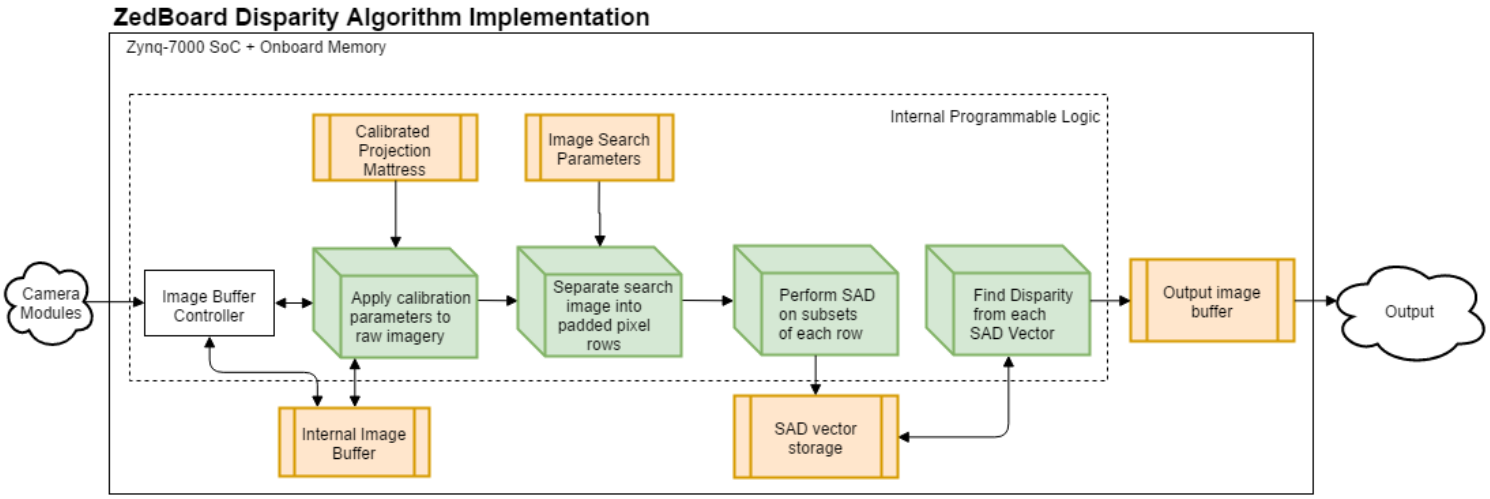


Figure 42: Disparity Final Implementation

GET RID OF CALIBRATION

6 Conclusions

References

[1] ARC Electronics, *RS232 Data Interface*. Available from: http://www.arcelect.com/rs232.htm.

[2] ASCII Table, *ASCII Table - ASCII Character Codes*. 2010. Available from: http://www.asciitable.com.

[3] Averlogic. *AL422B Data Sheets*. 2001. Available from: http://www.frc.ri.cmu.edu/projects/buzzard/mve/HWSpecs-1/Documentation/AL422B_Data_Sheets.pdf.

[4] AVNET, *ZedBoard Hadware User's Guide*. Version 2.2 datasheet, 2014.

[5] Robert Collins, *CSE/EE486 Computer Vision I Fall 2007 Lecture Notes*. Available from: http://www.cse.psu.edu/~rtc12/CSE486/.

[6] Davison, A.J., *Real-Time Simultaneous Localisation and Mapping with a Single Camera*. IEEE Computer Vision, 2003. 2(1).

[7] Dresser, L.H., A.W., *Accelerating Augmented Reality Video Processing with FPGAs*. 2016. Available from: https://www.wpi.edu/Pubs/E-project/Available/E-project-042716-172155/unrestricted/armqp-final-report.pdf.

[8] Hokuyo Automatic Co., *Scanning Laser Range Finder URG-04LX Specifications*. C-42-3389 datasheet, 2012.

[9] Hokuyo Automatic Co., *URG Series Communication Protocol Specification*. C-42-3320-A datasheet, 2012.

[10] Hokuyo Automatic Co., *UrgBenri data viewing tool*. 2014. Available from: https://www.hokuyo-aut.jp/02sensor/07scanner/download/data/UrgBenri.htm.

[11] Leopard Imaging Inc. *LI-VM34LP Camera Board*. 2009. Available from: http://www.leopardimaging.com/uploads/li-vm34lp_v1.1.pdf.

[12] *Serveball*. Available from: http://www.serveball.com/.

[13] Stefano Mattoccia, M.P., *A passive RGBD sensor for accurate and real-time depth sensing self-contained into an FPGA*. in *International Conference on Distributed Smart Cameras*. 2015.

[14] Sparkfun Electronics, *RS-232 vs. TTL Serial Communication*. 2010. Available from: https://www.sparkfun.com/tutorials/215.

[15] Stefano Mattoccia, *Stereo Vision*. Available from: http://www.slideshare.net/DngNguyn43/stereo-vision-42147593.

[16] Chris McCormick, *Stereo Vision Tutorial - Part 1*. Available from: http://mccormickml.com/2014/01/10/stereo-vision-tutorial-part-i/.

[17] On Semiconductor. *MT9V032: 1/3-Inch Wide-VGA CMOS Digital Image Sensor*. 2015. Available from: `http://www.onsemi.com/pub_link/Collateral/MT9V032-D.PDF`.

[18] On Semiconductor. *MT9V034: 1/3-Inch Wide-VGA CMOS Digital Image Sensor*. 2015. Available from: `http://www.onsemi.com/pub_link/Collateral/MT9V034-D.PDF`.

[19] Fatih Porikli, O.T., *Human Body Tracking by Adaptive Background Models and Mean-Shift Analysis*. in *IEEE International Workshop on Performance Evaluation of Tracking and Surveillance*. 2003.

