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 11, 2017

Advisor:
Professor R. James Duckworth

Contents

List of Figures

Abstract

This project implemented a system for generating detailed maps and imagery of an area in real-time by gathering camera imagery, as well as localization and distance measurements from an IMU and a rangefinder. It relied on a Xilinx Zynq FPGA, as FPGAs offer unique advantages in latency and power consumption, and create a favorable platform for common image processing algorithms. All output data from the system has been fully processed, allowing the data to be viewed without further processing.

Executive Summary

Robotic solutions for remotely observing inaccessible areas through video streaming and localization are becoming a quickly expanding field. Currently, many of said solutions rely on a simple sensor suite consisting of cameras that transmit image data wirelessly for remote viewing by first responders or military personnel. Although these devices are highly useful, there is an opportunity to gain much more information from a similar sensor platform at the expense of more processing power. This tradeoff is usually taken at the side of the user, and a relatively powerful Personal Computer (PC) must be used to extract additional information from sensor data.

This project sought to use similar sensors to those in existing solutions and couple them with the processing power of a Field Programmable Gate Array (FPGA) and System on Chip (SoC). This project leveraged FPGA technology in order to solve the current tradeoffs associated with remote mapping and observation through the creation of a proof of concept sensor suite. FPGAs were a clear candidate for such an implementation, since they are capable of performing the types of high-overhead calculations used in remote mapping through low-latency hardware parallelization. FPGAs also consume several orders of magnitude less power than standard computer processors, are highly cost efficient, and are a realistic solution for remote and battery operated devices. The result of this implementation was a proof of concept Simultaneous Localization and Mapping (SLAM) sensor suite that could serve as a replacement to a simple imaging sensor on a remote robotic platform.

The sensors used in this project included a pair of MT9V034 camera modules mounted on a customized stereo camera printed circuit board. The MT9V034 camera module is a global-shutter monochrome image sensor capable of capturing WVGA imagery at 60 frames per second. A Hokuyo URG-04LX scanning laser rangefinder was also used, and consists of a scanning infrared laser rangefinder with a 240° field of view and 99% distance precision. Using an image processing technique known as disparity mapping, it was possible to convert stereo camera imagery into 3D depth maps that could then be correlated with laser rangefinder

data. In order to geographically reference data, a Digilent PmodNAV Inertial Measurement Unit (IMU) was also used. The FPGA platform used was an Avnet ZedBoard, which contains a Xilinx Zynq7020 All-Programmable SoC. The Zynq family of Xilinx devices consist of a dual-core ARM Cortex A9 processor coupled with Xilinx Artix-7 FPGA fabric.

On the input side of the system, the two stereo cameras were physically connected to two video memory buffers, and the camera controls and video memory buffer data were accessed using the Zynq processor's FPGA fabric. This interface has been implemented using a customized printed circuit board that was created during initial project development. Both the scanning laser rangefinder and IMU were connected directly to the dual-core ARM Cortex A9 processor of the Zynq IC, and were communicated with by using low-level peripheral controls.

The first stage of data processing implemented consisted of a dual image buffer controller used for triggering image captures and for reading stereo camera imagery into local memory on the Zynq7020 processor. A 3D depth estimation algorithm then read in image data from local memory, and calculated the relative offset between objects contained in the stereo image pair. A portion of this data was then stored in local memory for correlation with data from the 2D scanning laser rangefinder. Data from the rangefinder and IMU was simultaneously pre-processed on the ARM Cortex A9 processor, and then passed to the system's programmable logic to undergo further coordinate-axis transformation. Several user output modes were included in the proof of concept implementation, including a raw image stream, 3D depth map, and combined 2D floorplan map.

This project successfully implemented stereo cameras, a scanning laser rangefinder, and an IMU in order to create a real-time 3D depth map in addition to a compass-referenced 2D map of the objects closest to the device. Originally, this project proposal included the addition of human object detection and wireless data transmission for the creation of an all-encompassing sensor suite. However, many difficulties were faced throughout this project, and issues with developing customized camera hardware and rangefinder communications created the need for a revised overall goal that could be met within a single-semester project deadline.

For future work, we recommend incorporating IMU displacement data so that the device can create a more realistic floorplan, compared to our device's radar-esque functionality. For this sensor suite to be utilized as intended, there must also be wireless data transmission for remote user access. In addition, incorporating human detection would be a luxury that could be added to the image processing portion of this project. A human detection algorithm could be implemented and combined with the existing 2D and 3D depth information in order to create an all-encompassing sensor suite. These proposed modifications were quickly found to be out of the scope of this project after beginning development, but would greatly improve this project's relevance and utility as a replacement for existing first responder remote observation products.

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 3D depth information on a given area as both a cost-effective and information-rich alternative to using a simple camera module on a device. Such a product would be able to implement high speed data processing techniques that 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 allow 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 field of image processing and high-speed computing, 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, which could be combined with the ability to detect and localize objects such as human beings. For the scope of this project, we would like to define our desired objective as Real-Time Remote Mapping, and will be focusing on the mapping portion of the proposed system.

A device that is capable of mapping its surroundings using real-time SLAM is applicable to many different fields. The proposed system could be designed as a sensor suite capable of performing these tasks, intended to replace standard video cameras on existing robotic systems. This type of technology allows for people such as first responders to wirelessly traverse dangerous and remote locations in search of people in need. The envisioned sensor suite will be able to provide a 2D floorplan of the area being traversed by the sensor suite, as well as a real-time depth-augmented video feed.

One type of technology that would be useful for performing the high speed data processing necessary for such a device is a Field Programmable Gate Array (FPGA). FPGAs pose several advantages for real-time data processing over standard computing or micro-controller technology, as they have the ability to manipulate digital information in parallel using hardware only. This allows for extremely high-speed performance; FPGA latency is mostly dependent on data inputs, as opposed to software latency which is dependent on task priority.

Although FPGA technology is highly applicable to performing SLAM-like tasks, there are currently few existing commercial products that use FPGAs for this purpose. 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 a real-time operating system (RTOS). This setup implies that data must first be collected by a sensor suite and then transferred to an external computing device that is limited to processing the data serially based on its arrival time. Although this type of setup is acceptable for performing real-time situational awareness analysis, we proposed that an embedded FPGA-based remote mapping 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. Since everything is running in parallel, more data inputs can be added to the system to increase functionality without introducing any latency into the system, as long as enough memory and hardware are 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" [8]. 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.

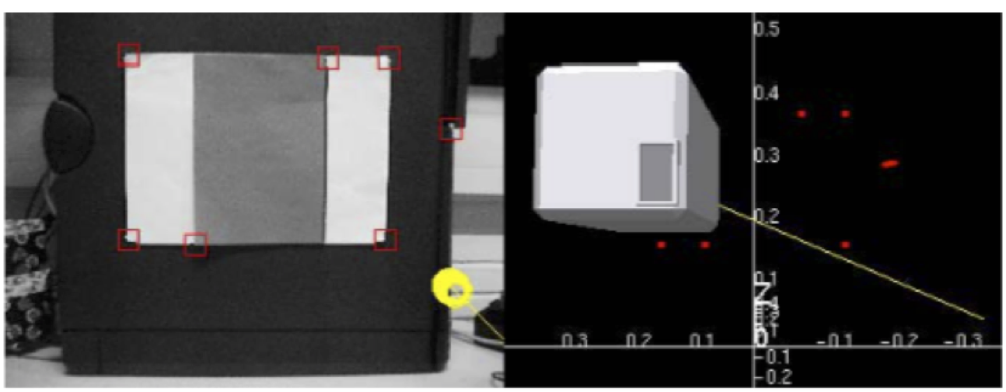


Figure 1: Real-Time SLAM with a Single Camera [8]

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} [17]. 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 [17]

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.

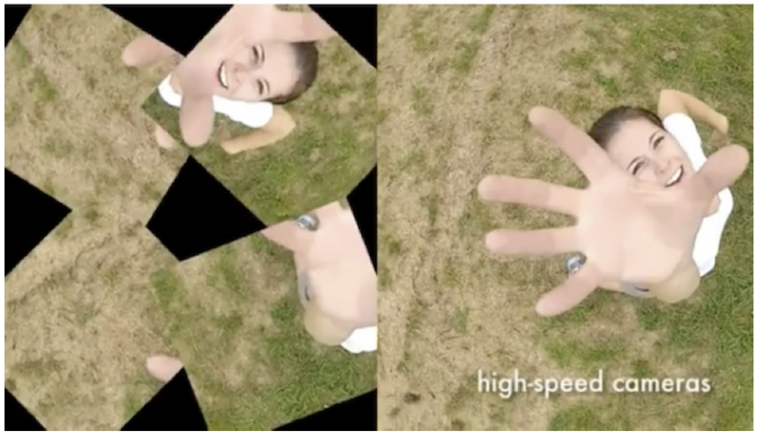


Figure 3: Serveball's Squito Input and Output [17]

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 [18]. 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 is 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.

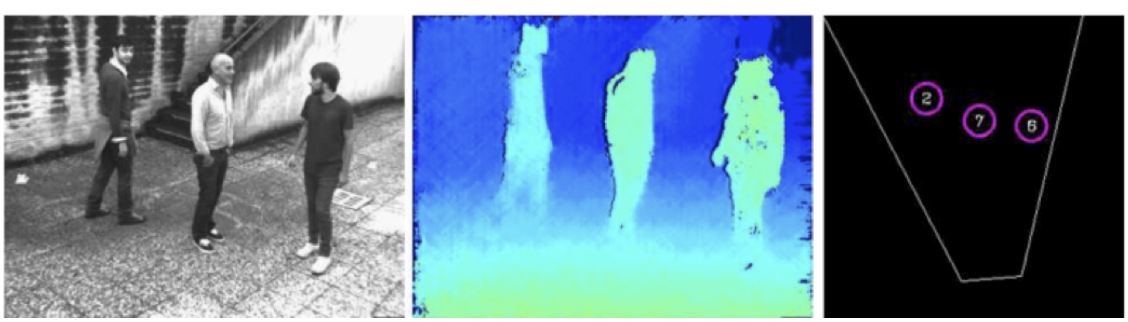


Figure 4: From Left to Right: Original Image, Disparity Map, Object Detection Results [18]

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" [26]. 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 [26]

The proposed device would ideally combine the ideas of the four systems examined. However, due to project time constraints we focused on SLAM-based area mapping, and have recommended the future addition of human detection algorithms. The proposed device would be capable of generating real-time 2D maps of the area it is traversing, as well as a 3D depth map of the device's current field of view.

In order to successfully implement this system, we proposed the creation of a device that would rely on two stereo cameras, a laser rangefinder, and an inertial measurement unit (IMU) as its sensor suite, as shown in Appendix Item B. Little to no comparable existing commercial products were capable of processing their gathered data locally and in real-time. Stereo cameras would 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 would allow for precise base distance readings, and an IMU would be used to localize all gathered data. All of this data was then combined with the disparity maps and image data in order to create flawless localization and mapping. All time-dependent processing required for the device would be mainly done in parallel using hardware on an FPGA, in order to reduce system latency. An overall functional block diagram of our intended implementation is shown in Figure 6 below.

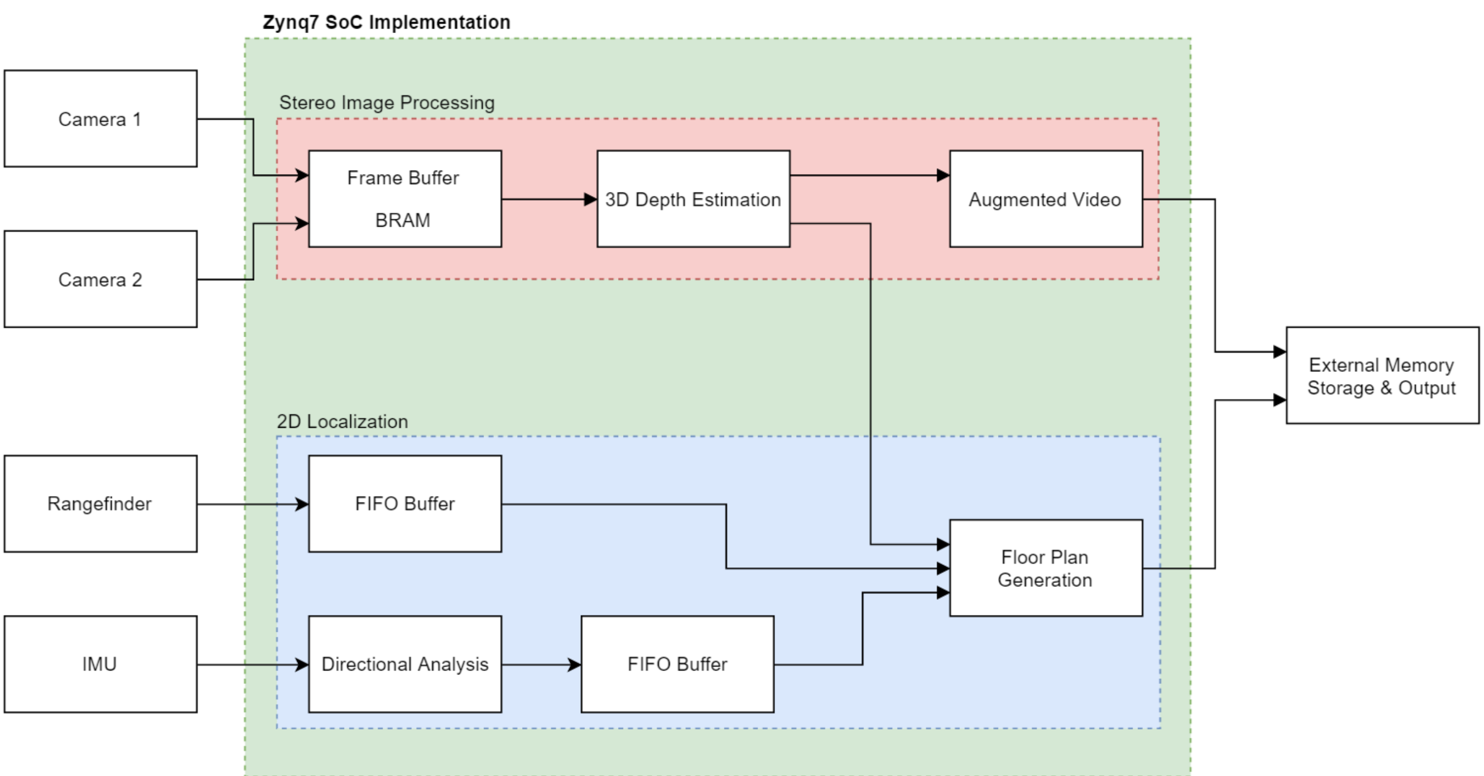


Figure 6: Functional Block Diagram

Most applicable previous camera-based systems have also focused on mapping from a stationary point, or edge detection from a mobile platform. Our project aimed to combine these concepts, by creating a mobile mapping device that would be especially of use in many first responder situations.

As our research has progressed over time, the 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 [27].

As our research progressed, we believed that we could use 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 product detailed maps of the sensor suite's 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, time constraints, and the resources that have been made available to us, in the coming terms we planned to use an electronic scanning laser rangefinder, IMU, and stereo camera pair to generate real-time SLAM video and floorplan information. Although we were also originally planning on including a thermal camera in our sensor suite as well, we 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. We also originally proposed the inclusion of human object detection in our project, but modified our proposal and included it as a future suggestion due to project time constraints.

3 System Design

Overall, the proposed system has been reduced to the functional blocks shown in Figure 7 below. On the input side of the system, the two stereo cameras are physically connected to two video memory buffers, and the camera controls and video memory buffer data are accessed through programmable logic on the Zynq processor. Since the rangefinder and IMU modules each communicate with UART and SPI interfaces, respectively, each are connected directly to the Zynq's dual-core ARM processor. Both sensors may then be communicated with using Xilinx's built-in ARM peripheral drivers, reducing overall implementation time. Note that an I^{2}C controller is also included as a peripheral for the ARM processor, allowing for communication with each camera's control registers.

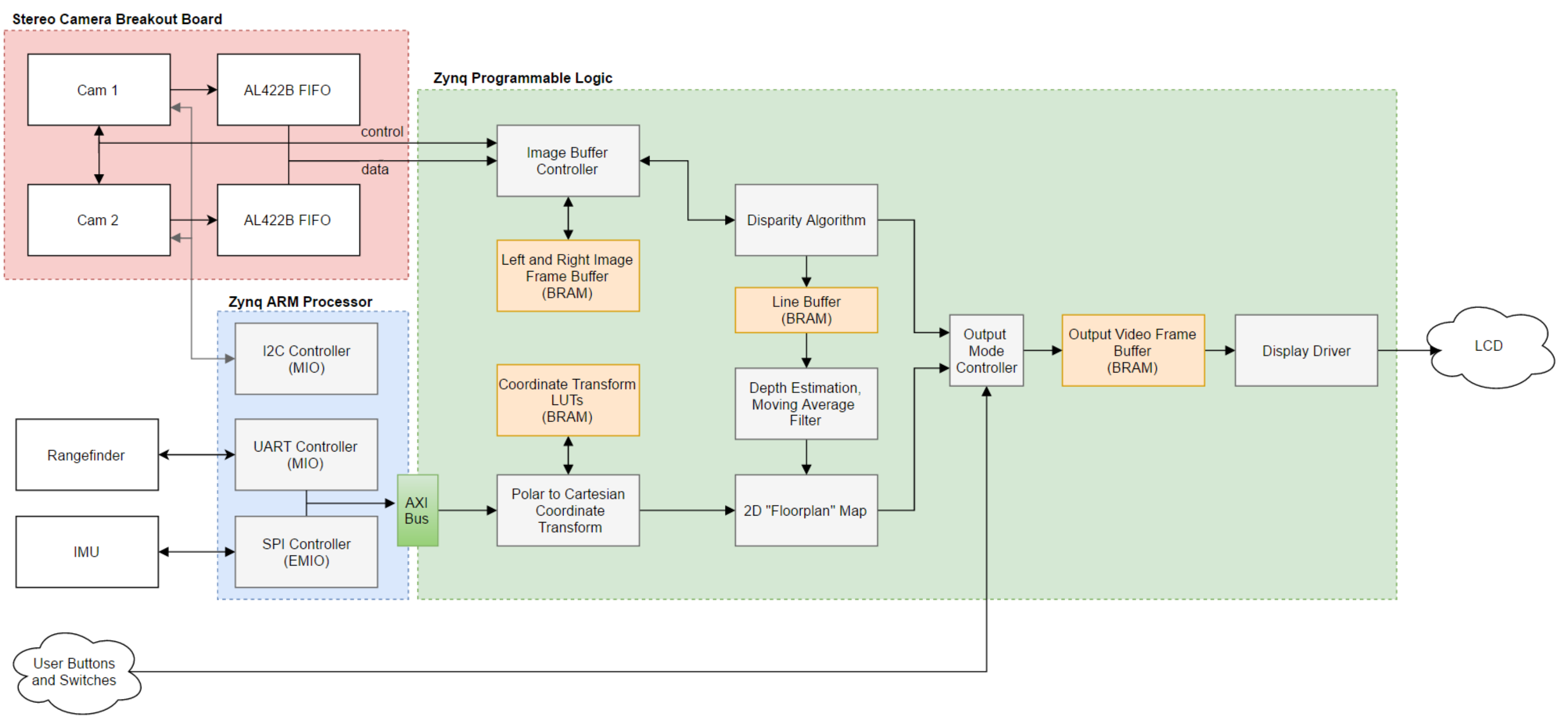


Figure 7: System Block Diagram

In terms of stereo camera data processing, the first stage of programmable logic consists of a dual image buffer controller used for triggering image captures and reading image data into local memory on the ZedBoard. A disparity module then reads in image data from local memory, and calculates the relative offset between objects contained in the stereo image pair. A portion of this data is then stored in local memory for correlation with data from the 2D scanning laser rangefinder. Depending on the position of the ZedBoard's user input switches, the output from the disparity algorithm is also passed to a video frame buffer and displayed via VGA.

In order to correlate disparity data with the 2D depth information from the scanning laser rangefinder, several horizontal lines of pixel data from the disparity algorithm are averaged together. This single line of depth information can be compared to the single line of depth information from the rangefinder. However, due to differences in the field of view of each sensor, the depth information is passed through a moving average filter before being correlated with rangefinder data.

Data from the scanning laser rangefinder and IMU modules is pre-processed in programmable software, allowing for the estimation of sensor rotation relative to the IMU's compass heading. A custom AXI peripheral bus is then used to pass data from the programmable software to the programmable hardware. At this stage, a pair of lookup tables are used to convert the rangefinder data from Polar to Cartesian coordinates, and the modified data is correlated with depth estimation data from the disparity algorithm. Depending on the status of the ZedBoard's user switches, the 2D floorplan map created using the combined sensor data is then passed to the output video buffer for external display.

4 System Implementation

After deciding that the proof of concept sensor suite would rely on a scanning laser rangefinder, IMU, and stereo camera interface, research was then performed in order to determine what specific sensors to use, as well as the operating modes of each chosen sensor. This chapter describes the specific research performed for each sensor, as well as the planned implementation of the overall sensor suite.

4.1 Rangefinder Operation

A rangefinder is a device that estimates the distance of the objects closest to it. Because this sensor suite 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, shown in Figure 8, is a durable, lightweight piece of equipment that has a field of view of 240° and can sense objects up to 4 meters away with an accuracy to within 10 millimeters, which is perfect for our application [13].

Figure 8: URG-04LX Scanning Laser Rangefinder [12]

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 [3].

The rangefinder can be connected to one of the ZedBoard's Peripheral Module (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 [21]. Figure 9 shows a timing diagram of both RS-232 and TTL communication protocols.

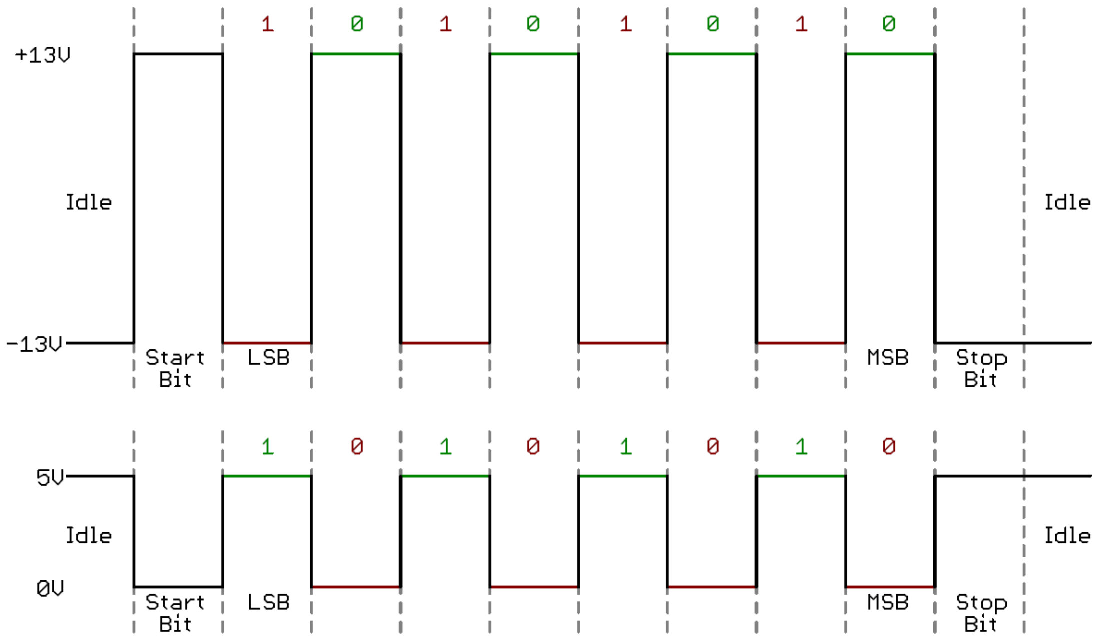


Figure 9: Timing Diagram of RS-232 (top) and TTL Communication Protocols [21]

Since these two serial communication formats have incompatible logic levels, an RS-232 to TTL converter was needed so that the rangefinder can communicate with the ZedBoard. The converter's TTL side was connected to the ZedBoard's Pmod connector, and the RS-232 side was connected to the rangefinder. For ease of connection and testing, the 9-pin DSUB RS-232 connector was connected to an RS-232 breakout board so that the pins were easily accessible. Figure 10 shows the RS-232 to TTL converter attached to the RS-232 breakout board.

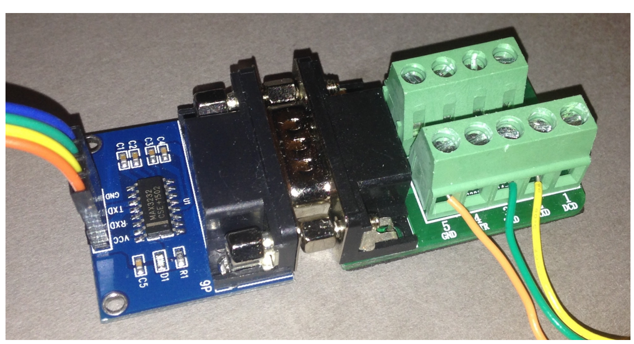


Figure 10: RS-232 to TTL Converter with RS-232 Breakout Board

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 [6, 13]. 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, and 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 was used as a test; as soon as it is received, the rangefinder transmits the device specific information. The laser illumination command was used to turn the laser on and off. The communication speed setting command was used to change the baud rate. The distance data acquisition command was used to request the distance data from the rangefinder [14].

The distance data acquisition command was 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 11 below shows a top-down view of the device field of vision with the steps labeled.

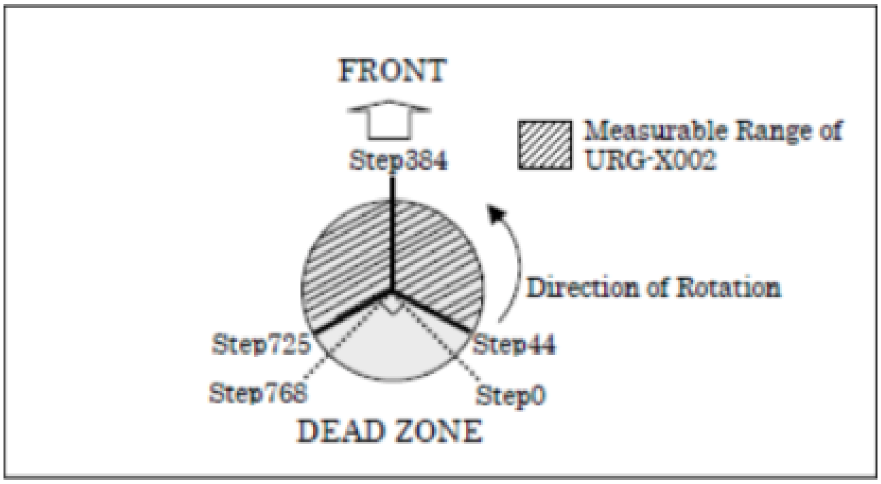


Figure 11: Top-Down View of Rangefinder Field of Vision [14]

For this project, the beginning point was set to '000' and the end point to '768' to obtain the device's maximum coverage of 240°. Note that the device's angular rotation per step was calculated by using Equation 1 below. Accordingly, each step around the rangefinder's field of view corresponds to a change of 0.3515625°.

$$\frac{360°}{1024 \text{ steps}} = 0.3515625° \text{ per step} \tag{1}$$

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 obtained the data acquisition command 'G00076801\n' which was transmitted from the ZedBoard to the rangefinder to request one cycle of data.

4.2 Rangefinder Implementation

With the data acquisition command set, the rangefinder was connected to the ZedBoard. Since the URG-04LX uses UART communication, there were a few reasonable options to create a UART controller on the ZedBoard.

4.2.1 UART Options

The ZedBoard has a few different options for controlling UART. UART can be controlled through linux, through a MicroBlaze soft-core processor, or through the Zynq-7000 Processing System. Running linux only for the purpose of using it as a UART controller is a waste of the ZedBoard's valuable resources, as linux provides extreme capability. The MicroBlaze soft-core processor is 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 [31]. Because of this, we utilized 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 [32]. Despite 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 was used to change the processing system's activated features. Figure 12 shows the processing system customization window.

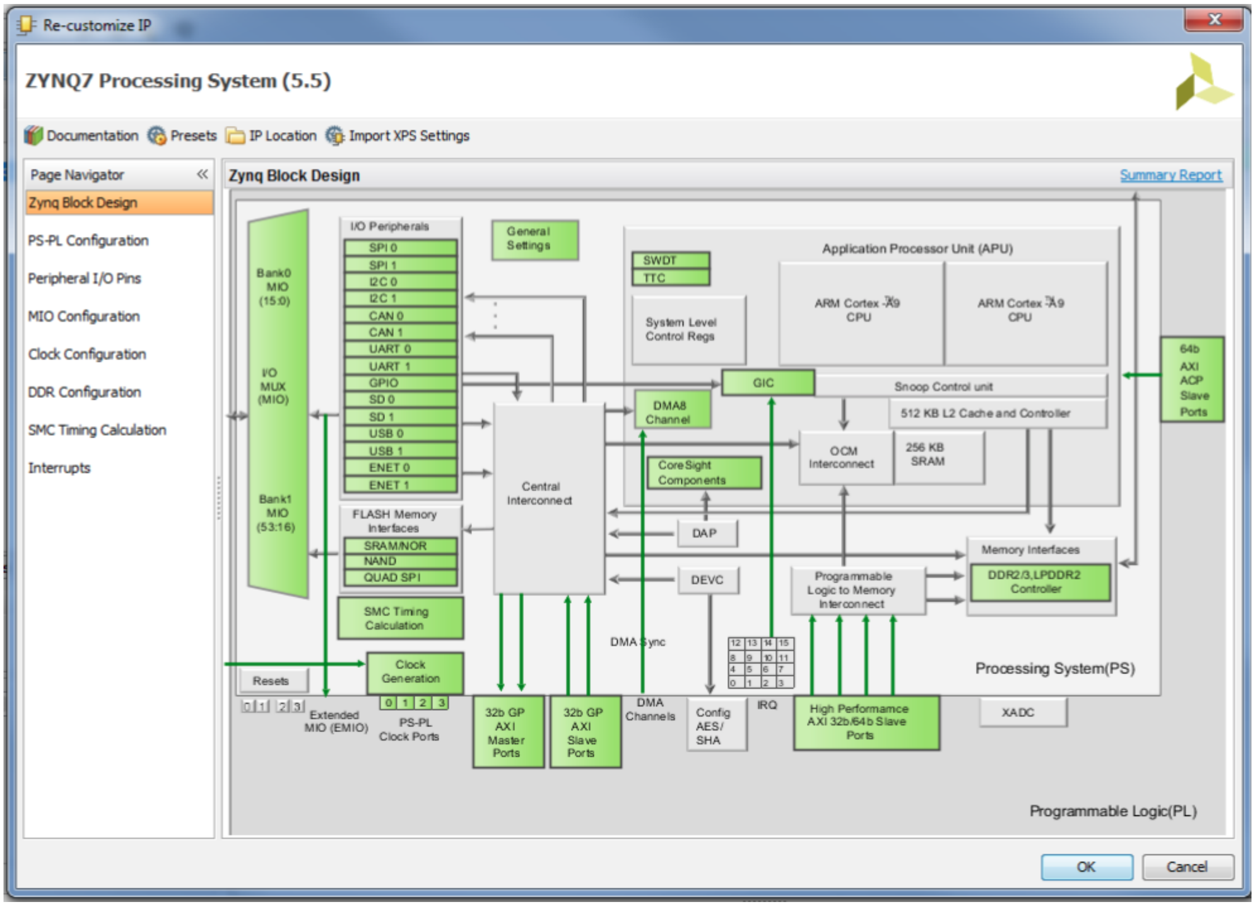


Figure 12: Zynq7 Processing System Customization Window [32]

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 [6]. 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 was configured to match with the rangefinder's default communication speed, 19200 baud [14]. This was done in the processing system's customization window under PS-PL Configuration on the sidebar in Figure 12. Figure 13 shows the PS-PL Configuration window.

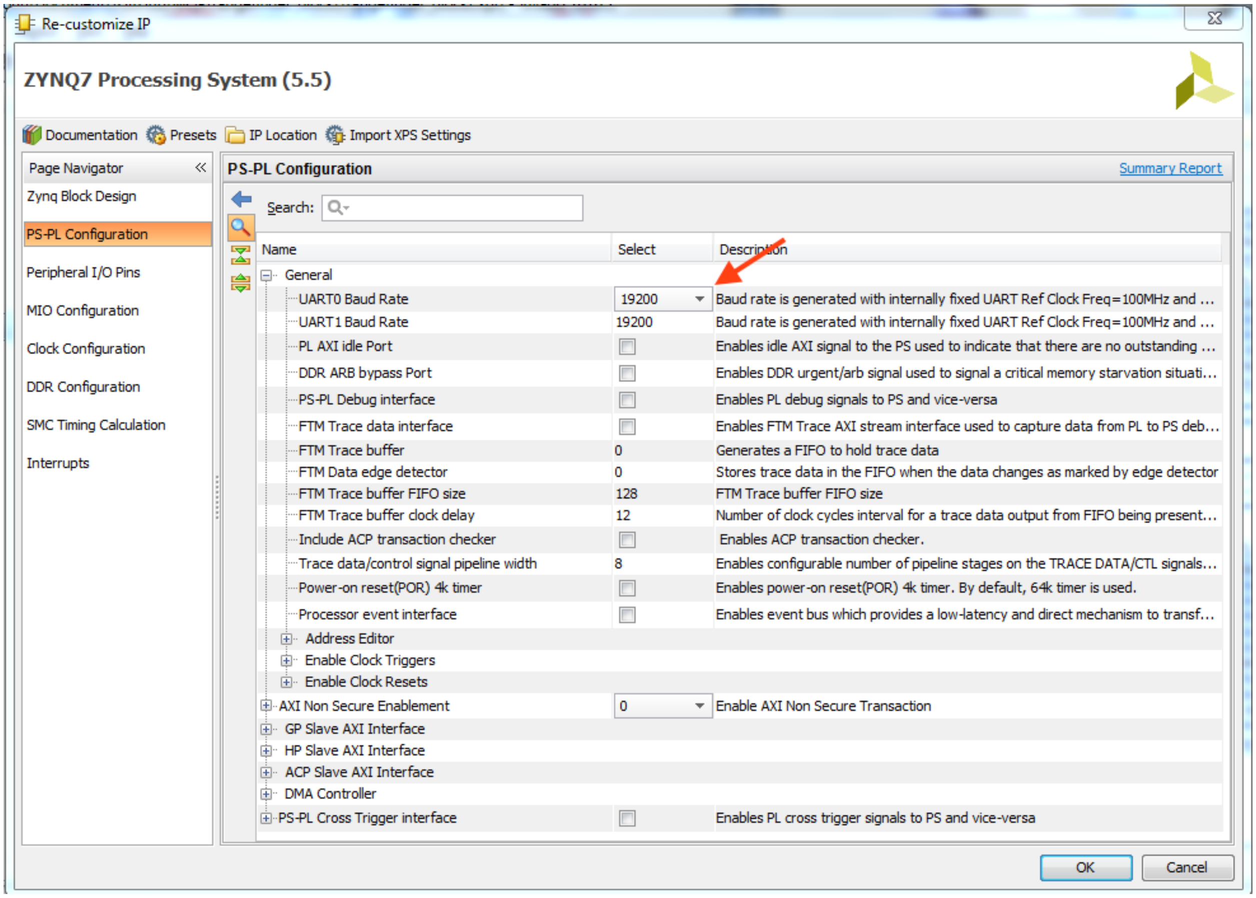


Figure 13: Zynq7 Processing System PS-PL Configuration Window

With the processing system customized in this fashion, the Programmable Logic's configuration is complete.

4.2.3 Designing with the Xilinx SDK

The Programmable Software (PS) is coded in the Xilinx SDK, and was edited in the project through Vivado. To launch the SDK, the design was first exported. This is done by generating a bitstream. The bitstream compiles all the project's customization into a *.bit* file which is used to program the FPGA on the ZedBoard. Generating a bitstream was done in Vivado under the Program and Debug section of the Flow Navigator, as seen in Figure 14.

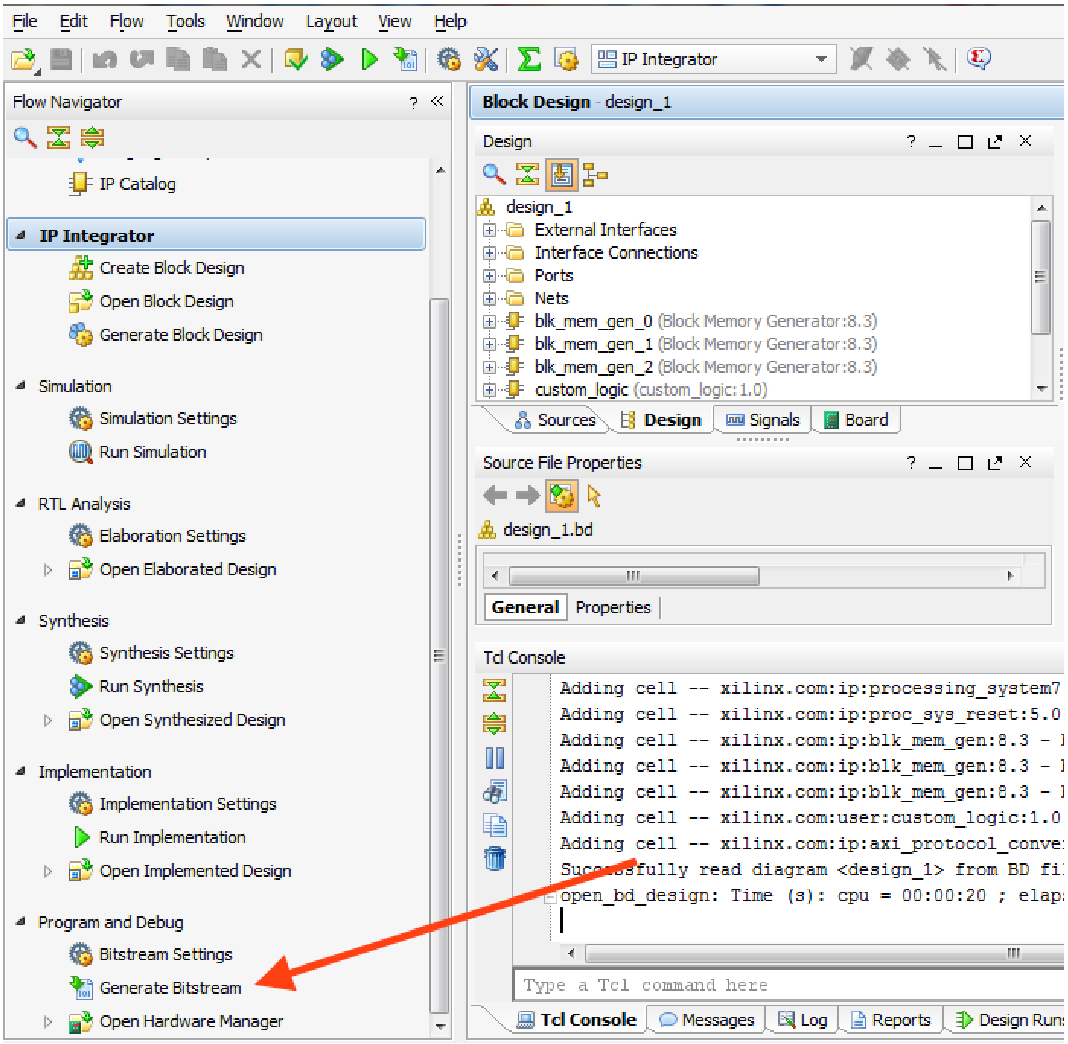


Figure 14: Generating a Bitstream in Vivado

Once the bitstream was generated, the hardware was exported to the SDK so that the PS had a platform to be coded on. This was done in Vivado by choosing File → Export → Export Hardware and including the bitstream. Finally, the SDK was launched by choosing File → Launch SDK.

When the SDK launches, there is a hardware platform project in the Project Explorer tab. This file contains the hardware platform that was exported from Vivado and will be used to program the FPGA on the ZedBoard. To begin programming the PS, an Application Project was created with the hardware platform. In the SDK, choose File → New → Application

Project. Enter a project name and choose Next. Note that the hardware platform exported from Vivado is selected in the Hardware Platform and is used to create the Application Project. Next, a template was chosen to begin designing. For this project, the 'Hello World' template was chosen. This process is shown in Figure 15. The new Application Project exists under the Project Explorer tab, and the PS can be edited in the project's source file folder.

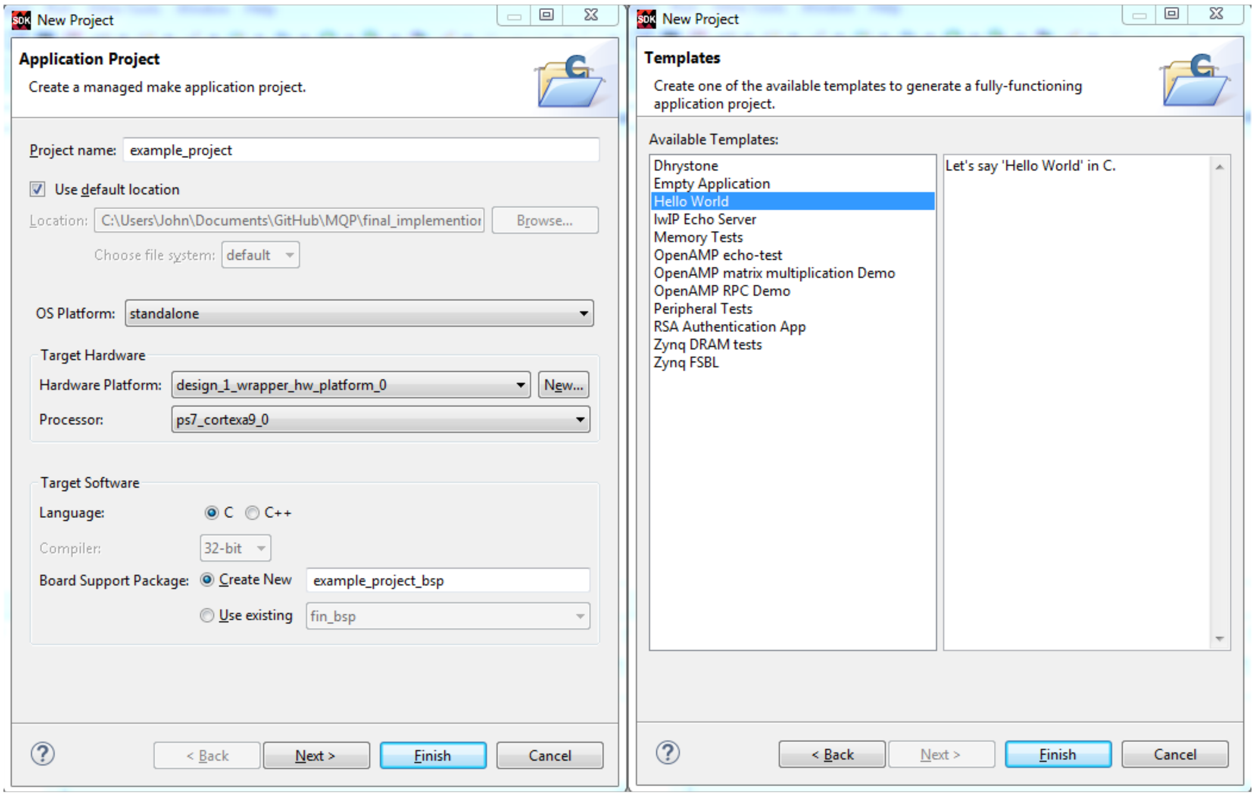


Figure 15: Creating a New Application Project in the Xilinx SDK

When we were ready to program the ZedBoard, the FPGA was programmed first in order to configure the PL by the hardware platform. This is done by choosing Xilinx Tools → Program FPGA. To indicate success, the ZedBoard's blue *DONE* LED, LD12, illuminated. Once that process was completed, the PS was uploaded by right clicking on the application project → Run As → Launch on Hardware (GDB).

4.3 Rangefinder Data Processing

With proper communication and the project set up in the SDK, the data processing began.

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 captures: 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. 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 written to the memory register connected to the PL, described further in Section 4.4.

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. To encode the data the 12 bits are separated into two 6-bit pieces, and then 30_{16} is added to each. The resultant data point is comprised of two ASCII characters [4]. The decoding process, taking place in the PL, is the inverse of encoding where 30_{16} is subtracted from each character and then they are merged together [14].

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 [29]. However, the sensor suite outputs data on a VGA screen which expresses data in the rectangular (Cartesian) coordinate system, so the data was converted from polar to rectangular coordinates. This was accomplished by using the step number which corresponds to an angle around a circle, as shown in Figure 11.

Converting from polar to rectangular coordinates requires basic trigonometry. Luckily, Xilinx supports a few options for performing trigonometry 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 implemented a LUT with a multiplier instead of a CORDIC function. The values in the LUT were used to extract the horizontal and vertical components from the polar coordinate. Taking advantage of a circle's symmetry, the LUT only holds 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.

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

This coordinate-axis transformation required both a horizontal and vertical scale factor, but only one address in the BRAM can be read from at a time. 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 E.iii.

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 4 clock cycles. This transformation converts the data from polar to rectangular coordinates. After this step, the data is right-shifted in a manner such that the data points are scaled according to a VGA screen with a 640x480 pixel resolution. Next the data was localized to the device's location, so the x- and y-coordinates were shifted by the device's location. After this step, the x- and y-location accurately reflect the distance data localized to the device. The rangefinder's data processing module is found in Appendix E.ii.

With proper x- and y-coordinates, the data is ready to be stored in memory. Another BRAM IP was created for this purpose. The VGA resolution is 640x480, so the BRAM requires 307,200 addresses[2]. 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. The write address was calculated by using Equation 3 with another Multiplier IP block.

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

With the data stored in memory, it was output to a VGA screen. To control the VGA logic, Digilent's VHDL VGA controller module, found in Appendix E.ii, was implemented. In addition, the ZedBoard's VGA pins were configured in a constraints file (.xdc) to support

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

[2]$640 \times 480 = 307,200$ addresses.

12-bit color resolution [6]. 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 PS-PL Communication

Communication between the Programmable Logic (PL) and Programmable Software (PS) was implemented so that data is able to traverse between the two. This process is configured in Vivado with the use of an Advanced eXtensible Interface (AXI) bus.

4.4.1 Advanced eXtensible Interface (AXI)

AXI is a type of on-chip interconnect specification intended for transaction based master-to-slave memory mapped operations, which makes it perfect for PS-PL communication. These AXI busses are integrally involved in most Xilinx IP cores and contain many signals and control flags.

The custom PL in this project was able to communicate properly with an AXI bus. Instead of attempting to interface with AXI by creating the signals, Vivado supports creating a new custom IP AXI Peripheral that abstracts away the complications of AXI communication.