[20] Sebastian Thrun, D.H., David Ferguson, Michael Montemerlo, Rudolph Triebel, and C.B. Wolfram Burgard, Zachary Omohundro, Scott Thayer, William Whittaker. *A System for Volumetric Robotic Mapping of Abandoned Mines*. in *IEEE International Conference on Robotics and Automation*. 2003.

[21] Wolfram MathWorld. *Polar Coordinates*. 2016. Available from: `http://mathworld.wolfram.com/PolarCoordinates.html`.

[22] Xilinx. *MicroBlaze Soft Processor Core*. 2015. Available from: `https://www.xilinx.com/products/design-tools/microblaze.html`.

[23] Xilinx. *Processing System 7 v5.5 LogiCORE IP Product Guide*. in *Vivado Design Suite*. PG082 datasheet, 2015.

7 Appendix

7.1 Useful Resources

This section is intended to serve as complete compilation of all resources gathered throughout D Term 2016 that we believe will be useful as we begin to work on the methodology portion of our project.

- Strother, Daniel. 2011. "Open-Source FPGA Stereo Vision Core Released." https://danstrother.com/2011/06/10/fpga-stereo-vision-core-released/.
- Field, Mike. 2013. "Zedboard OV7670." http://hamsterworks.co.nz/mediawiki/index.php/Zedboard_OV7670.
- "OV7670/OV7671 CMOS VGA CameraChip Implementation Guide." https://www.fer.unizg.hr/_download/repository/OV7670new.pdf.
- "MIPI CSI2-to-CMOS Parallel Sensor Bridge - Lattice Semiconductor." http://www.latticesemi.com/~/media/LatticeSemi/Documents/ReferenceDesigns/JM/MIPICSI2toCMOSPar pdf?document_id=50533.
- "OmniVision Serial Camera Control Bus (SCCB) Functional Specification." http://www.ovt.com/download_document.php?type=document&DID=63".
- Morvan, Yannick. "Multiple-View Depth Estimation." http://www.epixea.com/research/multi-view-coding-thesisse15.html.
- MathWorks. "Depth Estimation From Stereo Video." http://www.mathworks.com/help/vision/examples/depth-estimation-from-stereo-video.html.
- Stefano Mattoccia, Matteo Poggi. 2015. "A passive RGBD sensor for accurate and real-time depth sensing self-contained into an FPGA". International Conference on Distributed Smart Cameras.

- Mattoccia, Stefano. 2013. "Stereo Vision: Algorithms and Applications." http://www.slideshare.net/DngNguyn43/stereo-vision-42147593.

- Szeliski, Richard. 2010. Computer Vision: Algorithms and Applications. New York: Springer.

- Beau Tippets, Dah Jye Lee, Kirt Lillywhite, James Archibald. 2013. "Review of Stereo Vision Algorithms and Their Suitability for Resource-Limited Systems." Journal of Real-Time Image Processing 11 (1). doi: 10.1007/s11554-012-0313-2.

- Jouni Rantakokko, Joakim Rydell, Peter Stromback, Peter Handel, Jonas Callmer, David Tornqvist, Fredrik Gustafsson, Magnus Jobs, Mathias Gruden. 2011. "Accurate and Reliable Soldier and First Responder Indoor Positioning: Multisensor Systems and Cooperative Localization." IEEE Wireless Communications 18 (2):10-18. doi: 10.1109/MWC.2011.5751291.

- Davison, Andrew J. 2003. "Real-Time Simultaneous Localisation and Mapping with a Single Camera." IEEE Computer Vision 2 (1). doi: 10.1109/ICCV.2003.1238654.

- Fatih Porikli, Oncel Tuzel. 2003. "Human Body Tracking by Adaptive Background Models and Mean-Shift Analysis." IEEE International Workshop on Performance Evaluation of Tracking and Surveillance.

- Giovanni Pintore, Enrico Gobbetti. "Effective mobile mapping of multi-room indoor structures." The Visual Computer 30 (6):707-716. doi: 10.1007/s00371-014-0947-0.

- "iRobot 110 FirstLook." iRobot. http://www.irobot.com/$~$/media/Files/Robots/Defense/FirstLook/iRobot-110-FirstLook-Specs.pdf.

7.2 Component Selection

Component	Part Number	Supplier	Cost
FPGA	ZedBoard	Borrowed	N/A
IMU	ADIS16375	Borrowed	N/A
Rangefinder	URG-04LX	Borrowed	N/A
Stereo Cameras†	MT9V034	Mouser	$146

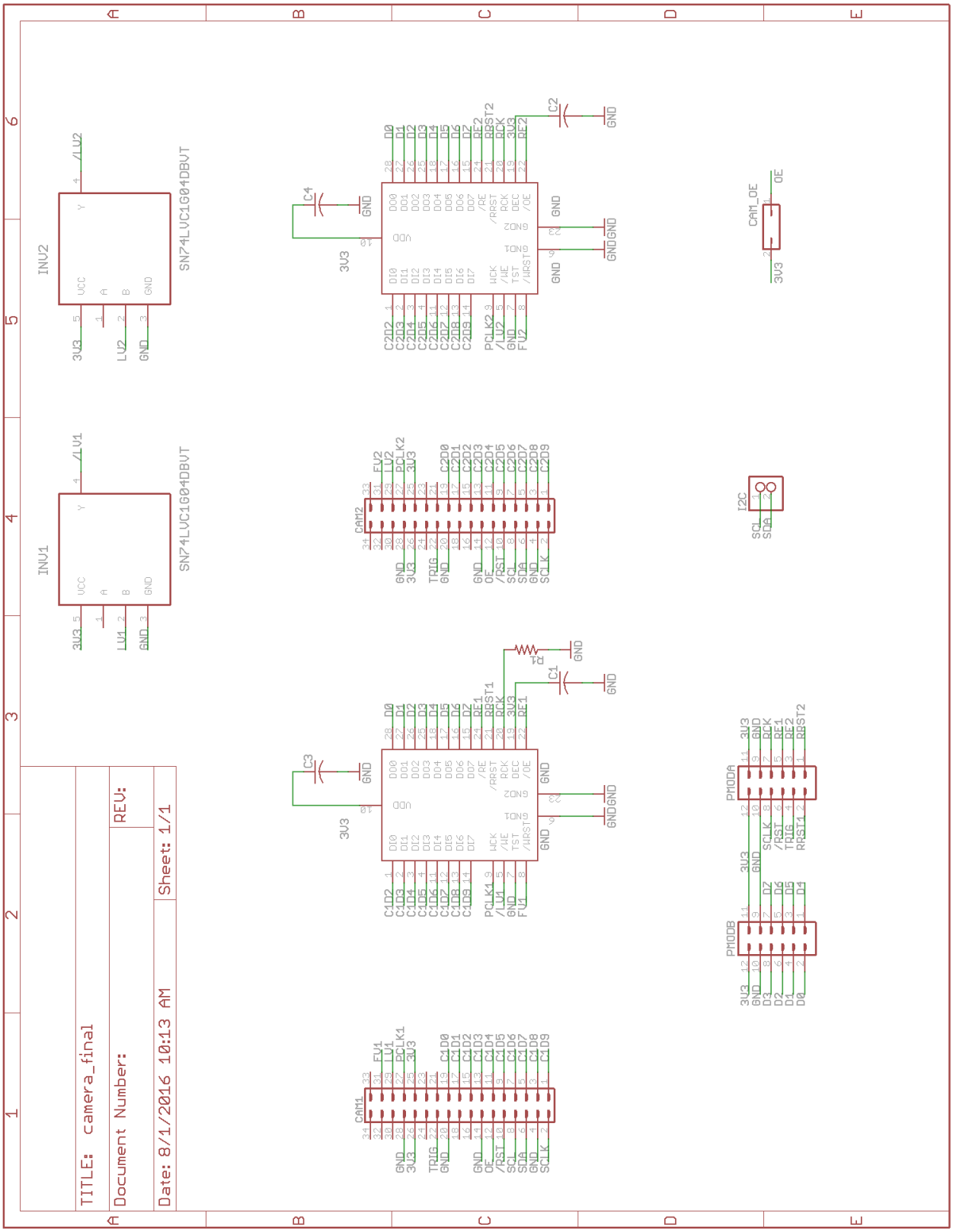
† Note that we originally planned to purchase a flir lepton thermal camera module and accompanying breakout board to support two stereo ov7670 camera modules. After experimenting with the ov7670 camera module on our FPGA board, we began to realize that these camera modules are highly limited due to their low frame rate and poor documentation, and realized that we wanted to search for a different camera module. In addition, at a price of $223 for a thermal camera with an 80x60 degree resolution, 25 degree fov, and 7-9Hz image sample rate, we believe that we are much better off spending our money on better camera modules that will be more usable for our task. For more information see Section 4.7.1.

7.3 Camera Module Control Register

Bit	Bit Name	Bit Description	Default in Hex (Dec)	Shadowed	Legal Values (Dec)	Read/ Write
0x07 (7) Chip Control						
2:0	Scan Mode	0 = Progressive scan. 1 = Not valid. 2 = Two-field Interlaced scan. Even-numbered rows are read first, and followed by odd-numbered rows. 3 = Single-field Interlaced scan. If start address is even number, only even-numbered rows are read out; if start address is odd number, only odd-numbered rows are read out. Effective image size is decreased by half.	0	Y	0, 2, 3	W
3	Sensor Master/ Slave Mode	0 = Slave mode. Initiating exposure and readout is allowed. 1 = Master mode. Sensor generates its own exposure and readout timing according to simultaneous/ sequential mode control bit.	1	Y	0,1	W
4	Sensor Snapshot Mode	0 = Snapshot disabled. 1 = Snapshot mode enabled. The start of frame is triggered by providing a pulse at EXPOSURE pin. Sensor master/slave mode should be set to logic 1 to turn on this mode.	0	Y	0,1	W
5	Stereoscopy Mode	0 = Stereoscopy disabled. Sensor is stand-alone and the PLL generates a 320 MHz (x12) clock. 1 = Stereoscopy enabled. The PLL generates a 480 MHz (x18) clock.	0	Y	0,1	W
6	Stereoscopic Master/Slave mode	0 = Stereoscopic master. 1 = Stereoscopic slave. Stereoscopy mode should be enabled when using this bit.	0	Y	0,1	W
7	Parallel Output Enable	0 = Disable parallel output. DOUT(9:0) are in High-Z. 1= Enable parallel output.	1	Y	0,1	W
8	Simultaneous/ Sequential Mode	0 = Sequential mode. Pixel and column readout takes place only after exposure is complete. 1 = Simultaneous mode. Pixel and column readout takes place in conjunction with exposure.	1	Y	0,1	W

Table obtained from MT9V032 Datasheet [17]