4.4.2 Creating Custom IP

Vivado supports creating custom packaged IP blocks as AXI Peripherals. As an AXI Peripheral, this IP block is able to communicate with any other Xilinx IP blocks that use AXI. For the purpose of this project, the custom IP block communicates with the Zynq7 Processing System via its AXI bus.

The custom IP was created in Vivado under Tools → Create and Package IP. The IP was set up as an AXI Peripheral, with its data width set to 32 bits and the number of registers were set to 4, which are both the minimum amount.

After setting up a custom AXI peripheral, users will be presented with a bare IP module containing an AXI peripheral. With the peripheral creation complete, the custom IP's auto-generated files were edited to allow for the user's custom logic to be used in the AXI bus. In the auto-generated file instantiated in the top module, several lines of code were edited in order to connect the AXI peripheral's input and output channels to user logic. In the case of this project, the IP was customized for the use of four registers, but only two were used: one for writing data to the PS, and the other for reading from the PS.

To write to the PS, the output register data was wired to the data to be transmitted. This is done in the AXI read address *case* statement that decodes addresses for reading registers. This can be seen on line 368 of our edited auto-generated custom IP code, shown in Appendix E.ii.

To read from the PS, the data stored in the AXI's input register was wired to a buffer that is used by the PL. This is done in the AXI write address *case* statement on line 239 of our edited auto-generated custom IP code, shown in Appendix E.ii.

In addition, other necessary I/O ports from the custom logic were created in the custom IP's top module, in the same manner as a normal top module. Our custom IP top module can be found in Appendix E.ii. Advanced user logic was also implemented within the IP core through modular instantiation.

4.5 Inertial Measurement Unit (IMU) Operation

With the rangefinder's data processing implementation completed, the data is ready to be offset based on the device's relative location and direction in space. This was done with the use of an Inertial Measurement Unit (IMU). An IMU is a device that measures linear and angular momentum, as well as the direction of the magnetic field at a point in space. It accomplishes this by reading data from an accelerometer, gyroscope, and magnetometer respectively.

4.5.1 Selection

This project required a sensitive IMU that is able to provide data to tell compass direction and change of position. Due to the time limitations and budgetary restrictions of this project, we chose the PmodNAV IMU that provides 10-degree of freedom functionality through the LSM9DS1 3-axis accelerometer, 3-axis gyroscope, and 3-axis magnetometer, and the LPS25HB barometer [20, 19]. The PmodNAV is shown in Figure 16.

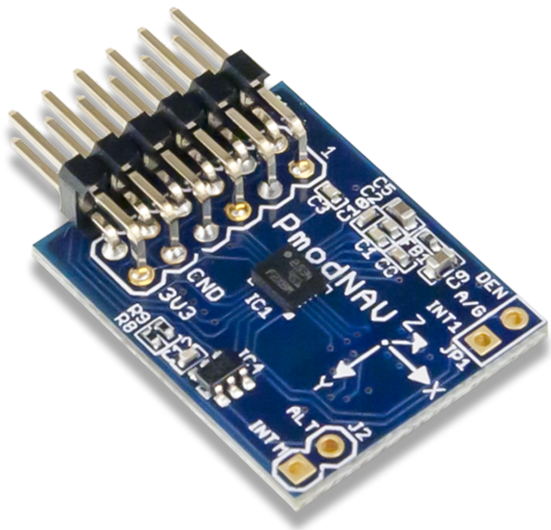


Figure 16: The PmodNAV 10-Axis IMU [9]

The PmodNAV's simple Pmod connector made the IMU easy to integrate into this design, as it was fixed in position connected to the ZedBoard. Also, its communication was directly compatible with the ZedBoard, requiring no intermediate hardware.

4.5.2 Communication

The IMU supports two means of communication: Serial Peripheral Interface (SPI) and Inter-Integrated Circuit (I^2C) Communication [20]. However the magnetometer on the LSM9DS1 is not addressable by the I^2C bus. Since the magnetometer was needed for its ability to tell compass direction, SPI communication was implemented. The magnetometer sensor SPI protocol is shown in Figure 17.

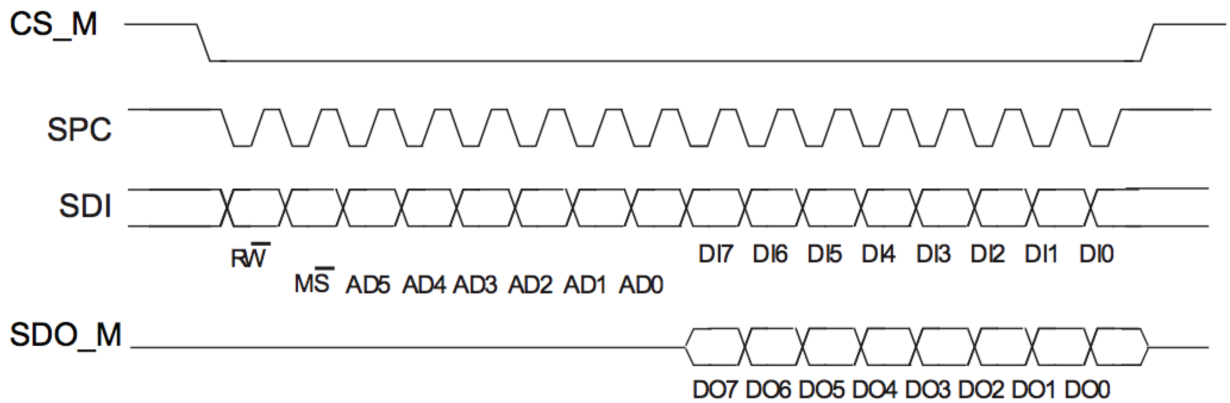


Figure 17: Magnetometer SPI Read and Write Protocol [20]

The CS_M line is the magnetometer chip select and is an active low. It goes low at the beginning of the transaction and high at the end. The SPC is the clock controlled by the master. SDI and SDO_M are the data input and data output lines, respectively. They are driven at the falling edge of SPC and should be captured at the rising edge [20].

The register read and write commands are completed in 16 clock pulses. The first bit sent from the master, bit 0 or $R\overline{W}$, is the read/write bit. When data is read from the IMU this bit is set to 1, otherwise it is set to 0. When bit 1, the $M\overline{S}$ bit, is set to 1 the address is auto-incremented, allowing for multiple reads or writes to be completed in the same SPI transaction. Figure 18 shows a multiple-byte SPI read protocol. Bits 2-7, the AD bits, are the address bits transmitted MSB first. When in write mode bits 8-15, the DI bits, are the data that is written to the device MSB first. When in read mode bits 8-15, the DO bits, are the data that is read from the device MSB first [20].

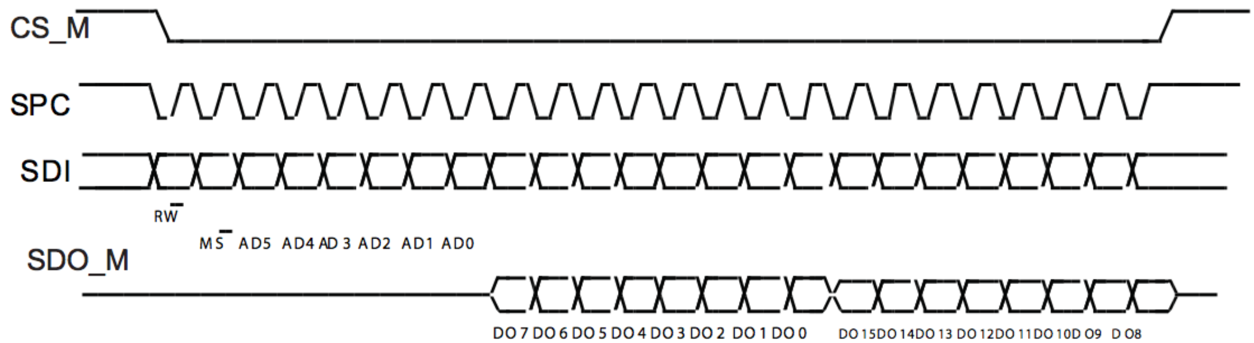


Figure 18: Magnetometer Multiple-Byte SPI Read Protocol [20]

4.5.3 Register Settings

The LSM9DS1 IMU required memory register setting in order to turn on the magnetometer and function properly. The memory register settings were set by writing to the IMU in the SPI communication format shown in Figure 17. For all of the settings adjustments the $R\overline{W}$ bit is set to 0, signifying a write command.

One register that was written to was the magnetometer's control register 1, CTRL_REG_1_M, which is at address 20_{16}. $7C_{16}$ was written to CTRL_REG_1_M to signify ultra high performance mode for the magnetometer's x- and y-axis [20].

The other register that was written to was the magnetometer's control register 3, CTRL_REG_3_M, at address 22_{16}. 80_{16} was written to CTRL_REG_3_M to turn off I^2C and turn on the magnetometer in the continuous-conversion mode [20].

Due to time constraints, the magnetometer on the PmodNAV was the only slave that needed to be interfaced to. The accelerometer, gyroscope, and barometer were not used so the above registers only needed to set once.

4.5.4 Read Registers

With the magnetometer turned on and its registers set, its data was ready to be read. As such, the $R\overline{W}$ bit was set to 1. There are two registers that were read from in order to ensure data accuracy.

One register that was read from was the status register, STATUS_REG_M, which is at address 27_{16}. This address was read from until the two least significant data bits read 11_2, which signified that new x- and y-axis magnetometer data was ready. Once new x- and y-axis data was ready, the corresponding data registers were read from [20].

The x- and y-axis data came in 16-bit resolution. Due to the SPI transfer protocol shown in Figure 17, data was read 8 bits at a time MSB first. Since each axis had 16-bit resolution, each axis had two addresses containing 8-bit data words. The x- and y-data addresses are consecutive, so the 32 bits of data was obtained in one cascading read in the format shown in Figure 18. Because of this, we performed a cascading read from address 28_{16}, OUT_X_L_M, to obtain the x-axis lower word, the x-axis upper word, the y-axis lower word, and then the y-axis upper word [20].

4.6 IMU Implementation

As the IMU was connected to one of the ZedBoard's Pmod connectors, it had the option of being controlled by either the Programmable Logic (PL) or the Programmable Software (PS). The IMU's magnetometer data was be used to rotate the rangefinder data according to a compass direction by offsetting the rangefinder's step value. Since the step value was set in the PS, all of the IMU's implementation has the ability to be done in the PS, which is a simpler and quicker solution than taking advantage of the PS-PL communication setup from Section 4.4. Another motivation for using the PS is that the IMU data processing involves complex trigonometry, which is simpler done in the PS than by using a CORDIC function in the PL.

As such, the Zynq7 Processing System was re-customized to provide this capability.

4.6.1 Re-Customizing the Zynq7 Processing System

The Zynq7 Processing System is easily re-customizable, as discussed in Section 4.2.2. SPI functionality was added in the processing system's customization window under I/O

Peripherals in the MIO Configuration tab. The SPI pins were routed to Extended MIO (EMIO) so that the ZedBoard's PL Pmod was controlled from the PS. Since both SPI0 and SPI1 have EMIO functionality, SPI0 was chosen over SPI1. This process is shown in Figure 19.

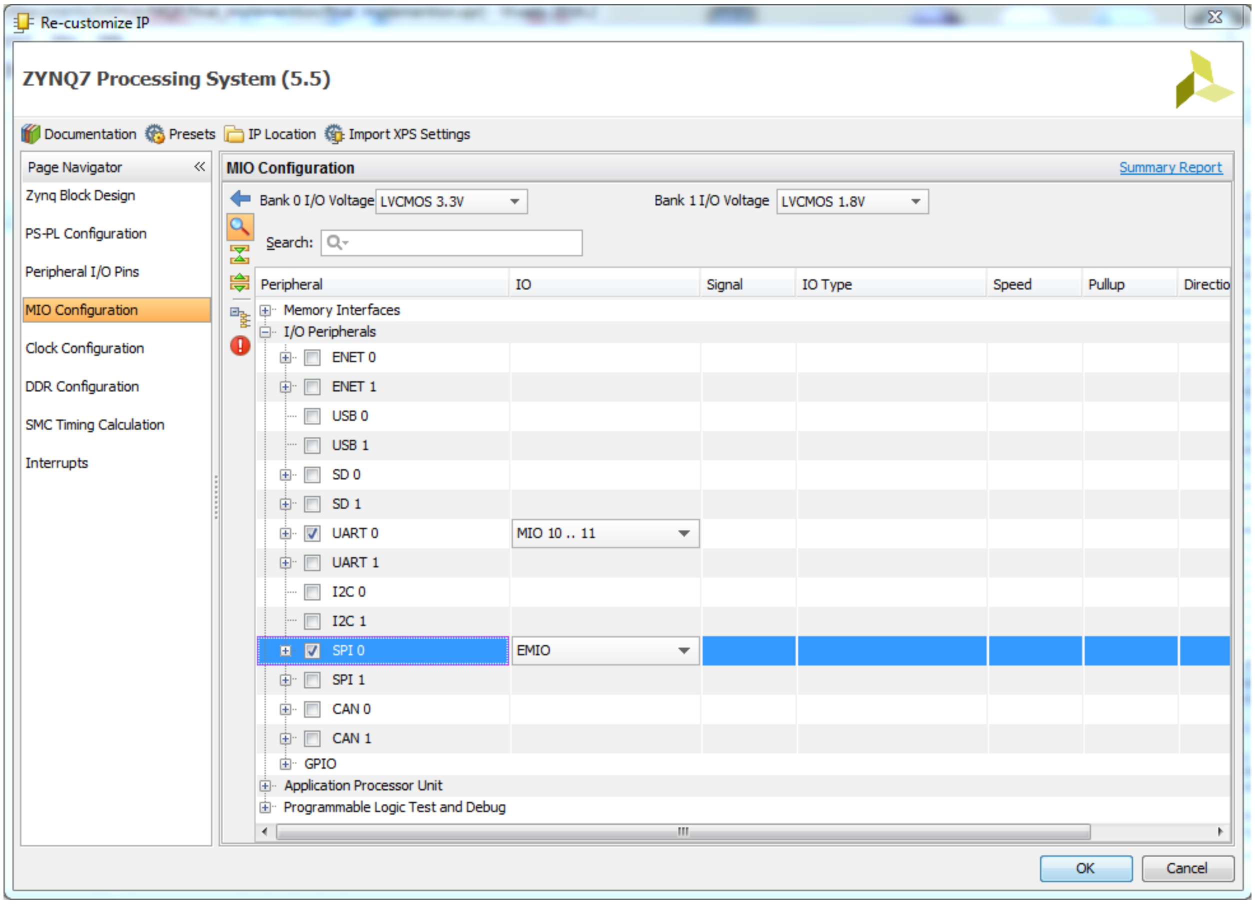


Figure 19: Re-Customizing the Zynq7 Processing System to Add SPI

EMIO functionality allows for pins normally connected to the Zynq7 processor's programmable logic to be connected to the processor's programmable software instead. In other words, EMIO peripherals allow for the user to control physical pins using C code running on the Zynq7 dual ARM processors rather than in Verilog-defined hardware. By creating an EMIO SPI peripheral, it was possible to route the physical connection of the interface to any pin on the ZedBoard that is accessible by programmable logic. In the case of this specific implementation, the EMIO SPI peripheral was routed to one of the ZedBoard's Pmod ports, allowing for the MIO Pmod port to remain unused and open for I^2C and UART peripheral communications.

4.7 IMU Data Processing

The IMU's data processing was implemented in the Programmable Software because it involves complex mathematics, and was easily integrated with the rangefinder's data in the Xilinx Software Development Kit (SDK).

4.7.1 Programmable Software

The SPI communication in the SDK was customized and implemented by following example code provided with the SPIPS driver under the Xilinx SDK. The examples are located in the following folder: `C:\Xilinx\SDK\2016.2\data\embeddedsw\XilinxProcessorIPLib\drivers\spips_v3_0\examples\`.

By following Xilinx's examples, the IMU's register settings were set and then the axis data was read from their respective registers. The axis data was signed and expressed in Two's Complement format[3] [20]. The axis data was read into an array of unsigned 8-bit numbers. The data points were rearranged and then stored into an integer buffer for each axis. Combining the data in this manner works if the *int* data type were only 16 bits. However, the data type *int* in the Xilinx SDK is 32 bits. For positive numbers this method was sufficient, but for negative numbers this process dropped the sign bit. The sign bit was the most significant bit of the axes' 16-bit data, which got lost when getting stored in a 32-bit integer. This problem was corrected by checking if the data stored in each axes' most significant word was greater than or equal to 80_{16}. If greater than or equal to 80_{16} then the sign bit must be '1', signifying a negative number. If necessary, the data was turned negative by subtracting $FFFF_{16}$, or $65{,}535_{10}$, and adding 1.

[3] Two's Complement is a way of encoding signed numbers in binary where the most significant bit is used as a sign bit, with '1' signifying a negative number and '0' signifying a positive number. To convert a positive number to negative, all of the bits are inverted and then 1 is added to the resultant number [11].

Before the complex math was performed on the data, the header file *math.h* was included into the project. This was done by right clicking on the application project, choosing Properties → C/C++ Build → Settings → Tool Settings → ARM v7 gcc linker → Libraries, and then adding *m* under the Libraries (-l) section, shown in Figure 20. In addition, *math.h* still was included by using *#include* in the project's source code file. This process allowed the included *math.h* header file to be linked into the project by the linker.

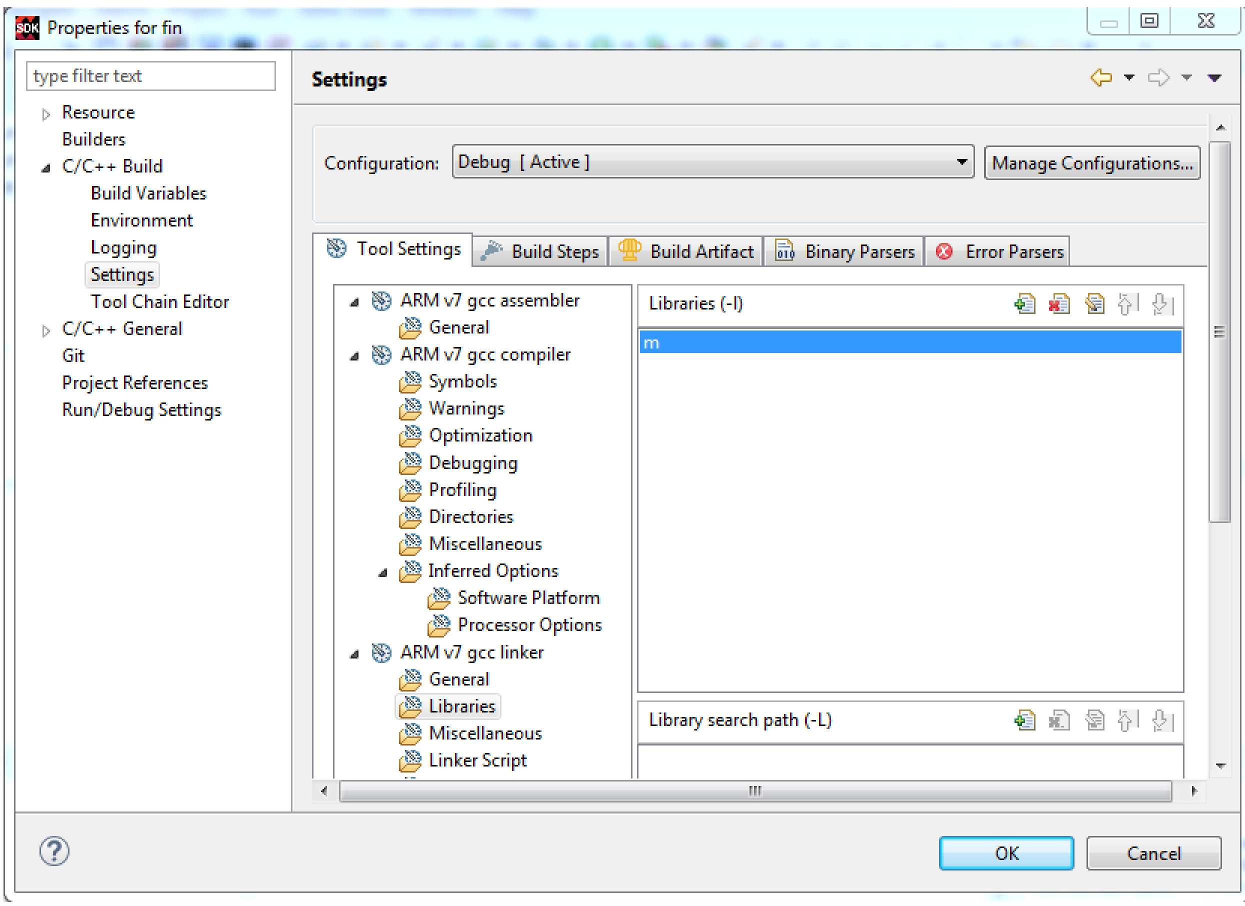


Figure 20: Linking *math.h* into the Application Project in the SDK

Once the magnetometer axis data was accurately stored in their corresponding buffers and *math.h* properly linked, the data began its transformation. The data was expressed in terms of milligauss per bit, which was converted to a compass heading in degrees by using Equation 5.

$$\text{Compass Heading} = \arctan(\frac{y}{x}) \times \frac{180}{\pi} \tag{5}$$

To convert the compass heading to a rangefinder step offset, Equation 6 was used.

$$\text{Step Offset} = \frac{\text{Compass Heading}}{\frac{360°}{1024\ steps}} \tag{6}$$

The resultant step offset was added to the rangefinder's step in order to account for the sensor suite's compass direction deviation from North. When the ZedBoard faces North the step offset equates to 0, so the rangefinder's data was not rotated. When the ZedBoard faces South the step offset equates to 512, which rotates the rangefinder data by 180°.

4.8 Camera Operation

Several important criteria were analyzed in order to find a proper camera module for this project. After selecting a camera for use, further research was then necessary to determine how to properly operate the chosen modules.

4.8.1 Camera Selection

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 made it so that our setup was 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 were the only low-cost global shutter option we were able to find in our research, and were 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 [16, 25]. Once the schematics of each camera module breakout were known, it was then possible to design a basic control interface for each camera.

4.8.2 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 [25]. 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 contains 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 21 shows the operation of these pins while transmitting an image.

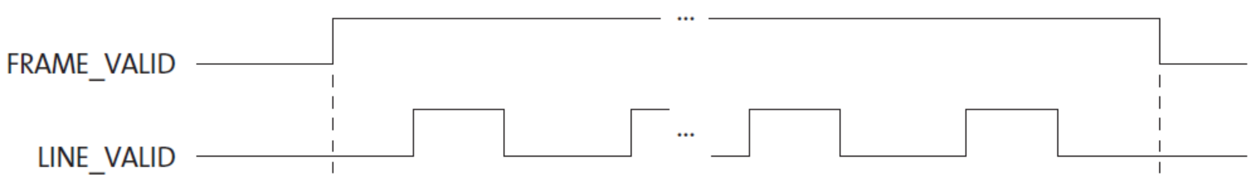


Figure 21: Frame and Line Valid [25]

Since the MT9V034 module transmits image data in parallel and each pixel contains 10 bits of resolution, 10 pins were 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 22.