7.4 Stereo Camera Schematic



7.5 Matlab Code

7.5.1 Camera Image Parsing

```
% Camera data parser - reads .log files from PuTTy for MT9V034 test
% Created by Georges Gauthier - 20 July 2016
clear all;
close all;

% prompt for a logfile; open selected file for reading
FILENAME =  uigetfile('*.log','multiselect','off');
fprintf('File %s selected\n\r',FILENAME);
fid = fopen(FILENAME,'r');

image = zeros(480,752); % empty matrix that will hold final image
XPOS = 1; % current pixel X position
YPOS = 1; % current pixel Y position
LINENUM = 1; % number of pixels iterated through
ERRNUM = 0; % number of invalid pixels (happens when Tx is set too fast)

h = waitbar(0,'Parsing image...'); % show a loading bar
c = fgetl(fid); % get rid of 1st line

while 1 % iterate through the log file
  c = fgetl(fid); % get the next line of the file
  if ~ischar(c), break, end
  if length(c) > 0 % if the given line contains valid data...
      image(YPOS,XPOS) = str2num(c)/255; % ... store it as a pixel val
  else % otherwise, throw an error
      ERRNUM = ERRNUM + 1;
      fprintf('Error #%d: Line %d contains no data\n\r',ERRNUM,LINENUM)
  end
  if XPOS<752 % update pixel x position
      XPOS = XPOS + 1;
  else % update pixel y position
      XPOS = 1;
      YPOS = YPOS + 1;
  end
  if mod(LINENUM,36096)==0 % update the loading bar every so often
    waitbar(LINENUM /360000);
  end
  LINENUM = LINENUM + 1; % current line in file (for debug)
end

% display the image...
figure, imshow(image);
% ... also save the image to a file, overwrite if already saved
[path,name,ext] = fileparts(FILENAME);
imgname = strcat(name,'.png');
if (exist(imgname, 'file') == 2)
    fprintf('File for image already exists... overwriting it\n\r')
    delete(imgname);
end
saveas(gcf,imgname);
fprintf('Saved figure to image %s\n\r',imgname);

% close the file and waitbar before exit
fclose(fid);
close(h)
```

7.5.2 Disparity Algorithm Implementation

```
% The following code was adapted from a Mathworks example available here:
% http://www.mathworks.com/help/vision/examples/stereo-vision.html
%
% Original Revision by Chris McCormick
% http://mccormickml.com/2014/01/10/stereo-vision-tutorial-part-i/
%
% Modified by Georges Gauthier - glgauthier@wpi.edu
%
% This script will compute the disparity map for the image 'right.png' by
% correlating it to 'left.png' using basic block matching

clear all;
close all;

% Set to 1 to use 'Cones' Dataset
% Set to 0 to use your own image data (lines 26-29)
EXAMPLE_DATA = 1;

% Load the stereo images.
if (EXAMPLE_DATA == 0)
    load('I1Rect.mat');
    leftI = I1Rect;
    load('I2Rect.mat');
    rightI = I2Rect;
else
    left = imread('left.png');
    right = imread('right.png');
    leftI = mean(left, 3);
    rightI = mean(right, 3);
end

% DbasicSubpixel will hold the result of the block matching.
DbasicSubpixel = zeros(size(leftI), 'single');

% The disparity range defines how many pixels away from the block's location
% in the first image to search for a matching block in the other image.
% 50 appears to be a good value for the 450x375 images from the "Cones"
% dataset.
disparityRange = 50;

% Define the size of the blocks for block matching.
halfBlockSize = 5;
blockSize = 2 * halfBlockSize + 1;

% Get the image dimensions.
[imgHeight, imgWidth] = size(leftI);

% Create a progress bar
h = waitbar(0,'Loading...');

% For each column 'm' of pixels in the image...
for (m = 1 : imgHeight)

    % Set min/max row bounds for the template and blocks.
    % e.g., for the first row, minr = 1 and maxr = 4
    minr = max(1, m - halfBlockSize);
    maxr = min(imgHeight, m + halfBlockSize);

    % For each row 'n' of pixels in the image...
    for (n = 1 : imgWidth)

        % Set the min/max column bounds for the template.
        % e.g., for the first column, minc = 1 and maxc = 4
        minc = max(1, n - halfBlockSize);
```

```
65          maxc = min(imgWidth, n + halfBlockSize);
66
67          % Define the search boundaries as offsets from the template location.
68          % Limit the search so that we don't go outside of the image.
69          % 'mind' is the the maximum number of pixels we can search to the left.
70          % 'maxd' is the maximum number of pixels we can search to the right.
71          %
72          % In the "Cones" dataset, we only need to search to the right, so mind
73          % is 0.
74          %
75          % For other images which require searching in both directions, set mind
76          % as follows:
77          %   mind = max(-disparityRange, 1 - minc);
78          mind = 0;
79          maxd = min(disparityRange, imgWidth - maxc);
80
81          % Select the block from the right image to use as the template.
82          template = rightI(minr:maxr, minc:maxc);
83
84          % Get the number of blocks in this search.
85          numBlocks = maxd - mind + 1;
86
87          % Create a vector to hold the block differences.
88          blockDiffs = zeros(numBlocks, 1);
89
90          % Calculate the difference between the template and each of the blocks.
91          for (i = mind : maxd)
92
93              % Select the block from the left image at the distance 'i'.
94              block = leftI(minr:maxr, (minc + i):(maxc + i));
95
96              % Compute the 1-based index of this block into the 'blockDiffs' vector.
97              blockIndex = i - mind + 1;
98
99              % Take the sum of absolute differences (SAD) between the template
100             % and the block and store the resulting value.
101             blockDiffs(blockIndex, 1) = sum(sum(abs(template - block)));
102         end
103
104         % Sort the SAD values to find the closest match (smallest difference).
105         % Discard the sorted vector (the "~" notation), we just want the list
106         % of indices.
107         [temp, sortedIndeces] = sort(blockDiffs);
108
109         % Get the 1-based index of the closest-matching block.
110         bestMatchIndex = sortedIndeces(1, 1);
111
112         % Convert the 1-based index of this block back into an offset.
113         % This is the final disparity value produced by basic block matching.
114         d = bestMatchIndex + mind - 1;
115
116         % Store the calculated disparity value in the resultant img matrix
117         DbasicSubpixel(m, n) = d;
118     end
119
120     % Update progress bar every 5th row.
121     if (mod(m, 5) == 0)
122         str = sprintf('  Image Row %d / %d (%.0f%%)\n', m, imgHeight, (m / imgHeight) * 100)
                ;
123         waitbar(m/imgHeight,h,str)
124     end
125
126 end
127
128 % close the progress bar
129 close(h);
130
131 % Display the disparity map.
```

```
% Passing an empty matrix as the second argument tells imshow to take the
% minimum and maximum values of the data and map the data range to the
% display colors.
figure, imshow(DbasicSubpixel, []);
axis image;
colorbar;

% Specify the minimum and maximum values in the disparity map so that the
% values can be properly mapped into the full range of colors.
% If you have negative disparity values, this will clip them to 0.
caxis([0 disparityRange]);

% Set the title to display.
title(strcat('SAD Block Matching:  ',num2str(blockSize),'x',...
    num2str(blockSize),' Block, ',num2str(disparityRange),'px Search Range'));

% plot both images in a final output graph
figure,
if (EXAMPLE_DATA == 0)
    subplot(1,3,1), imshow(leftI)
    title('Left Input Image')
    subplot(1,3,3), imshow(rightI)
else
    subplot(1,3,1), imshow(left)
    title('Left Input Image')
    subplot(1,3,3), imshow(right)
end
title('Right Input Image')
subplot(1,3,2), imshow(DbasicSubpixel,[])
title(strcat('SAD Block Matching: ',num2str(blockSize),'x',...
    num2str(blockSize),' Block, ',num2str(disparityRange),'px Search Range'));
```

7.6 Verilog Code

7.6.1 MT9V034 and Al422b Test Code

Top Module:

```
1  `timescale 1ns / 1ps
2  //////////////////////////////////////////////////////////////////////////////
3  // Created by Georges Gauthier
4  // July 09 2016
5  // Test module for controlling the Leopardboard LI-VM34LP camera breakout
6  //////////////////////////////////////////////////////////////////////////////
7   module mt9v034_top(
8                  input sysclk, // 100MHz fpga clk
9                  input reset, // oddr2 reset
10                 input cam_rst, // button for camera RESET_BAR
11                 input trigger, // button for camera trigger
12                 input SW_cam_oe, // switch for camera output enable
13                 input cam_LV, // line valid in from camera
14                 output LOCKED, // oddr2 LOCKED led
15                 output cam_sysclk, // sysclk out to camera
16                 output cam_reset, // reset_bar out to camera
17                 output cam_trigger, // trigger out to camera
18                 output cam_oe, // output enable out to camera
19                 output i2c_ready, // LED indicator for i2c bus ready
20                 output [6:0] cathodes, // 7seg cathodes
21                 output [3:0] anodes, // 7seg anodes
22                 input MICRO_SW, // SW2, used to trigger a new FIFO dump over UART from the
                       mcs
23                 input mcs_reset, // Microblaze reset
24                 output MICRO_LED0, // LED used to indicate if mcs is reading from FIFO
25                 input [7:0] FIFO_DATA, // DO[7:0] from AL422b fifo
26                 output FIFO_WE, // Write enable to fifo (LV inverted)
27                 output FIFO_OE, // read enable to fifo (active low)
28                 output FIFO_RRST, // read reset to fifo (active low)
29                 output FIFO_RCK, // rck to fifo (1MHz)
30                 output UART_Tx // UART send pin from mcs
31                 );
32
33 wire clk_20Hz_unbuf, clk_20Hz;
34 wire clk_10kHz;
35 wire clk_1MHz, clk_1MHz_unbuf;
36 wire clk_24MHz;
37 wire clk_100MHz;
38
39 // 24MHz clock for driving MT9V034's SYSCLK
40 // 100mhz out for FIFO
41 // note you can't connect sysclk to a dcm and other things
42 dcm CLK_24MHz
43    (
44     .CLK_IN1(sysclk),
45     .CLK_OUT1(clk_100MHz),
46          .CLK_OUT2(clk_24MHz),
47     .RESET(reset),
48     .LOCKED(LOCKED)
49          );
50
51 // further divide the dcm clock to other freqs
52 clk_div clks(
53          .reset(reset), // synchronous reset
54     .clk_24M(clk_24MHz), // 24MHz camera SCLK
55          .clk_fifo(clk_1MHz_unbuf), // 1MHz FIFO RCK
56     .clk_debounce(clk_20Hz_unbuf), // 20Hz clock pulse for debouncing stuff
57          .anodes(clk_10kHz) // 10k 7Seg anode driver
58     );
```

```
59
60  // clock buffer for 1MHz fifo rck
61  BUFG clk_1M (
62        .O(clk_1MHz),
63        .I(clk_1MHz_unbuf)
64  );
65  // clock buffer for 20Hz button debouncing
66  BUFG clk_20H (
67        .O(clk_20Hz),
68        .I(clk_20Hz_unbuf)
69  );
70
71  // forward the camera sysclk out using a dedicated clocking route
72  ODDR2 #(
73        .DDR_ALIGNMENT("NONE"), // Sets output alignment to "NONE", "C0" or "C1"
74        .INIT(1'b0),    // Sets initial state of the Q output to 1'b0 or 1'b1
75        .SRTYPE("SYNC") // Specifies "SYNC" or "ASYNC" set/reset
76     ) clkfwd0 (
77        .Q(cam_sysclk),     // 1-bit DDR output data
78        .C0(clk_24MHz),  // 1-bit clock input
79        .C1(~clk_24MHz), // 1-bit clock input
80        .CE(1'b1),      // 1-bit clock enable input
81        .D0(1'b0), // 1-bit data input (associated with C0)
82        .D1(1'b1), // 1-bit data input (associated with C1)
83        .R(1'b0),   // 1-bit reset input
84        .S(1'b0)   // 1-bit set input
85     );
86
87  // forward the fifo read clk out using a dedicated clocking route
88  ODDR2 #(
89        .DDR_ALIGNMENT("NONE"), // Sets output alignment to "NONE", "C0" or "C1"
90        .INIT(1'b0),    // Sets initial state of the Q output to 1'b0 or 1'b1
91        .SRTYPE("SYNC") // Specifies "SYNC" or "ASYNC" set/reset
92     ) clkfwd1 (
93        .Q(FIFO_RCK),     // 1-bit DDR output data
94        .C0(clk_1MHz),  // 1-bit clock input
95        .C1(~clk_1MHz), // 1-bit clock input
96        .CE(1'b1),      // 1-bit clock enable input
97        .D0(1'b0), // 1-bit data input (associated with C0)
98        .D1(1'b1), // 1-bit data input (associated with C1)
99        .R(1'b0),   // 1-bit reset input
100       .S(1'b0)   // 1-bit set input
101    );
102
103 // 7seg display controls
104 wire [15:0] displayVal;
105 seven_seg segs(
106     .values(displayVal), // values to be written to the four seven segment LEDs
107          .CLK(clk_24MHz), // 24MHz clock
108          .en(clk_10kHz), // 10kHz counter enable used for setting the segment refresh rate
109     .cathodes(cathodes),
110     .anodes(anodes)
111     );
112
113 // debounce trigger button input
114 debounce deb(
115     .clk(clk_20Hz),
116     .btn(trigger),
117     .btn_val(cam_trigger)
118     );
119
120 // debounce output enable switch
121 btnlatch sw_oe(
122     .clk(clk_20Hz),
123     .btn(SW_cam_oe),
124     .btn_val(cam_oe)
125     );
126
```

```
// debounce the microblaze input sw
wire read_en;
btnlatch fifoRead_en(
    .clk(clk_20Hz),
    .btn(MICRO_SW),
    .btn_val(read_en)
    );

// camera initialization sequence
reg [11:0] init_count = 12'h000;
always @(posedge clk_24MHz) // cam sysclk before ODDR2
begin
        if (cam_rst) // if cam_rst is pressed, redo the initialization sequence
                init_count <= 12'h000;
        else if(init_count < 2500) // keep cam_rst asserted for at least 20 cam_sysclk
            cycles - I use 30 since it's the minimum time for the i2c bus to be ready
                init_count <= init_count + 1'b1;
end
assign cam_reset = (init_count >= 20);
assign i2c_ready = (init_count >= 30);

// assert/de-assert RE and WE ~0.1mS after power on
wire fifo_rden;
assign FIFO_OE = fifo_rden;
assign FIFO_WE = ~cam_LV;

// Microblaze MCS for reading from local buffer/Tx over UART
wire fifo_read_en, fifo_reset; // tell fpga to put new data in the FIFO
wire [7:0] pixelVal; // value of a camera pixel from fpga line buffer -> microblaze
wire [9:0] pixelPos; // pixel position (0-751) on a line, from microblaze -> fpga line
    buffer
microblaze_mcs mcs_0 (
  .Clk(clk_100MHz), // input Clk
  .Reset(mcs_reset), // input Reset
  .UART_Tx(UART_Tx), // output UART_Tx
  .GPO1({fifo_read_en,
                 fifo_reset,
                 MICRO_LED0}), // output [3 : 0] GPO1
  .GPO2(pixelPos),
  .GPI1(pixelVal), // pixel data from FIFO/FPGA buffer
  .GPI2({read_en,mcs_read_en}) // sw1
);

// Buffer for storing a line of pixels from the FIFO
fifo_read linebuf(
        .reset_pointer(fifo_reset),
        .get_data(fifo_read_en), // from microblaze (sent to trigger new read from FIFO to
            FPGA buffer)
        .pixel_addr(pixelPos), // from microblaze, 0-751
        .fifo_data(FIFO_DATA), // 8 bit data in from fifo
        .fifo_rck(clk_1MHz), // 1MHz clock signal generated by FPGA
        .fifo_rrst(FIFO_RRST), // fifo read reset (reset read addr pointer to 0)
        .fifo_oe(fifo_rden), // fifo output enable (allow for addr pointer to increment)
        .buffer_ready(mcs_read_en),
        .pixel_value(pixelVal), // 8-bit pixel value from internal buffer
        .current_line(displayVal)
   );

endmodule
```