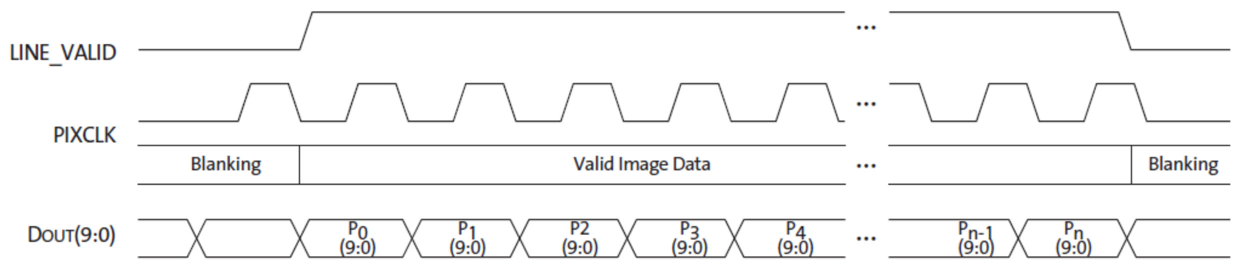


Figure 22: Line Data Transfer [25]

The default camera data transmission scheme was also examined using an oscilloscope, as shown in Figure 23, with channels 1-4 corresponding to camera PCLK, FRAME_VALID, LINE_VALID, and Data[0], respectively. In the case of Figure 23, the camera is initially powered off, resulting in an inactive PCLK signal during the beginning of the recording.

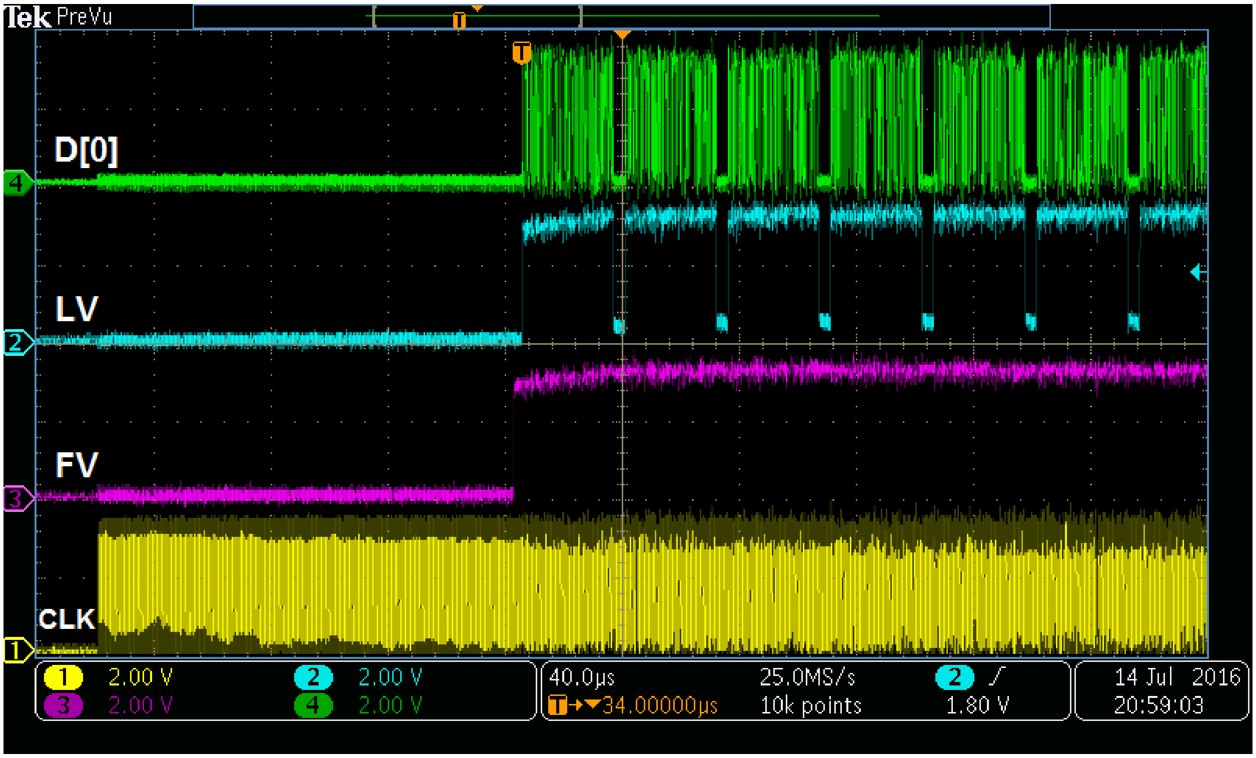


Figure 23: Camera Data Transfer

4.8.3 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 I^{2}C interface will use a Serial Data Line (SDA) and Serial Clock Line (SCL) that are normally pulled to 5V. When one connected I^{2}C 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 24. 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.

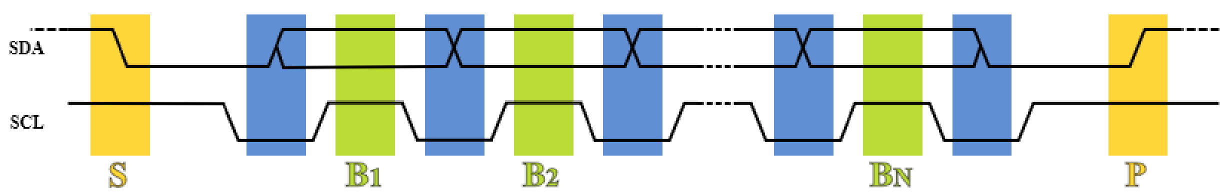


Figure 24: Example I^{2}C Data Transfer

Based on the LIVM34LP camera board schematic, each breakout board has been configured so that its camera is accessible at I^{2}C address 0x58 [16, 25]. Note that since both cameras come configured with the same I^{2}C bus address, a pullup resistor needed to be added to one of the cameras I^{2}C address lines so that both were individually accessible on a shared bus.

4.8.4 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 7.

$$\text{Image Size} = 752px * 480px * 10\frac{bits}{pixel} = 3609600\, bits * \frac{1\, byte}{8\, bits} * \frac{1kB}{1024\, bytes} = 440.625kB \quad (7)$$

In order to send a camera image to a computer or monitor for viewing, several steps needed to be taken. Although it would have been 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 was to buffer the image between the camera and the desired output source, since this allowed 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 was 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[4]. This left 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

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

specifically for buffering VGA imagery similar to that of the MT9V034 camera module, and can be connected directly to the camera module outputs [5]. 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 [5]. 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 needed to be truncated. This isn't a major issue, since the truncation corresponded to a 4/1024 reduction in the range of values that each pixel can map to.

4.9 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 capture imagery of the same scene from slightly different vantage points, depth information on the scene may be extracted by calculating 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 [7].

4.9.1 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 25 below. Although an ideal stereo camera setup would contain shared epipolar lines between camera images, raw image data from each camera contains 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.

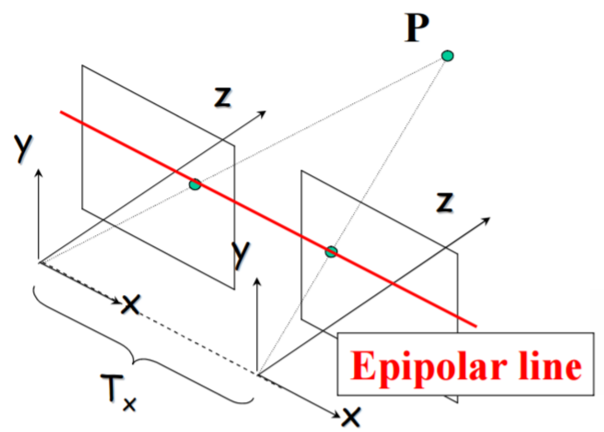


Figure 25: Horizontal Epipolar Lines [7]

A pictorial representation of the process of stereo image rectification is shown in Figure 26 below [22]. This process is achieved using a 3x3 matrix coordinate transform based on parameters obtained from the external calibration process.

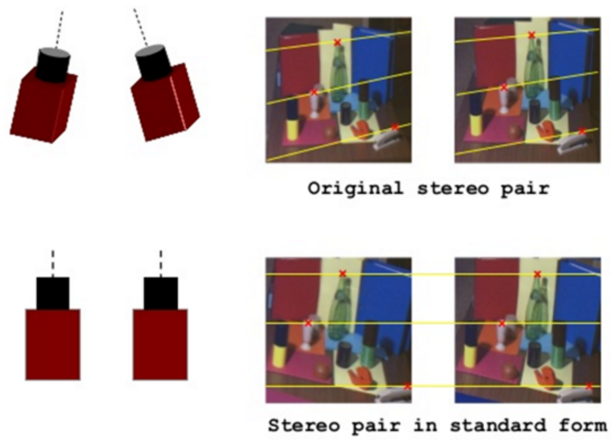


Figure 26: Stereo Image Rectification [22]

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.

4.9.2 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 was 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 was performed using 7x7 pixel search blocks over 20 pixel horizontal ranges, and was repeated throughout the image. The expression for the sum of absolute differences is shown in Equation 8 below.

$$SAD = \sum_{x} \sum_{y} |template - block| \tag{8}$$

A visual representation of the Sum of Absolute differences is shown in Figure 27, 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.

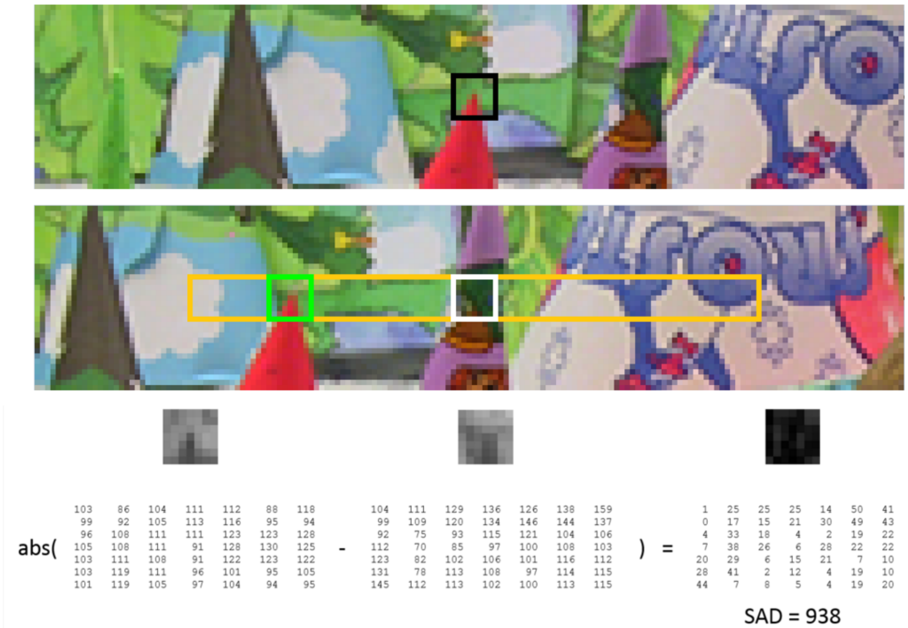


Figure 27: Sum of Absolute Differences [23]

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 28 below. In the case of Figure 28, higher match score values for each search block indicate lower SAD values.

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 unit of distance using the focal point f and baseline distance T_x between image sensors as shown in Equation 9 below.

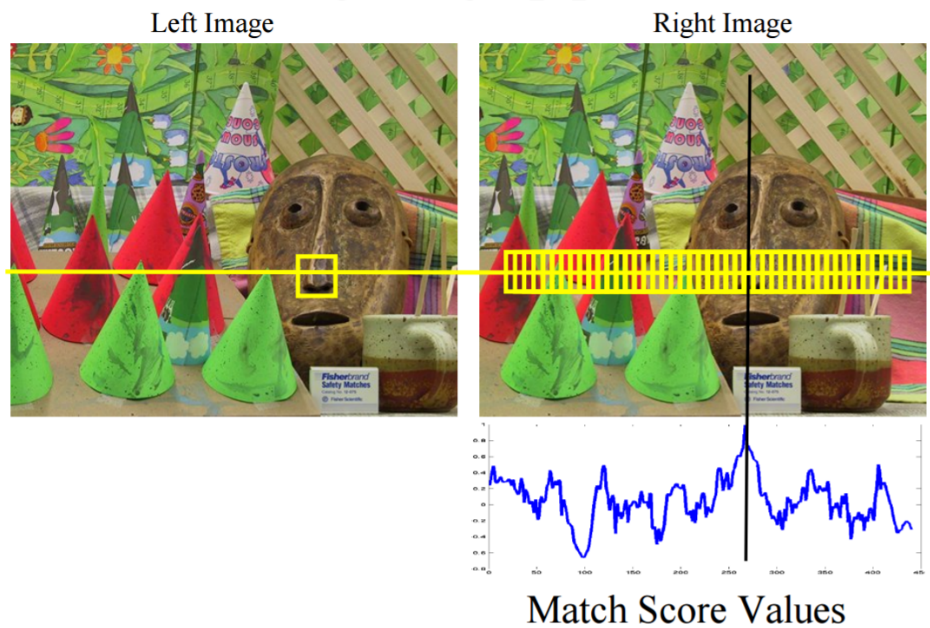


Figure 28: Block Matching Overview [7]

$$depth = Z = \frac{fT_x}{d} \tag{9}$$

Pixel coloration values in a disparity image are based on the distance calculation shown in Equation 9, 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 29 below.

Figure 29: Disparity Algorithm Output

4.10 Combined Implementation

The following sections describe the integration of each individual sensor's data into the final, combined implementation. The rangefinder and disparity data was combined to produce accurate depth estimations. This data was also integrated with navigational data to create a more accurate approximation of the sensor suite's relative surroundings.

4.10.1 Rangefinder and Disparity Data Integration

3D depth information from the disparity algorithm was combined with 2D depth information from the scanning laser rangefinder in order to increase the overall accuracy of the system. Since the rangefinder and stereo camera interface shared a horizontal viewing plane and both sensors gathered information on the same scene, there was some distinguishable overlap in sensor data. This overlap was taken advantage of in order to produce a more accurate 2D "floorplan" of the area being observed. This type of data integration is especially useful in situations where the scanning laser rangefinder is out of range.

Through the use of a moving average Finite Impulse Response (FIR) filter across a horizontal line of depth information from the disparity algorithm output, a single line of depth information obtained was correlated with rangefinder data obtained from the same scene. Note that although 2D rangefinder data is organized using a polar coordinate scheme, the output buffer used for displaying rangefinder data via VGA contains the same data in a Cartesian format. This Cartesian rangefinder data was easily combined with averaged disparity depth information at the output stage, where both sensors' data is displayed relative to the same central location on screen.

In order to correlate both sensors' data for a combined output mode, the field of view of each device was taken into account. Since each camera has an approximate 55° field of view, and camera imagery is 752 pixels wide, the stereo camera interface has a deg:pixel ratio of $\frac{752}{55} = 13.67\frac{px}{deg}$. Output data from the rangefinder is divided into 768 steps over a 270° field of view. In order to correlate disparity data with rangefinder data, the averaged disparity depth line was converted an equivalent number of "steps" worth of data. The conversion factor for pixels of disparity depth to "steps" was calculated as shown by Equation 10.

$$13.67\frac{px}{deg} * \frac{270°}{768\ steps} = 4.8\frac{px}{step} \tag{10}$$

The output from the disparity pixel line was scaled down by an approximate factor of 4.8 in order for it to correlate with depth information from the scanning laser rangefinder. Once this scaling process was complete, depth information from the disparity algorithm was directly overlaid on the 2D scanning laser rangefinder's output in order to produce a combined depth map.

4.10.2 Accounting for Navigational Data

With the IMU's accelerometer, gyroscope, and magnetometer, displacement and orientation was calculated. The orientation and displacement was displayed on a screen to show the device's behavior and its location. This was combined with the rangefinder's data to show the orientation of the device, where it has traversed, and the distance away of the objects closest to it.

The IMU's compass data was combined with the rangefinder's step count. Changing the step count rotates the distance data around the device by changing the starting point of the data processing. This changes the direction of the rangefinder's 240° field of vision. For example, an IMU reading corresponding to due West is a rotation of 90° counterclockwise from due North. By Equation 11, 90° equates to 256 rangefinder steps. So, the rangefinder's data essentially begins at step 256 and end at step 1024 or 0[5], seen in Figure 11.

$$90° \div \frac{360°}{1024\ steps} = 256\ steps \tag{11}$$

Due to time constraints, the IMU displacement data was not able to be incorporated into

[5] step 768 + 256 step offset = 1024. Since there are 1024 rangefinder steps in a circle, step 1024 is the same as step 0.

this project. The displacement data was planned to change the device's location on the VGA screen. Ideally, the device would appear to traverse the screen as the device itself traverses a location, with the rangefinder's data traveling with it. Since the rangefinder's data is stored in such a way that it is localized to the device's location, incorporating the IMU's displacement data is as easy as obtaining the data from the IMU in the PS and sending it to the PL via an extra PL read register discussed in Section 4.4.2.

In this chapter, each sensor's behavior was explored, and their implementation into the system was designed such that their data could be combined together. With this completed, each sensor's behavior was tested to ensure that their actual behavior matches their documented behavior.

5 Testing and Results

After learning how to interface with the chosen sensors, it was then possible to begin creating test implementations for each individual sensor. Many of the initial tests performed on each sensor were designed so that they could be later integrated into a finalized design, as detailed in the following sections.

5.1 Rangefinder Testing

The URG-04LX scanning laser rangefinder requires an external 5V power source connection. With the power connected and the device on, the device is ready and waiting for communication.

5.1.1 Testing via the Data Viewing Tool

The URG-04LX has a data viewing tool which is a useful application by Hokuyo Automatic Co. that can be used to view, record, and replay the device's data. To use this tool the device was plugged into a computer via its USB port. Figure 30 below shows a screen capture of the application recording data captured by the rangefinder.

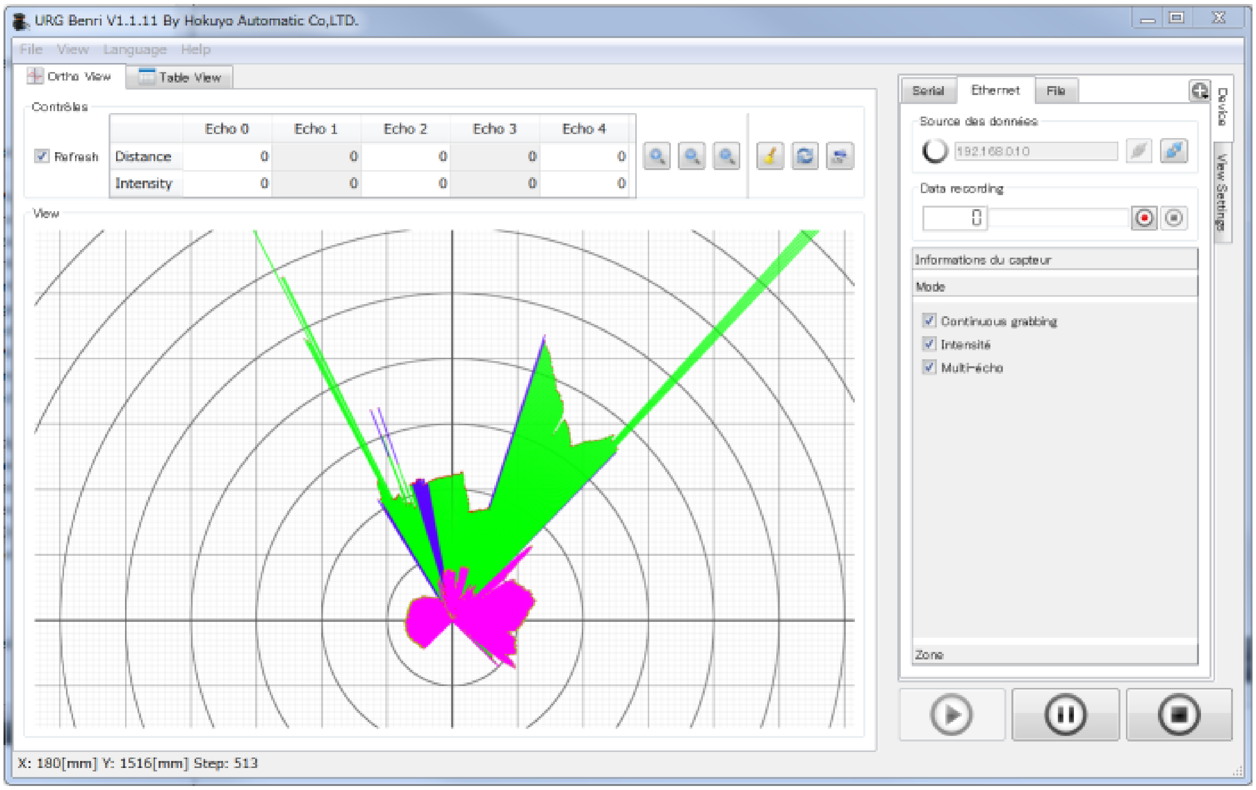


Figure 30: Screen Capture of the URG-04LX Data Viewing Tool [15]

Note that the start point of 0, end point of 768, and dead zone align to that shown in Figure 11. For this project the data viewing tool was used to verify our project's data processing functionality.

5.1.2 Command Testing

In addition to the data viewing tool, the rangefinder's commands were tested by connecting it to a laptop via its USB port. We used PuTTy, a serial console application, to communicate with the rangefinder. Figure 31 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.

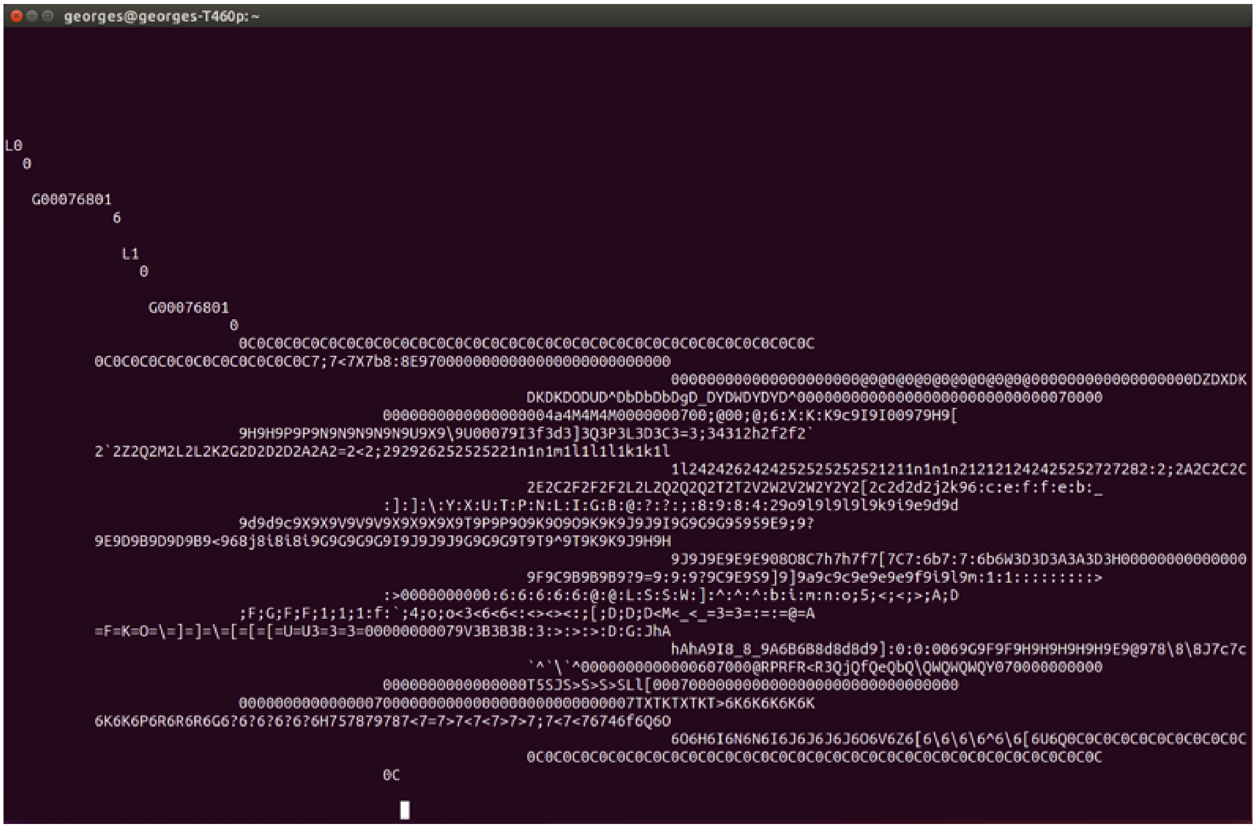


Figure 31: Rangefinder Communication Test via PuTTy

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 [14]. By communicating with the rangefinder via PuTTy, we were able to observe the rangefinder's behavior and confirm the data acquisition command functions properly. This testing also verified that communication via the rangefinder's USB port was working.