Local Data Buffer:

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Module for reading from the AL422b FIFO and storing pixel line data in a
// local buffer.
```

```
5  //////////////////////////////////////////////////////////////////////
6  module fifo_read(
7          input reset_pointer, // from microblaze, signal to assert fifo_rrst
8          input get_data, // from microblaze (sent to trigger new read from FIFO to FPGA
               buffer)
9          input [9:0] pixel_addr, // from microblaze, 0-751
10         input [7:0] fifo_data, // 8 bit data in from fifo
11         input fifo_rck, // 1MHz clock signal generated by FPGA
12         output reg fifo_rrst, // fifo read reset (reset read addr pointer to 0)
13         output reg fifo_oe, // fifo output enable (allow for addr pointer to increment)
14         output reg buffer_ready, // to microblaze, signal that buffer is ready to read from
15         output [7:0] pixel_value, // 8-bit value from internal buffer
16         output [15:0] current_line // value to seven segment displays
17     );
18
19 parameter [1:0] ready = 2'b00;
20 parameter [1:0] read = 2'b01;
21 parameter [1:0] done = 2'b10;
22 parameter [1:0] init = 2'b11;
23
24 reg [1:0] state = ready;
25 reg [1:0] prev_state, next_state = ready;
26
27 reg [7:0] pixel_line [0:751]; // implemented in BRAM
28 reg [9:0] pixel = 10'b00_0000_0000;
29 reg [15:0] num_lines = 16'h0000;
30
31 always @(posedge fifo_rck)
32         state <= next_state;
33
34 always @(state,get_data,pixel)
35         case(state)
36                 ready:
37                         begin
38                                 if(get_data)
39                                         next_state = init;
40                                 else
41                                         next_state = ready;
42
43                                 prev_state = ready;
44                         end
45                 init:
46                         begin
47                                 next_state = read;
48                                 prev_state = init;
49                         end
50                 read:
51                         begin
52                                 if(pixel == 751)
53                                         next_state = done;
54                                 else
55                                         next_state = read;
56
57                                 prev_state = read;
58                         end
59                 done:
60                         begin
61                                 next_state = ready;
62                                 prev_state = done;
63                         end
64         endcase
65
66 always @(posedge fifo_rck)
67 begin
68                 if(reset_pointer)
69                         begin
70                         fifo_rrst <= 1'b0;
71                         num_lines <= 16'h0000;
```