5.1.3 Communication via USB On-The-Go (OTG)

With communication via the rangefinder's USB port working, we decided to continue with this mode of communication. The ZedBoard supports USB On-The-Go (OTG) which is a specification that allows USB devices to act as a host for other USB devices [28]. With USB OTG, a device chooses to act as a peripheral or a host if necessary. For the purpose of this project, the ZedBoard should act as the host by initiating communication with the rangefinder. Enabling USB OTG can be done in the Zynq7 Processing System and controlled through the PS. The rangefinder's laser illumination command was chosen to be transmitted from the ZedBoard to test the communication. This command was chosen because when received, the status LED on the rangefinder blinks until the laser is turned back on, which is a simple way of verifying successful communication. In addition, when a command is transmitted via UART from the ZedBoard, its TX LED flashes. A fully successful transaction observes the ZedBoard's TX LED flashing and then the status LED on the rangefinder blinking.

The ZedBoard was programmed, the rangefinder was turned on, and the two devices were connected by a standard micro-USB to mini-USB cable. The ZedBoard transmitted the command, as signified by the blink of the TX LED. However the rangefinder did not acknowledge the command; its status LED stayed lit signifying the laser was on. Due to this failure[6], using USB OTG was not implemented. Instead the methodology described in Section 4.1.2 was implemented.

[6] This communication failure was most likely due to the lack of necessary hardware, as USB OTG requires an adapter that controls which device will be hosting the communication. Without this adapter, both USB devices act as a peripheral, and neither will initiate communication [28].

5.1.4 Communication via Pmod

Once we decided not to continue with USB OTG, we routed the UART signals to a Pmod connector, described in Section 4.1.2. To make sure that UART via Pmod was functioning correctly, the transmit pin was measured with an oscilloscope. The laser illumination command "L0\n" was transmitted and observed indicating success, as shown in Figure 32. Note that this is a TTL signal.

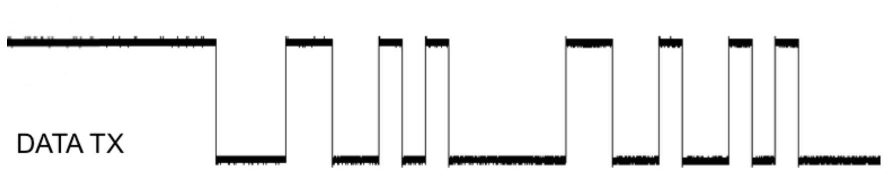


Figure 32: Laser Illumination Command TTL Oscillogram

The RS-232 to TTL converter with the attached breakout board was attached to the ZedBoard. The converter's V_{CC} and GND were connected to the ZedBoard Pmod's respective V_{CC} and ground pins. When these pins were connected, the converter's power LED turned on. In addition, the converter's RX and TX pins were connected to the ZedBoard's respective TX and RX pins. The breakout board's TX pin was measured on the oscilloscope to observe the resultant RS-232 waveform. However when the command was transmitted from the ZedBoard, there was no change on the oscilloscope. We disconnected the converter's TX and RX pins and reconnected them such that the converter's RX and TX pins were connected to the ZedBoard's respective RX and TX pins. The laser illumination command was re-transmitted and the waveform in Figure 33 was observed on the oscilloscope. The oscillogram shows a waveform from +6V to -6V, which is a valid RS-232 signal.

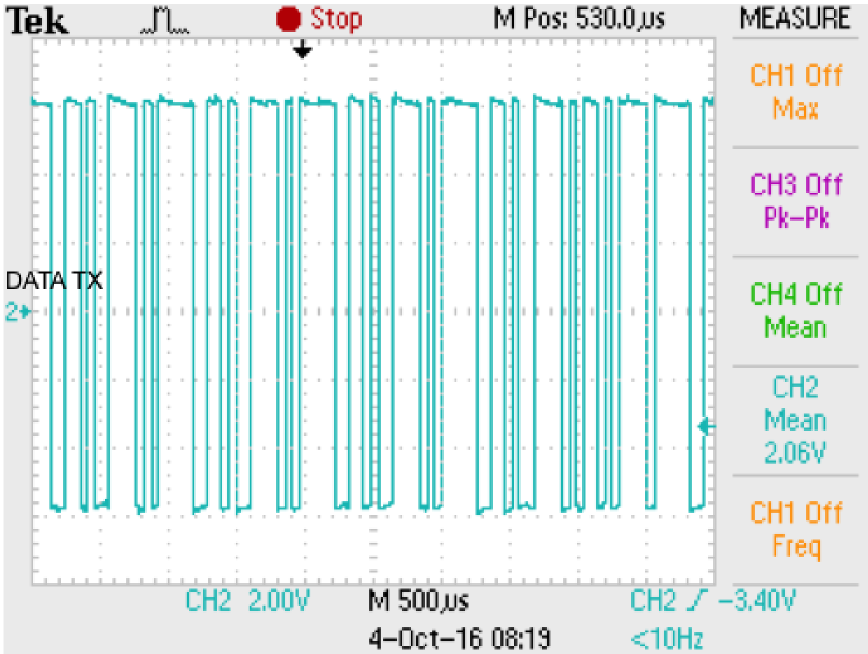


Figure 33: Laser Illumination Command RS-232 Oscillogram

With the communication functioning properly, the rangefinder's RX and TX were connected to the breakout board's respective TX and RX pins, and the laser illumination command was transmitted from the ZedBoard. The rangefinder's status LED started blinking, signifying that it received the laser illumination command and the laser was turned off. This test's success indicated that the rangefinder's communication was completely successful.

5.1.5 PS-PL Testing

Once UART communication was verified, the next step was to test the PS-PL communication. In order to test the communication, UART was reconfigured in the processing system to be routed to USB UART.

PL to PS communication was tested first, by using a PL button press to initiate a UART transfer. With UART routed to USB UART, the TX LED will flash when data is transmitted. In the PL, BTNR was used as an input and was wired into the AXI's output register, *reg_data_out*, as *slv_reg*0, as seen on line 368 of the custom IP's instantiated file located in Appendix E.ii. In the PS, the data was read from the AXI bus by pointing to the address in memory where the PL's output register, slv_reg0, is located. This address was found in the SDK in the $system.dhf$ file, which contains the hardware platform specifications. The base address is the cell with the same name as the custom IP. For this project, the base address was $43C00000_{16}$. Since slv_reg0, the first of the four designated memory registers, was used there does not need to be any address offset. Reading from the PL was implemented on line 30 of the PS, shown in Appendix E.iv, by using Xilinx's function Xil_In32 to read the data from the memory address that is $baseaddr\_p$[7]. Once the setup was complete, the ZedBoard was programmed and connected to a serial console. BTNR was pressed and the TX LED lit up, indicating that PL to PS communication was functioning properly.

PS to PL communication was tested next per use of the VGA screen. For this test, the PS receives the transmit signal from the PL and then waits for 768 data points to be received, just as if the rangefinder were connected. Since UART was routed to USB UART, the ZedBoard communicated with a serial console. Through the serial console, rangefinder communication was simulated by inputting a block of rangefinder data. The data was written to the PL one data point at a time by writing to slv_reg1, as on line 239 of the custom IP's instantiated file located in Appendix E.ii. This register is located one memory register from the base address of the custom IP because it is the second of the four designated memory registers. The function Xil_Out32 was first tested but no results were observed, so a pointer was used to write to the base address offset by one memory register. This is seen on line 198 of the PS, in Appendix E.iv. The data written to slv_reg1 was the distance data point, a data valid flag, and the rangefinder step. These were manipulated to fit into one 32-bit integer by shifting each to a unique bit location of a buffer, $data\_enable\_step$.

To test data accuracy in addition to PS to PL communication, the block of data sent from the serial console was constant. With data constant across all steps of the rangefinder's field of view, 270° of a circle are intended to be drawn around the rangefinder on the VGA screen. With the VGA module set up and the rangefinder data processing ready to be tested,

[7]This could also have been accomplished by using a pointer to read the data from the memory address that is $baseaddr\_p$.

the ZedBoard was programmed. When BTNR was pushed, an image similar to Figure 34 was observed on the VGA screen with the red dot being the device and the black lines being the rangefinder's distance data.

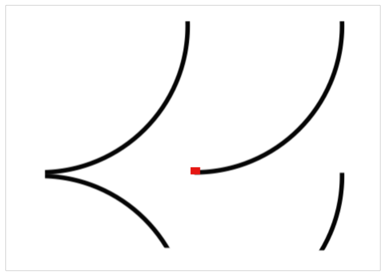


Figure 34: PS to PL Communication Test with Constant Data

Although a circle was not observed, this test confirmed the PS to PL communication was functioning properly. The shape appears to look like four quadrants of a circle in the wrong direction. Since the lines seem semi-circular, the polar-to-rectangular transformation were successful, too. The problem was a minor sign issue with the rangefinder's data processing in the PL. The signs in each necessary quadrant were fixed and the test was repeated with Figure 35 observed on the VGA screen.

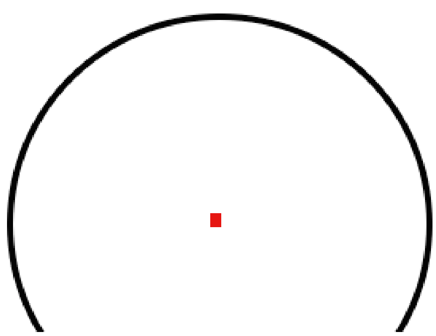


Figure 35: PS to PL Communication with Constant Data and Edited Data Processing

With the PS-PL communication and rangefinder data processing functioning perfectly, the rangefinder itself was attached and tested.

5.1.6 Data Testing

UART was re-routed to the PS Pmod in order to test the entire rangefinder implementation. The rangefinder was powered by the lab bench power supply and was connected on the RS-232 breakout side of the RS-232 to TTL converter, with the ZedBoard connected to the TTL side. The ZedBoard was connected to the VGA screen, and then was programmed. BTNR was pushed to initiate the UART transfer and Figure 36 shows the VGA output.

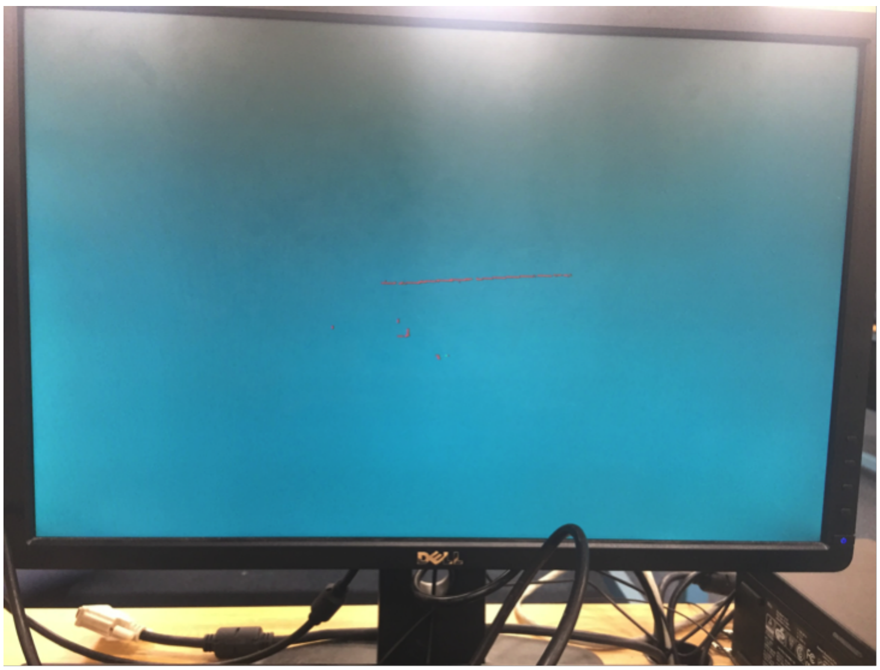


Figure 36: Rangefinder Data Observed on VGA Screen

Next BTNR was pushed again to start another data transfer, but there was no observed functionality. The button was held down until the subsequent data transfers in Figure 37 were observed on the VGA screen.

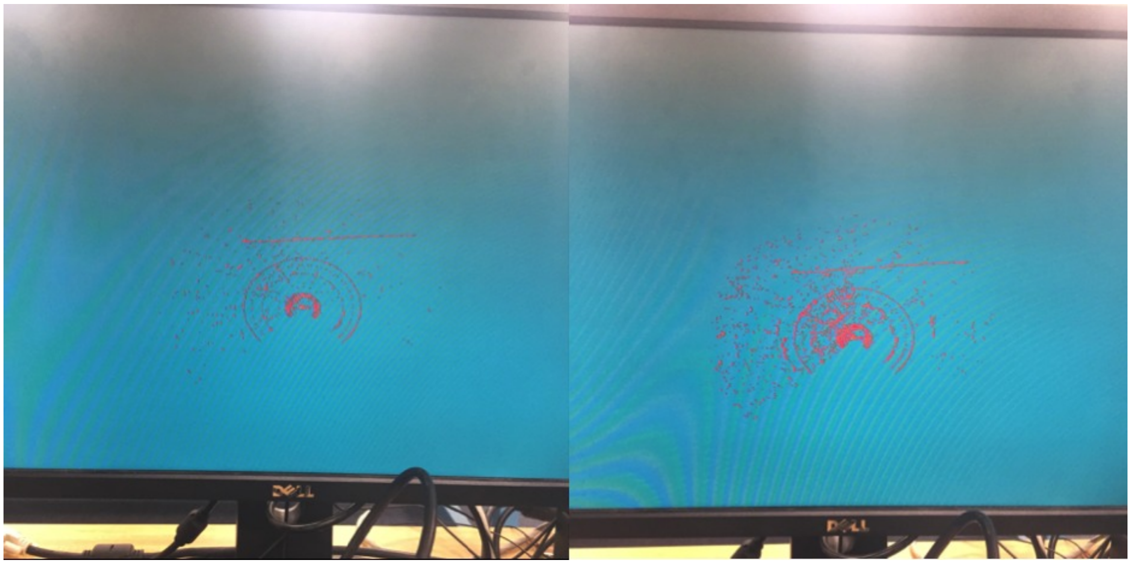


Figure 37: Subsequent Rangefinder Data Observed on VGA Screen

In the SDK the PS was not accounting for enough data points. When the rangefinder receives a command from the ZedBoard it echoes back the command. This is used as a test to ensure data accuracy. Since not enough data was being accounted for, the extra data was writing into the next data transfer's input echo buffer. As a result, the echo received from the rangefinder did not match the command transmitted and the rest of the data was garbage. The PS was edited to account for all of the data points and the data transfers were able to be triggered every time BTNR was pressed.

Figure 38 shows a test of the rangefinder where its output was compared to the objects around it.

(a) Lab at WPI

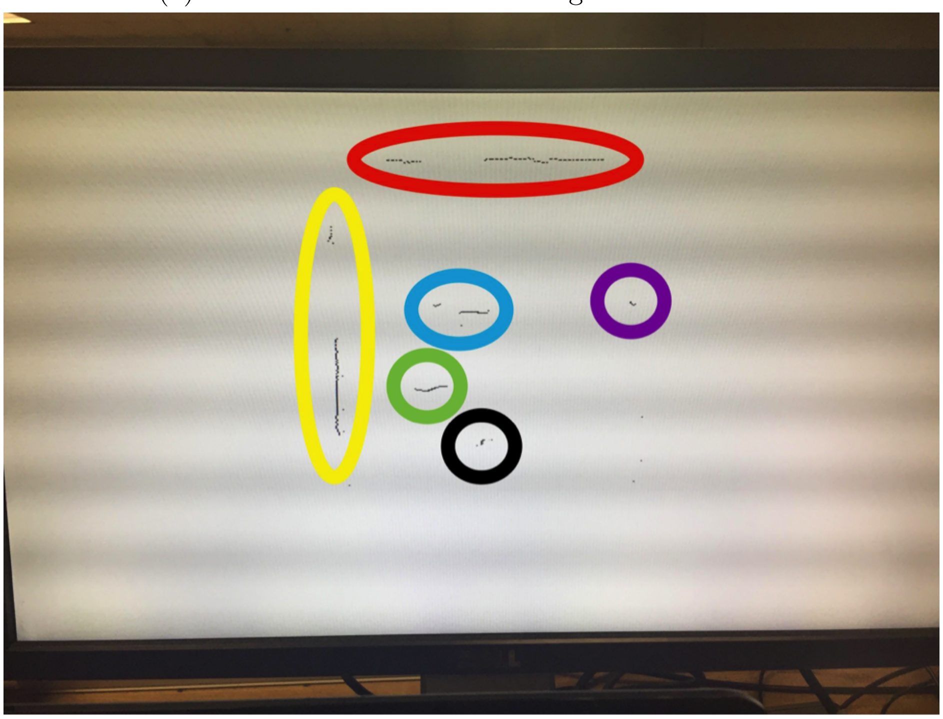
(b) 2D Rangefinder "Floorplan" of Lab at WPI

Figure 38: First Lab Test of Rangefinder Functionality

In Figure 38(b), the data points in the black circle represent the rangefinder and the lab bench oscilloscope that is right next to it. The straight line circled in yellow is the wall directly to the right of the lab bench. The lines circled in red show the lab doorway with the door open. The data in the green circle is the student in Figure 38(a), and the lines in blue are the oscilloscopes and computers in front of the rangefinder.

Next, the screen was cleared, the rangefinder was rotated 180°, and the process was repeated.

(a) Lab at WPI with 180° Change of Orientation



(b) 2D Rangefinder "Floorplan" of Lab at WPI with 180° Change of Orientation

Figure 39: Second Lab Test of Rangefinder Functionality

In Figure 39(b) the black circle is around the rangefinder, the yellow is circling the wall, and the blue is circling the computer and oscilloscope. The green is circling my body while the rangefinder's data capture was being triggered[8]. The red is circling the windows, and the purple is circling a support beam in the middle of the lab.

Next, the screen was cleared once more. The rangefinder was moved to the first orientation once more to capture data, and then was rotated to the second orientation without clearing the screen. Figure 40 shows the resultant VGA output.

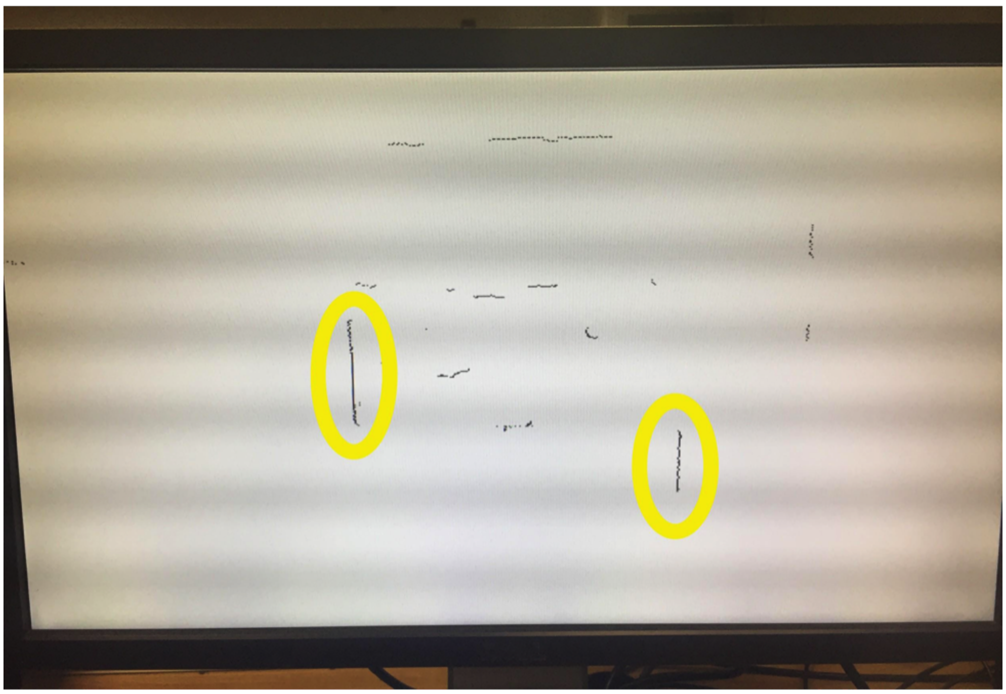


Figure 40: Two Overlaid Rangefinder "Floorplan" Captures with 180° Offset

Both rangefinder data captures were triggered successfully and without loss of data. Despite the 180° change of orientation, both data captures were overlaid on top of each other. Note that the walls circled in yellow are actually the same wall. This issue was resolved by incorporating the IMU's rotational data. With the IMU's rotational data used to offset the direction of the device, 2D floorplan accurately reflected the relative location of objects.

[8]Note that I am not shown in Figure 39(a) because I moved in order to take the photo.

5.2 IMU Testing

The PmodNAV IMU is a small device that can be directly connected to the ZedBoard's Pmod connector. As such, it requires no external power source or other intermediate connections.

5.2.1 Communication Testing

With the rangefinder connected to the ZedBoard's PS MIO Pmod, JE, the PmodNAV IMU requires an Extended MIO Pmod so that it could still be controlled by the PS. As such, the IMU's SPI pins were routed to the JD Pmod. Since the main point of communication with the IMU is its magnetometer, the IMU's slave select and register settings were adjusted accordingly, as discussed in Section 4.5.3. To choose the magnetometer and deselect the accelerometer/gyroscope and barometer, the magnetometer's slave select was brought low for each SPI transfer while the other two were left high. The behavior of Pmod JD's pins were observed with an oscilloscope during an SPI transfer. We noticed that as soon as the magnetometer's slave select line was asserted there was unidentified behavior with all of the other pins. Since the IMU was disconnected we believed this issue to be with routing the SPI pins to EMIO incorrectly, so we decided to test the SPI transfer via Pmod JE.

The pins were re-routed to the MIO Pmod JE and more undefined functionality was observed when the magnetometer's slave select line was asserted. The ZedBoard's SPI errata was investigated until we found AR# 47511, which describes an unresolved issue in the MIO interface where the SPI controller resets itself when slave select 0 signals asserts [30]. This errata was the cause of the EMIO's undefined behavior too, since this issue affects the SPI controller itself.

Luckily, the PmodNAV supports I^2C communication. To avoid this errata entirely I^2C was used and routed to Pmod JD. We began testing the IMU's I^2C behavior only to realize that the PmodNAV's magnetometer is inaccessible through the I^2C bus, which is undocumented in the device's datasheet [20]. As such, SPI needs to be implemented.

The SPI pins were again routed to Pmod JD. Attempting to avoid the errata, slave select 1 was assigned to the magnetometer. The other two slave select pins were routed away from Pmod JD so that they would not be configured or used in any manner. In their place are two GPIO pins configured as pull-ups so that these pins idle high and never have to be written to. This setup was configured in Vivado's Synthesis tab, as shown in Figure 41.

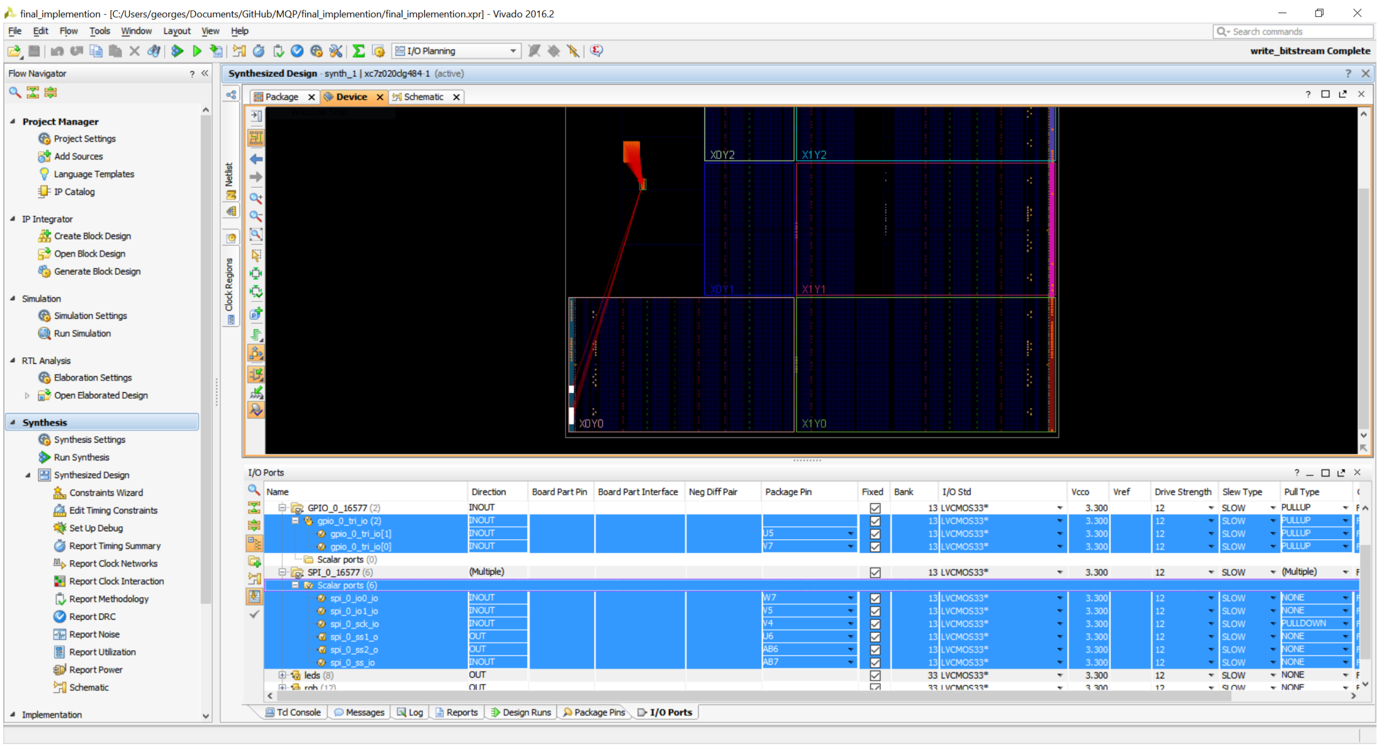
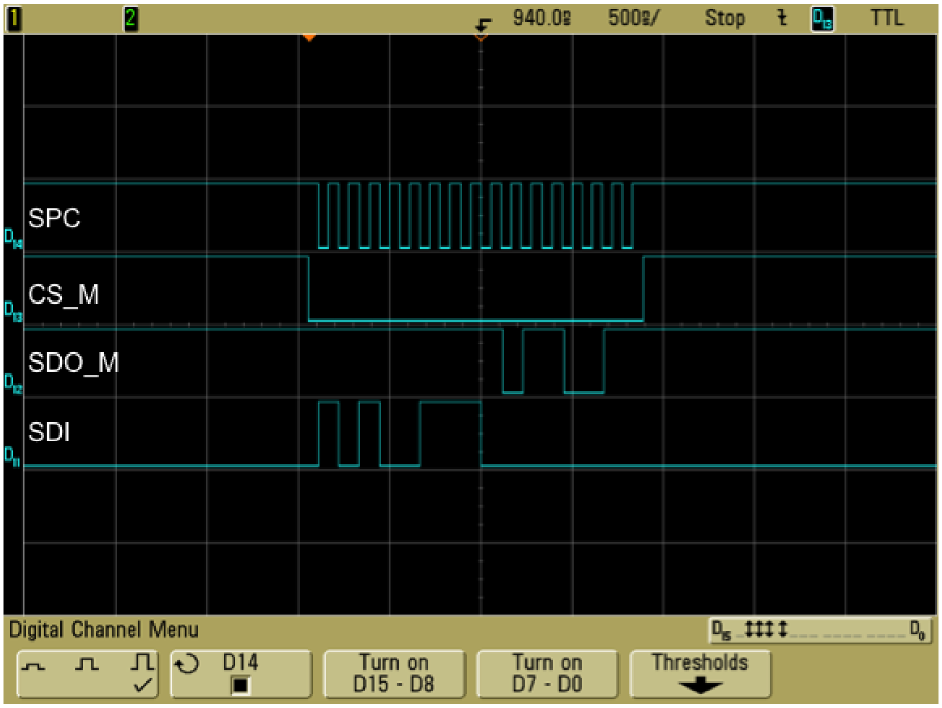
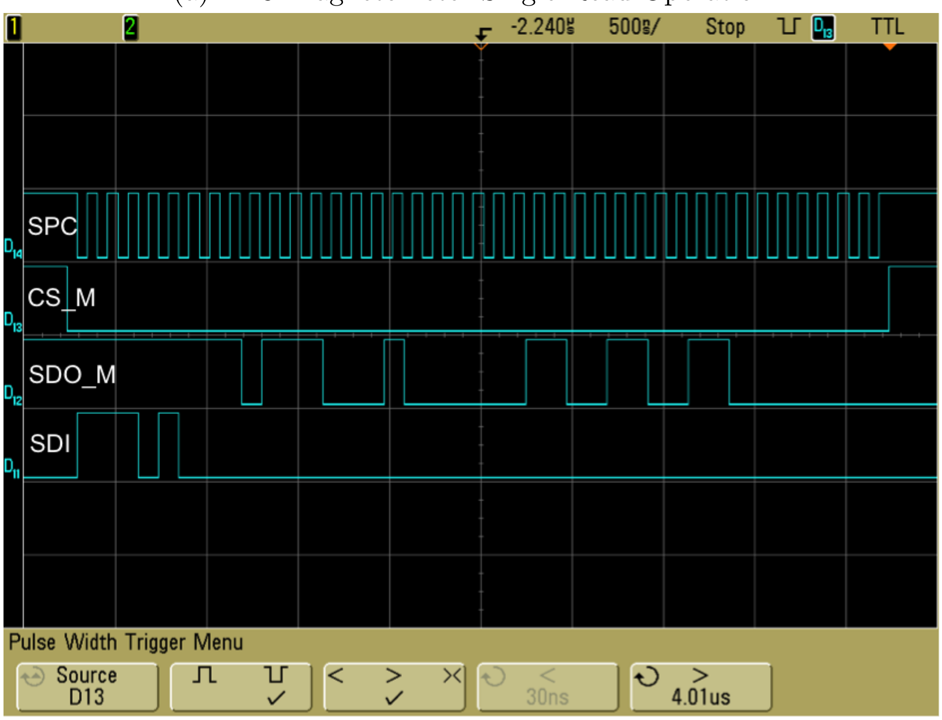


Figure 41: EMIO SPI Configuration for PmodNAV

With this configuration a waveform similar to the correct waveforms, shown in Figures 17 and 18, were observed. The IMU was connected to the ZedBoard and successful communication was observed on the oscilloscope, as shown in Figure 42.



(a) IMU Magnetometer Single Read Operation



(b) IMU Magnetometer Multiple-Byte Read Operation

Figure 42: Successful IMU Communication via EMIO SPI

5.2.2 Data Testing

The IMU's data was tested by transmitting it via UART. The magnetometer's data was transformed into a compass heading. The ZedBoard's UART was routed to USB UART so that it connects with a serial console. The ZedBoard, with the IMU connected, was rotated while the compass heading was being observed. The results were inaccurate and inconsistent until the ZedBoard was moved as far from the lab bench as the wires would allow. An IMU is a very sensitive piece of equipment that picks up electromagnetic interference, in this case most noticeably by the ZedBoard's own power supply. Once the device was further away from the lab bench, accurate and repeatable compass headings were observed. The further the ZedBoard and connected IMU were from the device's power supply, the better it performed.

5.2.3 Interfacing with the ADIS16375 IMU

Although we were ultimately able to interface with the PmodNAV IMU, this was not our first choice IMU. We originally attempted to interface with the ADIS16375 Six Degrees of Freedom Inertial Sensor, shown in Figure 43.

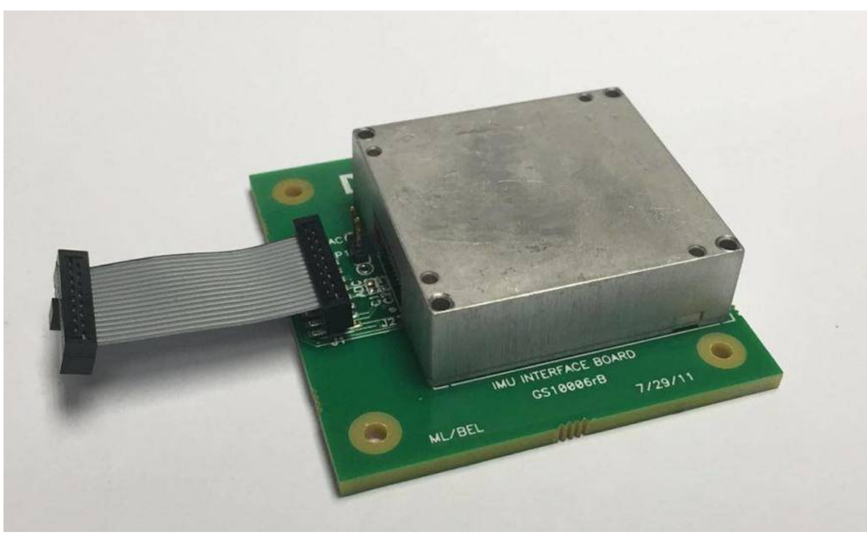


Figure 43: The ADIS16375 Six Degrees of Freedom Inertial Sensor [1]

This IMU is a highly sensitive, heavy duty device. It has a tri-axis gyroscope, a tri-axis accelerometer, and a temperature sensor, has onboard functionality to calculate delta-angle and velocity, and uses SPI communication. Although the ADIS16375 does not have a magnetometer, its gyroscope's sensitivity combined with its sample rate and onboard delta-angle calculation make it perfect to accommodate for device rotation. In addition, the device's onboard velocity calculation is extremely useful for displacement calculations. However, we were not able to communicate with it. There was no way to test the ADIS16375 so we could not verify it was functioning in any manner. With a strict deadline approaching, the PmodNAV was implemented as a quick and simple solution.

5.3 Single Camera Testing

After obtaining two of the MT9V034 cameras chosen through the process described in Section 4.8.1, 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 needed to be supplied with an external Master Clock and Output Enable signal in order to operate [25]. 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 44 was also created to make the module's pins more easily accessible.

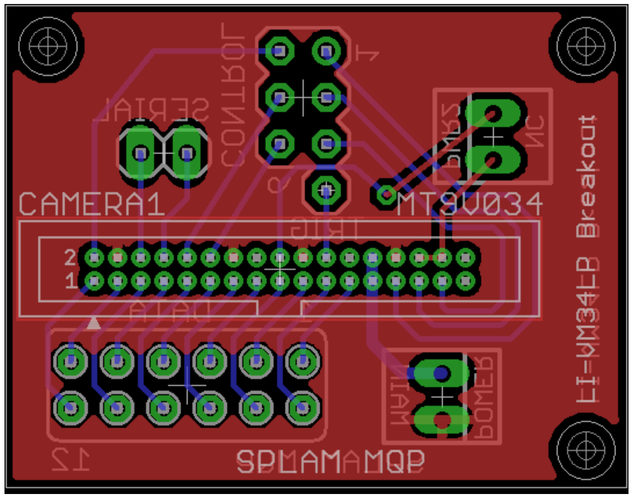


Figure 44: LI-VM34LP Breakout Board

5.3.1 I^{2}C Control

Although the MT9V034 camera control registers are closed source, the previous model's registers were available in the camera module datasheet, and were found to work with the current model [24]. As a baseline, the camera module was sent a read request at address 0x00, which should have returned 0x1324 for the MT9V034 camera module. An oscilloscope screenshot of this request is shown in Figure 45, 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.

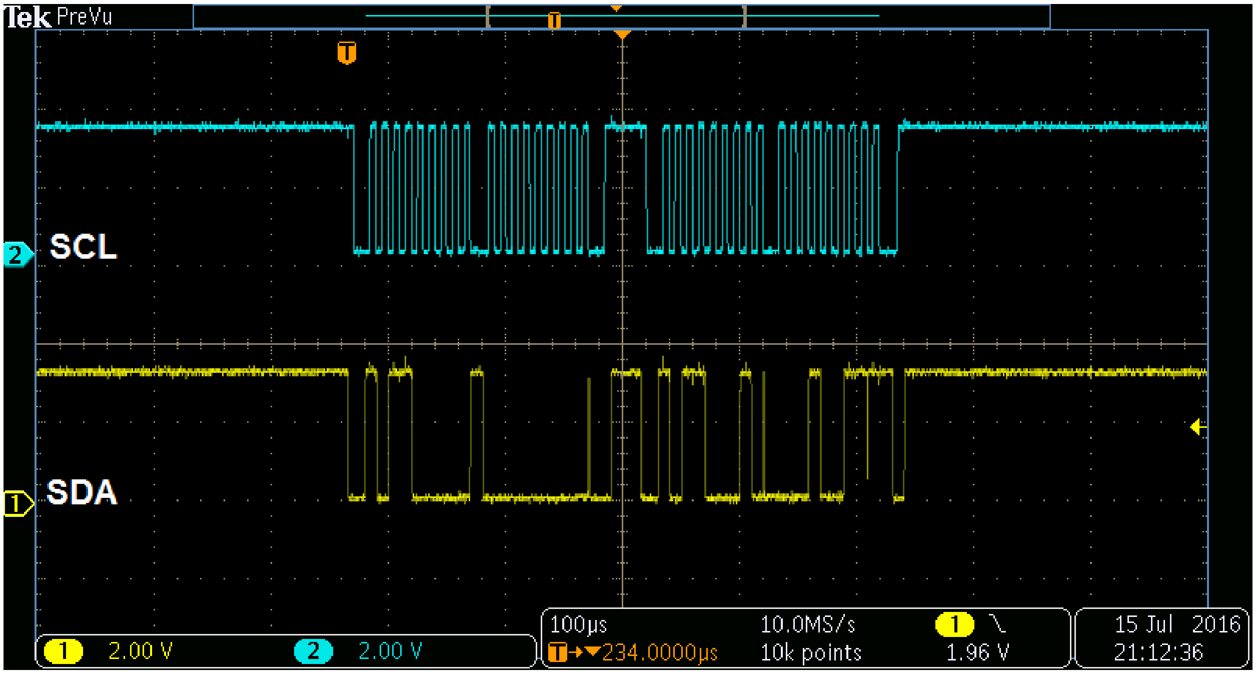


Figure 45: Example I^{2}C Transfer with Camera

After the camera I^{2}C was deemed working, the camera control register needed to be modified to put the camera in "snapshot" mode. In this mode, the camera module would no longer continuously take pictures, and would only gather new images when an external trigger was activated. This is the mode in which each camera needed to operate 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 would be set to a value of 0x0388, which corresponds to master mode with parallel output and simultaneous readout of pixel data enabled [24]. In order to put the camera in trigger mode, the control register needed to be written with value 0x0198, which allowed 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 C [24].

A button input was then attached to the camera's TRIGGER input line, and the TRIG-

GER and FRAME_VALID lines were observed on channels one and two of the oscilloscope, as shown in Figure 46. This oscilloscope screenshot can be seen as an example of how the camera was no longer in continuous operation, since FRAME_VALID only asserts itself in response to a TRIGGER input.

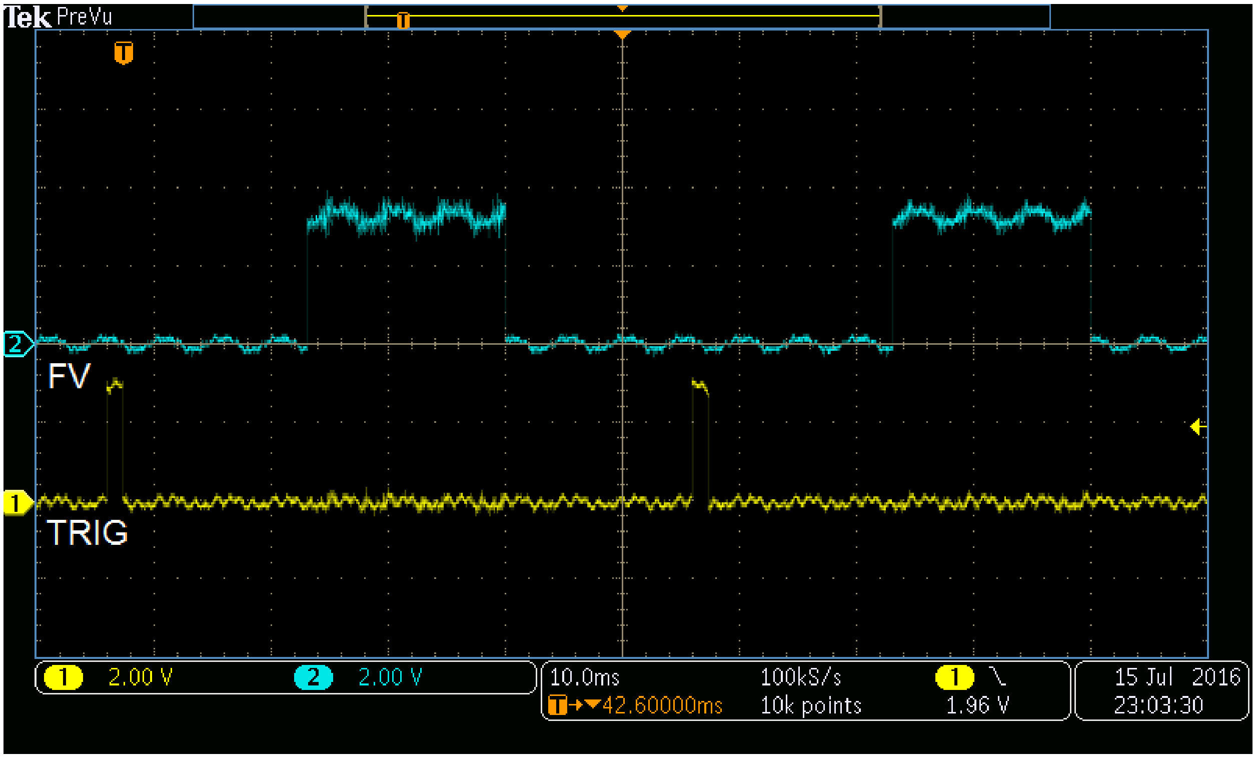


Figure 46: Camera Trigger and FV in Trigger Mode

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

5.3.2 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 was 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 47 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 had been stored on the FIFO, the FPGA was capable of reading the image line-by-line into an internal buffer. An internal System on Chip (SoC) was used to control FPGA reads from the FIFO into this internal buffer. An image dump would begin when the SoC microcontroller signaled to the FPGA to read a new line of pixels into its internal 8-bit by 752-address pixel buffer. The FPGA would then signal to the microcontroller when this buffer had been filled, and the microcontroller would 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 finished printing out the value of each pixel in the line buffer, it would signal to the FPGA to read in a new line of pixels. This process repeated 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 E.i.

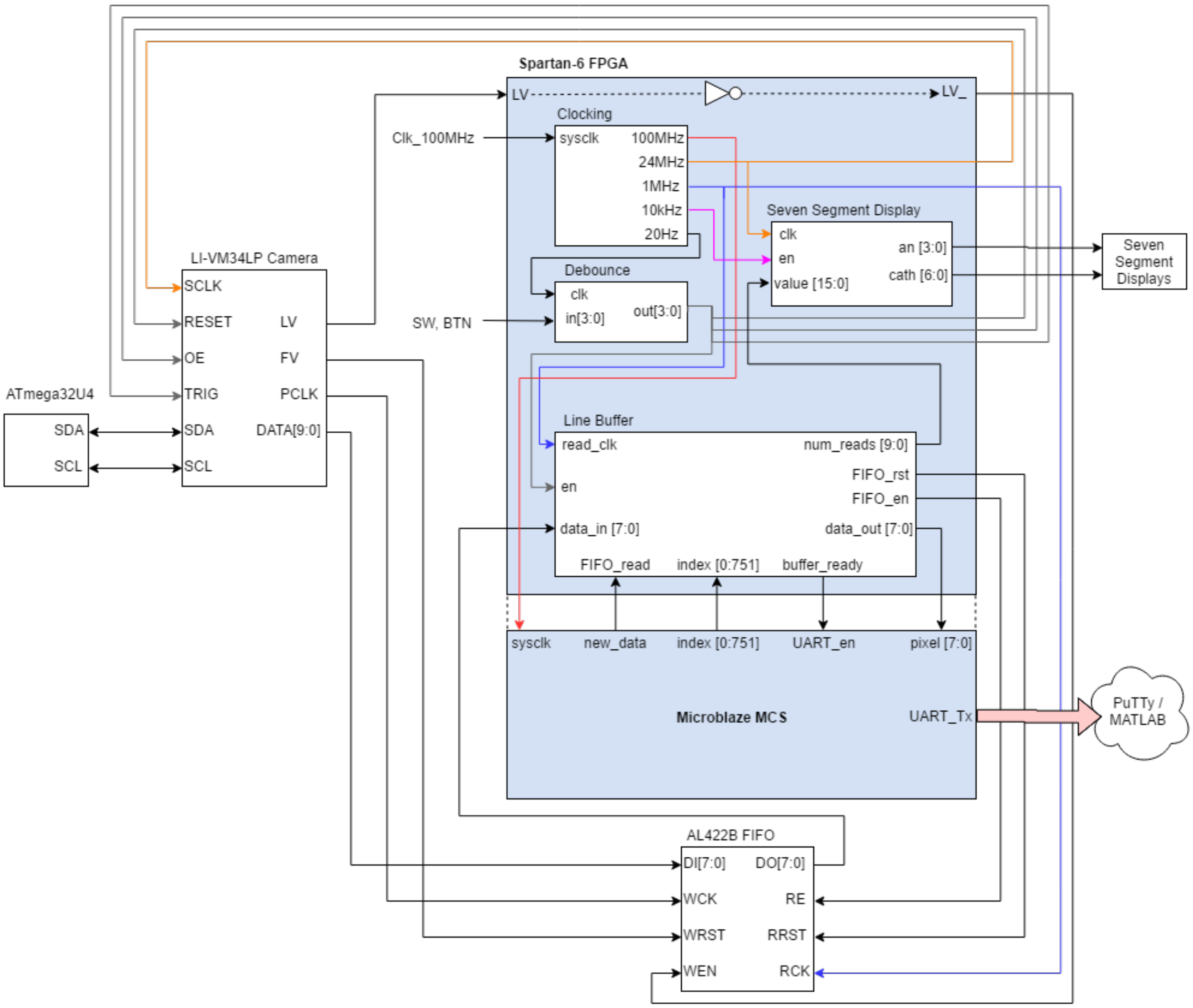


Figure 47: Camera Test System Block Diagram

An example of the transmission of one line of pixel data from the FIFO to the FPGA is shown in Figure 48. 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 read in one line of pixel data at a time, this process took 752 read clock cycles, as measured in Figure 48. In order to simplify debugging, an internal counter and seven-segment display controller were implemented on the FPGA, and would display a running count of the number of pixel lines that were read into the FPGA's internal buffer, ranging from 0x0000-0x01E0 (0-480).