```
72                      end
73              else if(state==ready) // allow for MCS to read from pixel_line
74                      begin
75                      //pixel_value [7:0] <= pixel_line[pixel_addr];
76                      fifo_rrst <= 1'b1; // make sure read addr doesn't get reset
77                      end
78              else if(state == init) // prepare to read new data from the AL422 into
                    pixel_line
79                      begin
80                      pixel <= 10'b00_0000_000;
81                      num_lines <= num_lines + 1'b1;
82                      buffer_ready <= 1'b0;
83                      fifo_oe <= 1'b0; // allow for read pointer to increment
84                      end
85              else if(state == read) // read data in from the AL422
86                      begin
87                      if(next_state == done)
88                              fifo_oe <= 1'b1; // turn off read enable
89                      if(prev_state != init) // one cycle delay between init and valid
                            data
90                      begin
91                              pixel_line[pixel] <= fifo_data;
92                              pixel <= pixel + 1'b1;
93                      end
94                      end
95              else if(state == done)
96                      begin
97                      buffer_ready <= 1'b1;
98                      end
99  end
100
101 // display number of lines written on 7seg display
102 assign current_line = (num_lines);
103 // allow for MCS to read stored pixel line at given addr if state==ready
104 assign pixel_value [7:0] = pixel_line[pixel_addr];
105
106 endmodule
```

7.7 C Code

7.7.1 Camera Image Parsing

```c
/*
 * Source code for printing values from the AL422B FIFO / FPGA Buffer over UART
 */

#include <stdio.h>
#include "platform.h"
#include "xparameters.h"
#include "xiomodule.h"

// GPO1
#define GETDATA (1<<2) // load a new line of pixels into the FPGA buffer
#define RRST (1<<1) // reset to address 0
#define LED (1<<0) // LED indicator
// GPI2
#define SW_READ (1<<1)
#define BUF_READY (1<<0)

void print(char *str);
void _EXFUN(xil_printf, (const char*, ...));


int main()
{
    init_platform();
    int pixel_position = 0,row = 0;;
    u8 data=0x00,GPO1=0x00,GPI2=0x00,swState=0x00,prevState=0x00;

    XIOModule gpi;
    XIOModule gpo;

    // GPI1 = pixel_value(7:0)
    XIOModule_Initialize(&gpi, XPAR_IOMODULE_0_DEVICE_ID);
    XIOModule_Start(&gpi);

    // GPO1 = (GETDATA)|(RRST)|(LED)
    XIOModule_Initialize(&gpo, XPAR_IOMODULE_0_DEVICE_ID);
    XIOModule_Start(&gpo);

    print("\n\rMT9V034 controller and AL422B FIFO reader\n\r");
    while(1)
    {
        // get switch position
        GPI2 = XIOModule_DiscreteRead(&gpi,2);

        if((GPI2&SW_READ)!=0){
            swState = 1;
            if (row >= 480) GPO1 &=~(LED);
            else GPO1 |= LED;
            GPO1 &=~(RRST|GETDATA);
            XIOModule_DiscreteWrite(&gpo,1,GPO1);
        }else{
            GPO1 &= ~(RRST|LED|GETDATA);
            XIOModule_DiscreteWrite(&gpo,1,GPO1);
            row = 0;
            swState = 0;
        }

        // code below runs only once based on SW state change
        if (prevState != swState){
            if(swState){
```

```
61                  print("\n\rReading from FIFO...\n\r");
62
63                  GPO1 |= (RRST); // reset FIFO position to 0th index
64                  GPO1 &=~ (GETDATA); // make sure we're not trying to read data
65                  XIOModule_DiscreteWrite(&gpo,1,GPO1);
66                  pixel_position = 0;
67                  GPO1 &=~(RRST);
68                  XIOModule_DiscreteWrite(&gpo,1,GPO1);
69                  GPO1 |= (GETDATA); // make sure we're not trying to read data
70                  XIOModule_DiscreteWrite(&gpo,1,GPO1);
71
72                  u32 pixelsRead = 0;
73
74                  while(row<480){
75                      // make sure read_sw hasn't been turned off
76                      GPI2 = XIOModule_DiscreteRead(&gpi,2);
77                      if ((GPI2&SW_READ)==0) break;
78
79                      u8 i=0;
80                      // check to see if BUF_READY is good to go
81                      GPI2 = XIOModule_DiscreteRead(&gpi,2);
82                      // wait until it is
83                      while((GPI2&BUF_READY)==0){
84                          if(i==0){
85                              i++;
86                              //print("\n\t buffer not ready \n\r");
87                          }
88                          GPI2 = XIOModule_DiscreteRead(&gpi,2);
89                      }
90                      GPO1 &=~ (GETDATA); // make sure we're not trying to read data
91                      XIOModule_DiscreteWrite(&gpo,1,GPO1);
92
93                      while (pixel_position < 752){
94                          // update pixel position for FPGA buffer
95                          XIOModule_DiscreteWrite(&gpo,2,pixel_position);
96
97                          // make sure read_sw hasn't been turned off
98                          GPI2 = XIOModule_DiscreteRead(&gpi,2);
99                          if ((GPI2&SW_READ)==0) break;
100
101                         // read value at pixel position from FPGA buffer
102                         data = XIOModule_DiscreteRead(&gpi,1);
103
104                         //print the value
105                         xil_printf("%d\n\r",data);
106
107                         // increment to the next pixel position
108                         pixel_position++;
109                         pixelsRead ++;
110                     }
111                     // signal to the FPGA that we want more data!
112                     GPO1 |= (GETDATA);
113                     XIOModule_DiscreteWrite(&gpo,1,GPO1);
114                     pixel_position = 0;
115                     row++;
116                     //xil_printf("Row: %d",row);
117
118                 }
119                 //xil_printf("%d Pixels Read by MCS",pixelsRead);
120             } else {
121                 GPO1 &=~(GETDATA);
122                 GPO1 |= RRST;
123                 XIOModule_DiscreteWrite(&gpo,1,GPO1);
124                 print("\n\rReset for new sequence\n\r");
125             }
126         }
127
128         prevState = swState; // update prev switch position
```