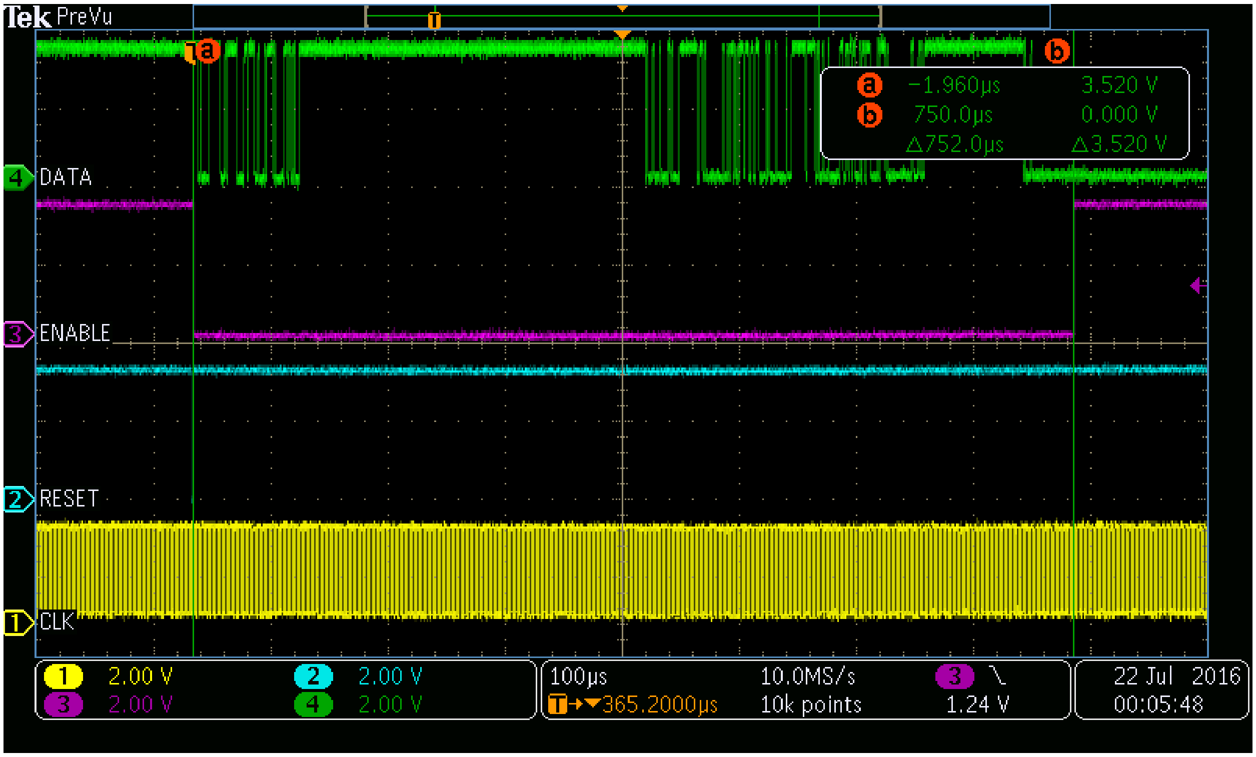


Figure 48: Transferring Line Data from FIFO to FPGA

5.3.3 Transmitting Images Over UART for Analysis

Once the FIFO and FPGA line buffer interfaces were created, the source code found in Appendix item E.i 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 49. The microcontroller would print the value of each pixel followed by a newline and carriage return, starting with the top left pixel in the acquired image.

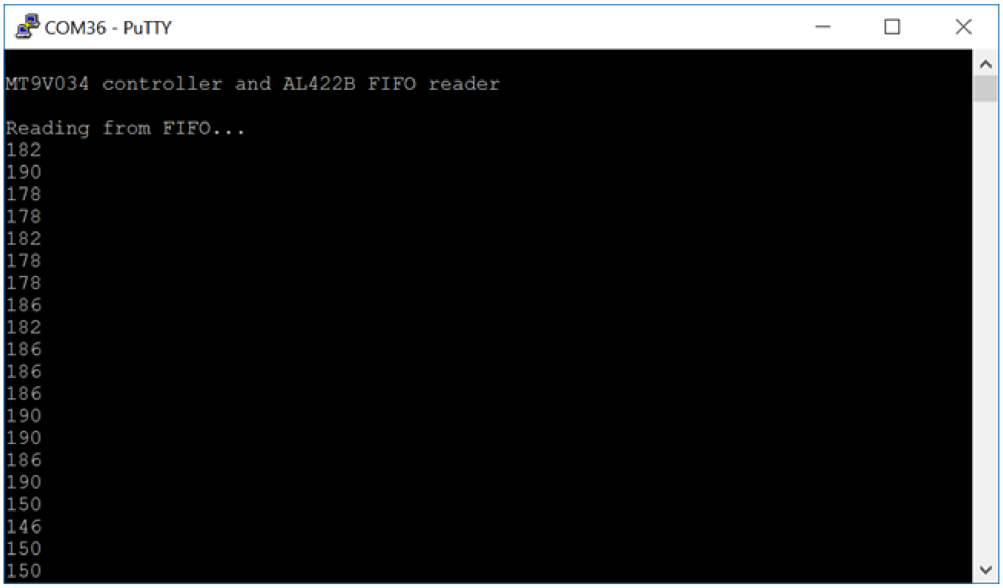


Figure 49: Reading FIFO Data

After the image was received through PuTTy, the MATLAB script found in Appendix item E.i was used to parse the corresponding logfile into a greyscale image. An example image created through this process is shown in Figure 50. Note that the sub-optimal quality of this image was due to signal interference and degradation in the test setup's long wiring, as shown in Figure 51.

Figure 50: 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 made it so that it could easily be implemented on any system, including the Zynq family of processors that were used in the final system implementation.

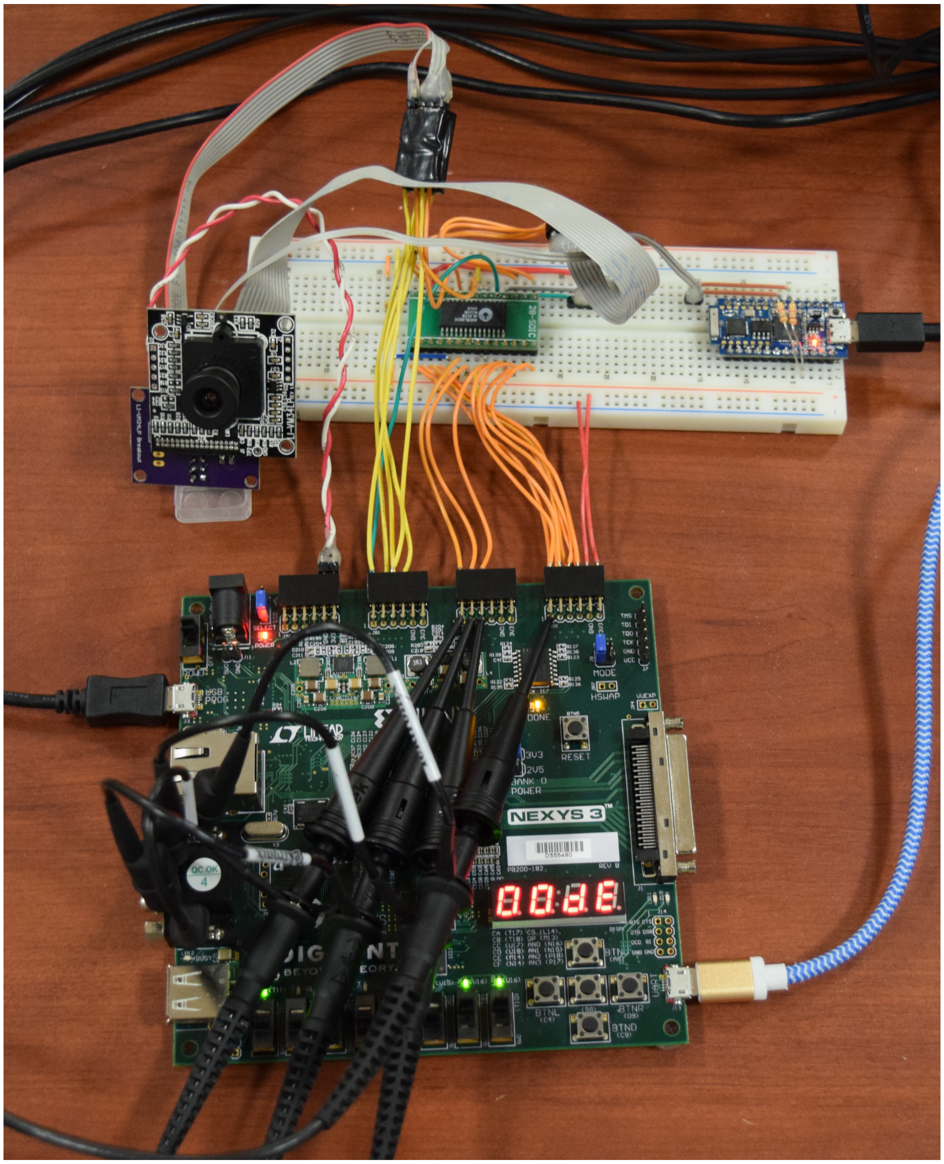


Figure 51: Camera Test Setup

5.4 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. This implementation had several design constraints, as it needed to successfully interface both cameras with the ZedBoard as a stereo pair without consuming too many pins.

5.4.1 Stereo Camera Breakout Board

Although it was possible to interface each camera module directly to the ZedBoard's GPIO using the camera breakout, this setup was not feasible. A pair of the original camera breakout boards shown in Figure 44 would have consumed every available Pmod pin on the board, leaving no additional pins for the IMU or rangefinder[9]. One solution originally investigated was the use of the ZedBoard's FPGA Mezzanine Card (FMC) connector, since it contained 68 available GPIO pins and would be more than adequate for interfacing the stereo cameras with the board. However, the FMC connector was configured to provide logic voltage levels of only 1.8 or 2.5 volts without modification to the ZedBoard. Since each camera module was only compatible with 3.3 volt logic, the FMC connector was therefore not feasible for our designs.

This left 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 was to include an individual AL422B FIFO per camera. Based on the testing described in the previous section, it was already determined that these FIFO modules were compatible with the MT9V034 cameras, and were capable of significantly reducing memory requirements on the FPGA. A second major advantage of including these FIFO modules in the camera interface was that their data output lines could be placed in a high-impedance state. This meant that the individual data output lines of each FIFO module could be connected in parallel, with a single FIFO driving the lines at a

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

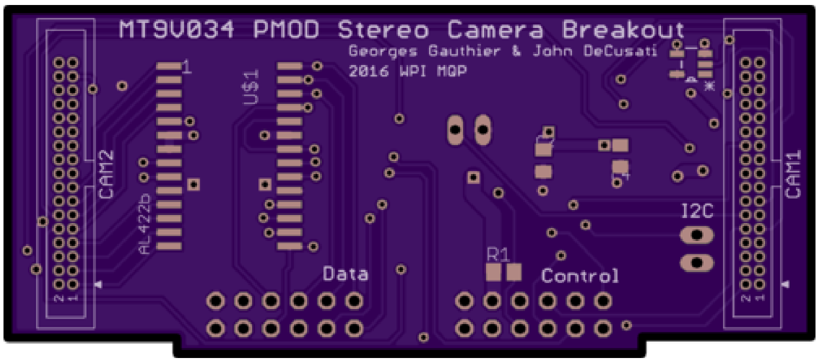
time. Since the bulk of each camera module's required pin count lied in its data lines, the ability to connect these lines in parallel reduced the overall camera GPIO requirements by 8 pins. Since each AL422B FIFO module was capable of being read from at a clock speed of up to 50MHz and the maximum master clock rate of each MT9V034 camera module was 27MHz, the inclusion of the FIFO modules also didn't cause a significant decrease in the overall speed of the stereo camera system [5, 25].

Along with the shared camera data lines between each AL422B FIFO module, it was 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 was already desirable to connect both camera TRIGGER lines together. The RST, OE, SDL, SCA, and SCLK lines of each camera module could also have been tied together in pairs of two, and the OE lines could simply be held at 3.3 volts. Lastly, since each camera LV signal needed to be inverted for use with the AL422B FIFOs, a discrete inverter IC was used to save on FPGA GPIO. Overall, these modifications saved a total of 25 pins, as shown in Equation 12.

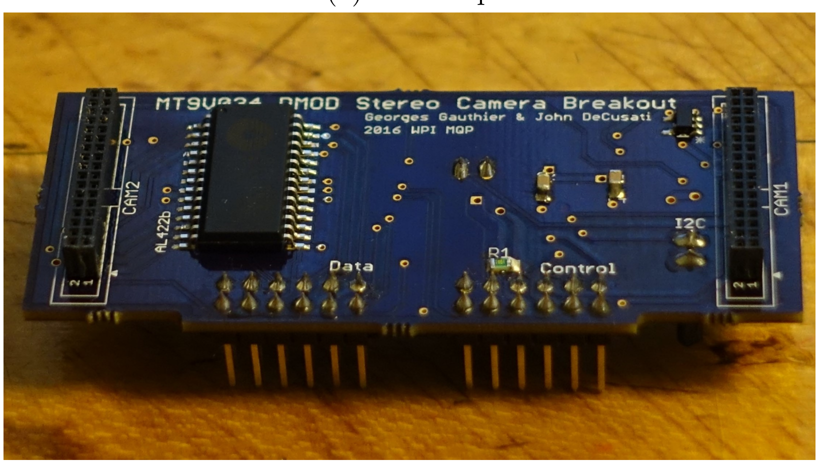
$$\begin{aligned} 36\ Pins - (8\ Data + 4\ truncated\ bits) - (TRIGGER + SCLK + RST) \\ -2 * (OE + PCLK + FV + LV) = 13\ pins\,(!) \end{aligned} \tag{12}$$

Note each FIFO needed to 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 brought the total pin count required by the stereo camera setup to 16 pins plus two I^{2}C pins, which was conveniently the number of GPIO available in two Pmod headers. This setup was implemented as shown in Appendix item D, and the final stereo camera breakout board shown in Figure 52 was then created.

A Verilog module was created using a modified version of the MT9V034 camera test code found in Appendix item E.i and a VGA controller in order to test the stereo camera breakout board on the Nexys3 platform. A switch input was used to select one of the two camera modules for image acquisition, and a binned 60x92 pixel set from the center of the camera's image was buffered locally for VGA display. The image was then independently written to the display according to internal VGA timing. This process was 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 53.



(a) PCB Top



(b) Assembled PCB

Figure 52: Stereo Camera Pmod PCB

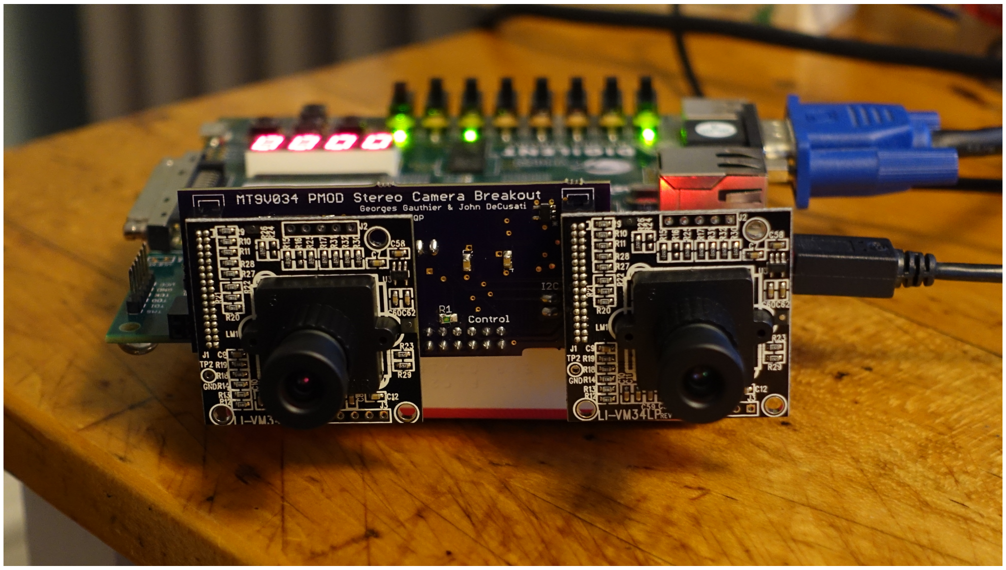


Figure 53: Stereo Camera Breakout Under Test

After attempting to manually focus each camera using the VGA module described above, the code used in Section 5.3.3 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 54 below, the new stereo camera setup was far less susceptible to data loss in comparison to the previous version. For further comparison, please refer back to the test image acquired using the original camera test setup, as shown in Figure 50.

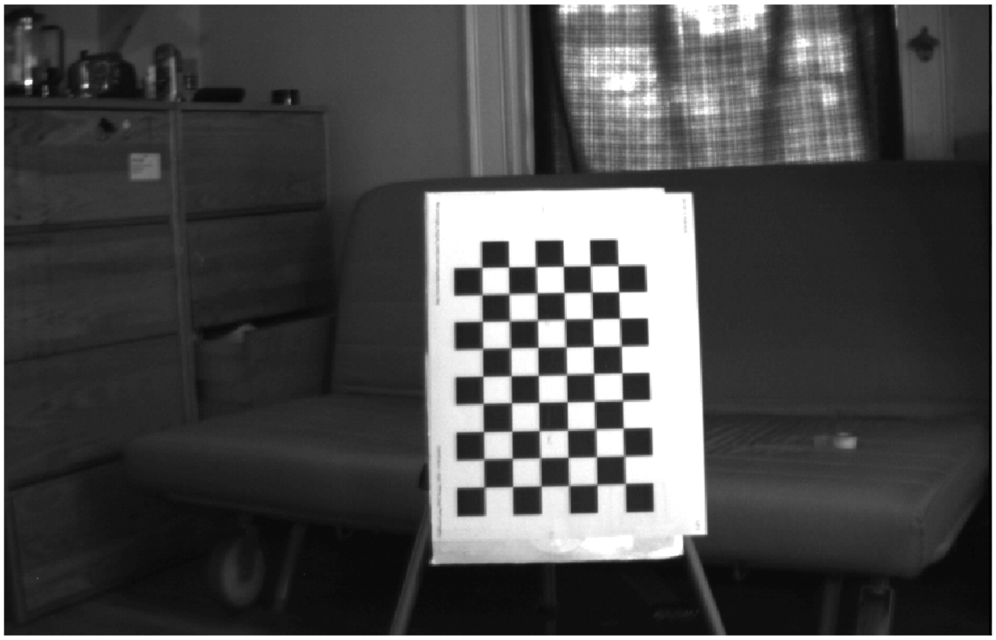


Figure 54: Stereo Camera Breakout Sample Image

5.4.2 Image Buffering

After the camera setup was deemed working based on the results of the Nexys3 test implementation, a finalized camera controller module was created for the ZedBoard. This began with the simple implementation shown in the block diagram in Figure 55 below. This implementation contained a customized camera controller IP based around the same code used for creating the camera controller described in Appendix Item E.i, with the exception that internal BRAM was used to buffer an entire image captured from the cameras. Note that a custom AXI interface was also included in the test implementation, allowing for the option of reading image data into the Zynq Processing System for more advanced testing and export via PS peripherals such as UART.

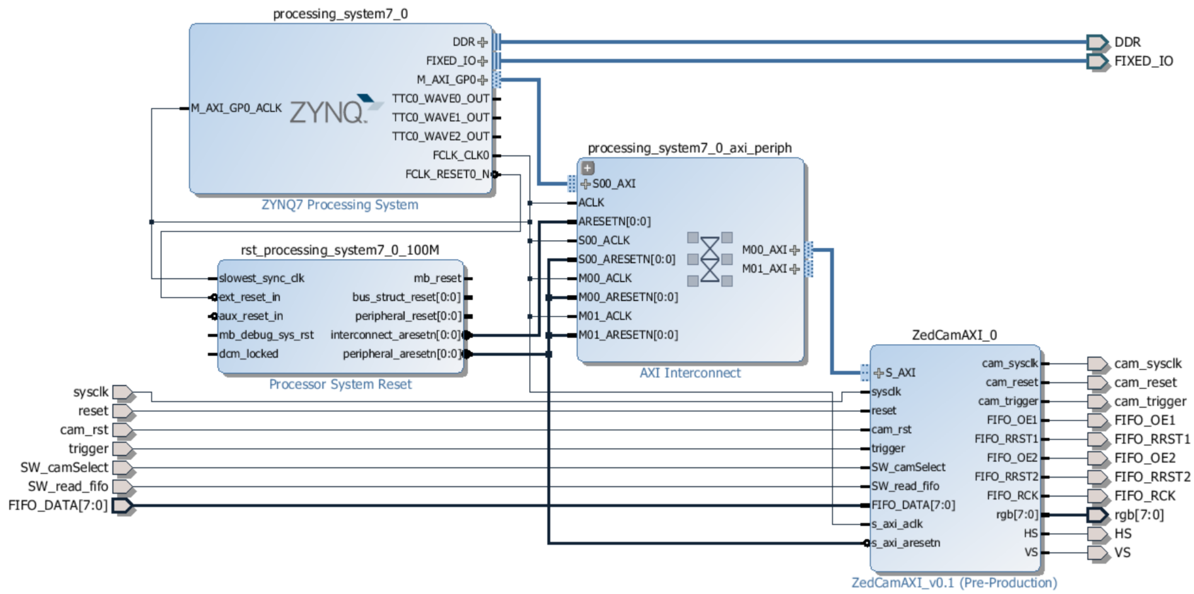


Figure 55: ZedBoard BRAM Camera Test Block Diagram

Since a single camera image contained 8 bits $\times$ (752 $\times$ 480) pixels, a simple dual-port BRAM module containing $752 \times 480 = 360960$ 8-bit addresses was created for storing the output of the AL422B FIFO reader module. Dual-port BRAM was used to allow for external VGA logic to read from the image buffer without the need for read/write protection. Overall, the purpose of this implementation was to test the capabilities of the ZedBoard's internal BRAM for image buffering, as well as to get a simple visual confirmation via VGA output that the implementation was working. An example of the output from this implementation is shown in Figure 56 below.