```
129     }
130     cleanup_platform();
131     return 0;
132 }
```

7.8 Coefficient Files

7.8.1 LUT Initialization Code for Transformation from Polar to Cartesian

Coefficient File for $0° \leq \theta \leq 45°$

```
; COE initialization file 1.
; 13-bit wide, 129 deep coordinate vector.

memory_initialization_radix = 10
memory_initialization_vector =
4096, 4096, 4096, 4095, 4095, 4094, 4093, 4092, 4091, 4090, 4088, 4087,
   4085, 4083, 4081, 4079, 4076, 4074, 4071, 4068, 4065, 4062, 4059, 4055,
    4052, 4048, 4044, 4040, 4036, 4031, 4027, 4022, 4017, 4012, 4007,
   4002, 3996, 3991, 3985, 3979, 3973, 3967, 3961, 3954, 3948, 3941, 3934,
    3927, 3920, 3912, 3905, 3897, 3889, 3881, 3873, 3865, 3857, 3848,
   3839, 3831, 3822, 3812, 3803, 3794, 3784, 3775, 3765, 3755, 3745, 3734,
    3724, 3713, 3703, 3692, 3681, 3670, 3659, 3647, 3636, 3624, 3612,
   3600, 3588, 3576, 3564, 3551, 3539, 3526, 3513, 3500, 3487, 3474, 3461,
    3447, 3433, 3420, 3406, 3392, 3378, 3363, 3349, 3334, 3320, 3305,
   3290, 3275, 3260, 3244, 3229, 3214, 3198, 3182, 3166, 3150, 3134, 3118,
    3102, 3085, 3068, 3052, 3035, 3018, 3001, 2984, 2967, 2949, 2932,
   2914, 2896
```

Coefficient File for $45° \leq \theta \leq 90°$

```
; COE initialization file.
; 13-bit wide, 129 deep coordinate vector.

memory_initialization_radix = 10
memory_initialization_vector =
2896, 2878, 2861, 2843, 2824, 2806, 2788, 2769, 2751, 2732, 2713, 2694,
   2675, 2656, 2637, 2618, 2598, 2579, 2559, 2540, 2520, 2500, 2480, 2460,
    2440, 2420, 2399, 2379, 2359, 2338, 2317, 2296, 2276, 2255, 2234,
   2213, 2191, 2170, 2149, 2127, 2106, 2084, 2062, 2041, 2019, 1997, 1975,
    1953, 1931, 1909, 1886, 1864, 1842, 1819, 1797, 1774, 1751, 1729,
   1706, 1683, 1660, 1637, 1614, 1591, 1567, 1544, 1521, 1498, 1474, 1451,
    1427, 1404, 1380, 1356, 1332, 1309, 1285, 1261, 1237, 1213, 1189,
   1165, 1141, 1117, 1092, 1068, 1044, 1020, 995, 971, 946, 922, 897, 873,
    848, 824, 799, 774, 750, 725, 700, 675, 651, 626, 601, 576, 551, 526,
   501, 476, 451, 426, 401, 376, 351, 326, 301, 276, 251, 226, 201, 176,
   151, 126, 101, 75, 50, 25, 0
```

7.9 LaTeX Coding Examples

This section isn't intended to remain here, but can serve as an example for how to set things up later on

7.9.1 Figures

Figure 43: A Test Figure

Using the `\ref` command, I'm able to reference Figure 43 by calling `\ref{wpiLogo}`.

7.9.2 Code Snippet

Code snippets can be created by calling `\begin{lstlisting}[style=<Language_Name>]`, inserting all code, and then calling `\end{lstlisting}`. Also call `\singlespacing` before the code snippet and `\doublespacing` after to keep things from getting too big.

Note that you can also use

`\lstinputlisting{src.ext}[style=<Language_Name>,firstline=lineNumber, lastline=lineNumber]`

with reference to source files and listings will import the code for you. Way less work!

```
//verilog code example
always @ (x, y, z)
  x <= y + z;
```

```
%Matlab code example
clear all;
close all;
FILENAME =  uigetfile('*.log','multiselect','off');
fprintf('File %s selected\n\r',FILENAME);
return;
```

```
//C code example
#include "someheader.h"
XIOModule gpo;

void GPOWrite(u8 value){
```

```
6       XIOModule_DiscreteWrite(&gpo,1,value);
7       xil_printf("Wrote %d to GPO\n\r",value);
8       return;
9   }
```

7.9.3 Using the bibliography

All bibliographic references are contained in `bib.tex`. To cite a reference in the paper, use the `\cite` command.

As an example, I can cite *Serveball* at the end of this sentence by calling `\cite{serveball}`.[12]

To cite multiple references, call `\cite{ref1,ref2}`.[12, 19]