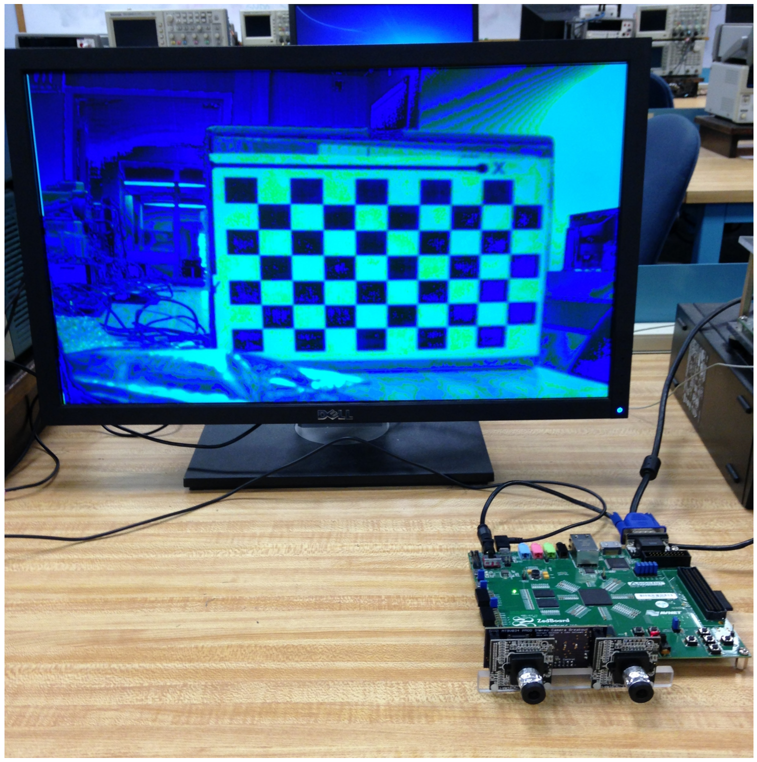


Figure 56: ZedBoard BRAM Camera Test

After determining that the Zynq Processor's internal BRAM would be usable for storing image data, several tradeoffs associated with the memory requirements of buffering image data in BRAM needed to be addressed.

5.4.3 Resource Management

One major issue encountered while dealing with resource management on the ZedBoard was the usage of Block RAM. The Zynq7 processor used on the ZedBoard contained 140 individual blocks of 36Kb BRAM, which is equivalent to 630,000 8-bit bytes of memory [33]. Although this was plenty of memory for buffering a single 752x480 camera image, three separate image buffers need to be implemented in BRAM for this project. Two of said memory buffers were used for storing left and right camera images for processing by the disparity algorithm, and a third was used for storing a resultant output image that could be displayed via VGA.

In order to address this issue, input camera imagery was centrally windowed to a resolution of 384x288 pixels, or $0.6 \times VGA$. The output display buffer was also been reduced from WVGA (752x480) to VGA (640x480). In total, this resulted in the use of 27 36Kb Block RAM modules for each of the windowed left and right camera images, and 75 36Kb Block RAM modules for the VGA display buffer. Note that each buffer was configured using an individual Block RAM IP, and each consisted of a simple dual-port RAM with an 8-bit data length. Overall, this implementation consumed 129 out of the 140 available 36Kb Block RAM modules available on the ZedBoard's Zynq7 processor, leaving additional resources for use in the IMU and rangefinder implementations.

5.5 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.

5.5.1 Image Rectification

In order to perform the most accurate block matching as possible on camera image data, it would have been ideal to rectify the images as outlined in Section 4.9.1. However, since the image data used in the disparity calculation contained a central 384x288 image taken from the middle of each 752x480 input image, a large portion of the input image was cropped out. Since many of the lens artifacts corrected using a rectification process are contained on the external edges of the input imagery, no additional camera calibration was performed in the disparity or camera controller implementations [7].

After extensive testing with the stereo camera breakout board and disparity module detailed in the following sections, it was also found that the camera imagery captured through the stereo camera interface contained consistent horizontal epipolar lines between both images. These lines are accurate enough for the custom disparity algorithm to process without additional calibration, saving a large amount of calibration time in the image processing pipeline. Along with the aforementioned reasons, a full camera calibration and image rectification process was not implemented in FPGA hardware due to time constraints, although a cursory calibration process was examined in Matlab. With the issue of camera calibration and image rectification addressed, it was then possible to begin implementing a test disparity algorithm in Matlab.

5.5.2 Matlab Implementation

The Sum of Absolute Differences algorithm discussed in Section 4.9.2 was first implemented using Matlab, and can be found in Appendix item E.v [23]. This implementation was created to operate on the "cones" standard test image set, and produced a resultant disparity image from its given input images. In the case of this specific example, the algorithm performed a 7x7 Sum of Absolute Differences block matching process on 50 search ranges of horizontal epipolar lines between the two images. However, the block size and search range could be customized by the user to test the functionality of the algorithm.

Overall, the Matlab disparity test implementation may be broken down into the following steps:

1. Load in image data (also convert to grayscale if using the "cones" image set)
2. Determine the size of the template image and create a resultant matrix to store output disparity values in
3. For each full row of pixels across an image, perform the following steps:
    (a) Set minimum and maximum row bounds for the current block of pixels being used for SAD
    (b) For each column in the given row, perform the following steps:
        i. Set minimum and maximum column bounds for the current block of pixels being used for SAD
        ii. Determine the number of blocks that will be used in the current search. Note that this number will be the Disparity Range until the blocks being searched are closer in pixels to the right edge of the image than the Disparity Range

iii. Create a memory block for holding the SAD value for each block comparison based on the number of blocks from (ii), and create a template block from the right image at the current column/row

iv. For the number of blocks calculated in (ii)

   A. Compute the Sum of Absolute Differences for each left image block along the current pixel row with respect to the right image template block, and store the calculated value in the memory block created during (iii)

v. Find the smallest value in the memory block containing SAD values. Use the index of this block to determine the pixel offset from the template block location. This value is the disparity for the particular point

vi. Store the calculated disparity value in the resultant image matrix. Go back to (b) if there are more columns (pixels) remaining in the current row, otherwise go to (3)

4. When the entire image has been iterated through, display the resultant disparity matrix, and scale pixel coloration based on the minimum and maximum disparity values for better contrast.

An example of the output of this test implementation is shown in Figure 57 below.

Figure 57: Disparity Implementation Output

5.5.3 Verilog Test Bench

The original Verilog disparity test implementation used closely follows the MATLAB disparity algorithm discussed in the previous section. This algorithm was implemented using a finite state machine with five states, as shown in Figure 59 below. In order to maintain simplicity, the test algorithm was implemented to operate on the 20x7 pixel test images shown in Figure 58. The search range and block size for this module were defined as 15 pixels and

5x5 pixels, respectively. By default, the disparity module remained in an idle state until an external enable signal was toggled high using a button input. This caused the finite state machine to advance to its READ state, and image data for the left and right camera images was then read in from the stereo camera breakout board. After image data was received, the state machine would then advance to a cyclical set of states used for iterating through each image and calculating disparity.

Figure 58: Disparity Test Images

The disparity module would 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 would advance to its SAD state, and would calculate the sum of absolute differences between the template and search block. This value was placed in a vector that matches the length of the search range. If the vector hadn't been completely filled, indicating that there were more search blocks to compare to the template, the state machine would revert back to the separate state, isolating a new search block from the right camera image. When the SAD vector was full, the state machine would then advance to its finalization state.

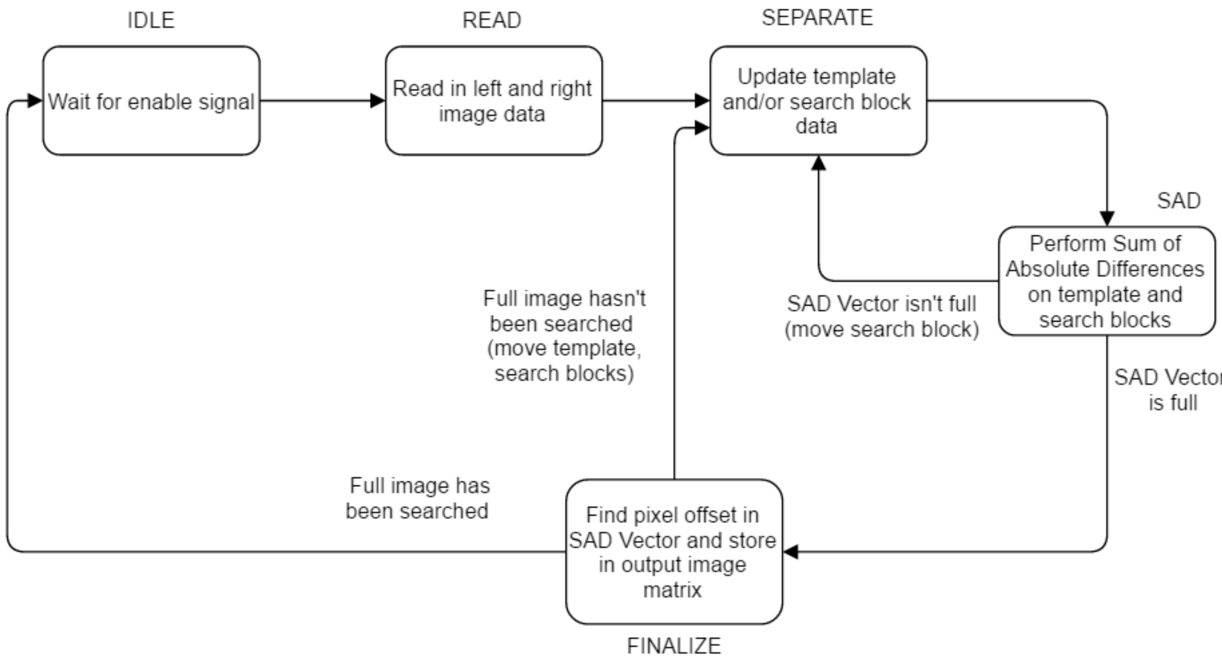


Figure 59: Disparity Test Implementation

The finalization state was 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 was used to create a disparity value for the given template block location. This value was then converted to a distance using Equation 9, and was stored in the output image location. If the output image hadn't been fully populated with distance values, the state machine would then revert back to the separate state. Otherwise, the state machine would advance to its idle state, and the resulting disparity image could 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.

5.5.4 Test Bench Results

The READ state of the disparity state machine was first analyzed using the Verilog Test Bench, and the these test results are shown in Figure 60 below. The state machine is shown transitioning from IDLE (0) to READ (1) in the beginning of the timing diagram, as shown by output indicator `state_LED`. After transitioning to its READ state, the disparity module read in each image horizontally from left to right, as dictated by `buffer_href` and `buffer_vref`. The left camera image was read first, and output `image_sel` was then toggled to signal a second read sequence from the right camera image buffer. During each rising clock cycle, input `image_data` was stored in an internal BRAM module for the associated camera's image data, with the write address based on the current value of `buffer_href` and `buffer_vref`.

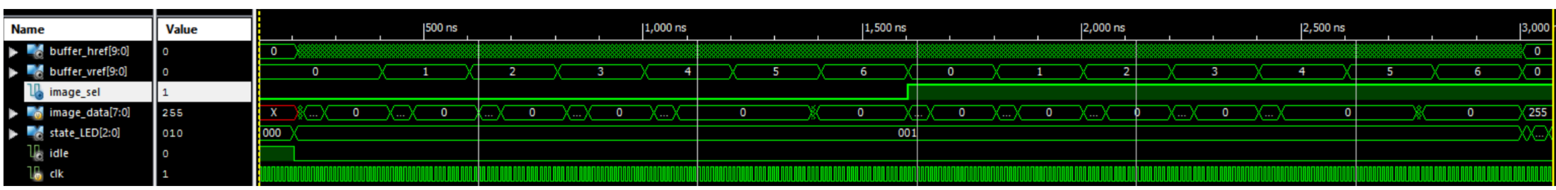


Figure 60: Image Read Sequence

After the state machine finishes reading both images into local BRAM on the Zynq7 processor, it would iterate through the SEPARATE (2) and SAD (3) states until an entire search range of search blocks had been compared to the given template block. After the search range had been traversed, the state machine would advance to its FINALIZE (4) state to find the disparity value for the given search and place the value in the output buffer. Figure 61 shows an example of this process, where a template block set by pixel row bounds `minr` and `maxr` and pixel column bounds `t_minc` and `t_maxc` was compared to 15 individual search blocks set by row bounds `minr` and `maxr` and column bounds `b_minc` and `b_maxc`.

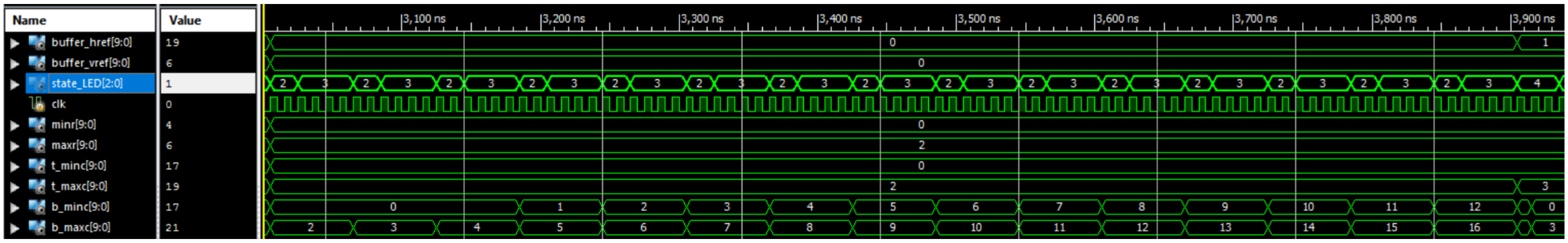


Figure 61: Disparity Search Vector

Note that in the case of the disparity search shown in Figure 61 above, a disparity value was calculated for the pixel location (0,0), or the top left corner of the image, as defined by `buffer_href` and `buffer_vref`.

After each disparity search was completed, the state machine would advance from the FINALIZE state to the SEPARATE state to isolate a new template block and search block. Internal counters for horizontal and vertical pixel location of the disparity search were also updated at this point, triggering an update of the template and search block parameters, as well as the number of blocks analyzed in the current disparity search. This number decreased as the template block got closer to the right side of the image, since the search range would eventually exceed the distance from the template block to the width (right edge) of the image. An example of multiple disparity searches across one horizontal line of pixels in the top row of the 20x7 test image is shown below in Figure 62. In the case of this example, `numBlocks` represents the number of blocks included in the current disparity search. Since the width of the image was 20px and the search range was set to 15px, `numBlocks` begins to decrease after the 4th disparity search is performed, as shown below.

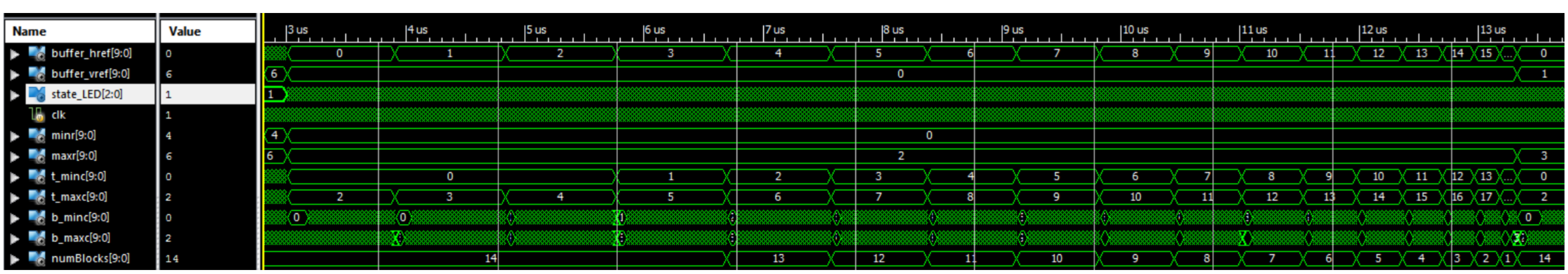


Figure 62: Horizontal Pixel Row Search

After an entire horizontal row of pixels had been analyzed by the disparity algorithm, the vertical location of the template block was increased, and the overall process was repeated continuously until the entire image has been analyzed. An example of a disparity search through the entire 20x7 image is shown below in Figure 63.

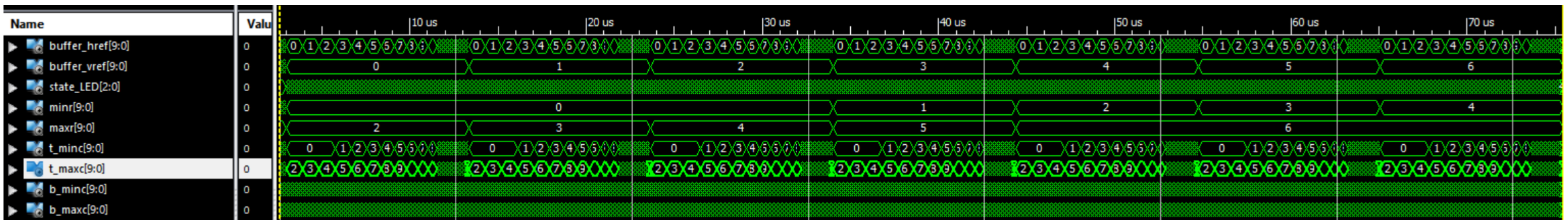


Figure 63: Full Image Search

The output disparity image of the test analyzed above is shown below in Figure 64. Note that the artifacts in the resultant image were due to the fact that a 5x5 template and search block set was being used on a 20x7 image, making it impossible to avoid the block contained in the upper left corner of the search image. The direction of artifacts around the block in the lower right corner of each image were a function of the search direction, since the search blocks descended downwards and to the left.

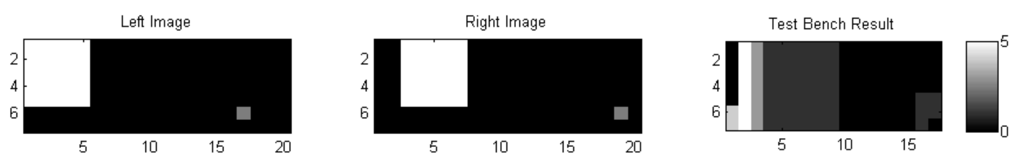
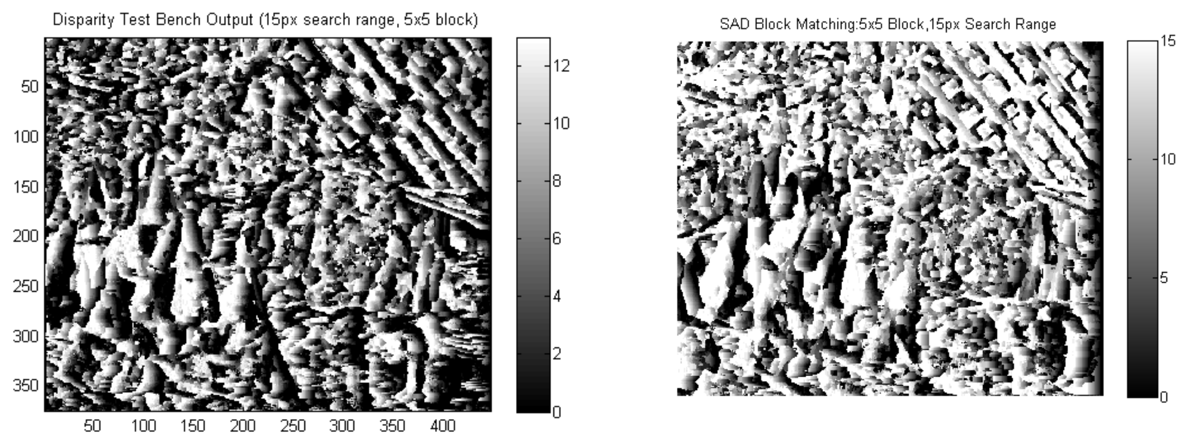


Figure 64: Disparity Test Results

After the disparity module was deemed working on the 20x7 test image, further testing on image data was performed. First, a MATLAB script for converting image data to a format recognizable by the Verilog Test Bench's `$readmemb` command was created. Using these converted images, it was then possible to compare the Verilog disparity implementation's results to those of the MATLAB implementation. Note that due to limitations in computer memory, the disparity search range and block sizes capable of being processed by the Verilog Test Bench were limited to 15 pixels and 5x5 pixels, respectively.

Figure 65 below contains a comparison between the test bench and MATLAB results for a disparity search on the "cones" test image set. Note that the outputs from the MATLAB and Verilog disparity search algorithms are noticeably close in comparison, as well as in pixel intensity. Losses in the the output of the Verilog disparity algorithm were likely due to the fact that all operations were performed using integers rather than floating point values.



(a) Test Bench Result

(b) MATLAB Result

Figure 65: MATLAB vs. Verilog Test Bench Results

5.5.5 Final Implementation

In order to increase the speed of the disparity algorithm's output, several portions of the original disparity Verilog module used in the previous section were modified to increase parallelization and decrease the overall latency associated with various calculations. Most of this parallelization was based around taking the 2D memory arrays used for storing the template and search blocks, and breaking said arrays into individual vectors. For example, instead of having a 7x7 memory array for storing one "block" of pixels, the block was broken into seven separate 1x7 vectors that could be operated on individually. This parallelization decreased the overall latency of the system by reducing time spent waiting to read from individual addresses within memory arrays. An overall block diagram of the parallelized disparity implementation is shown in Figure 66 below.

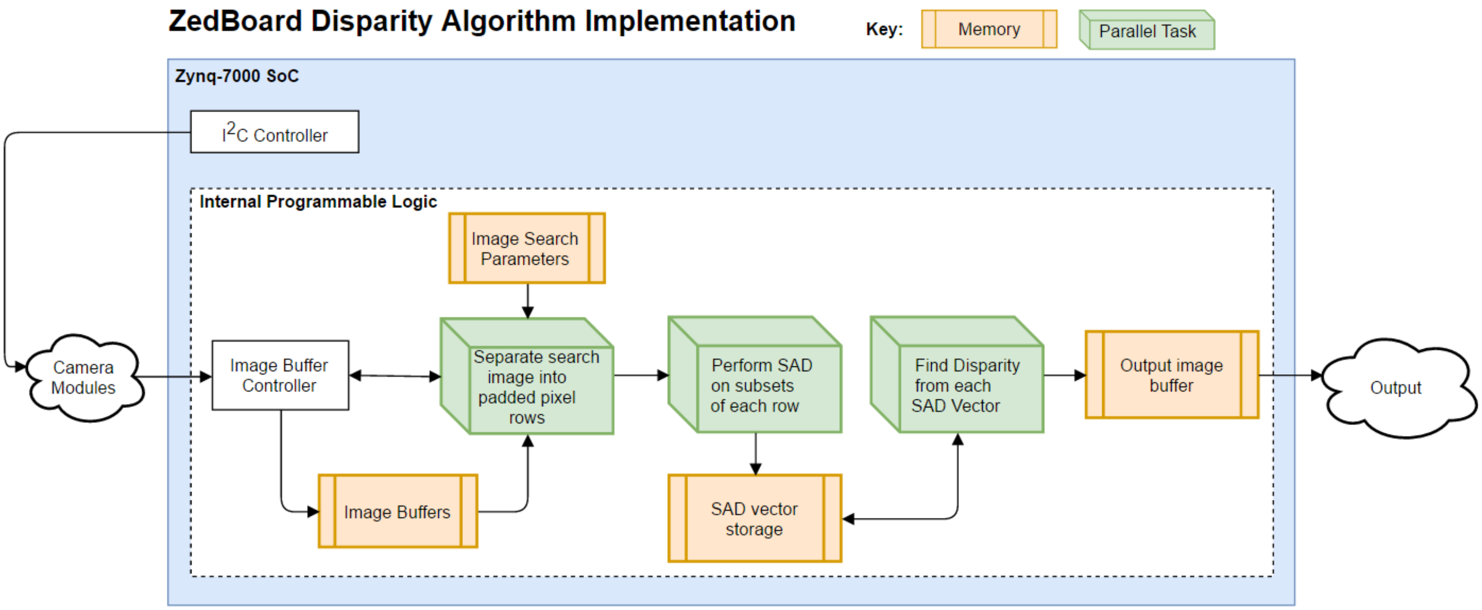
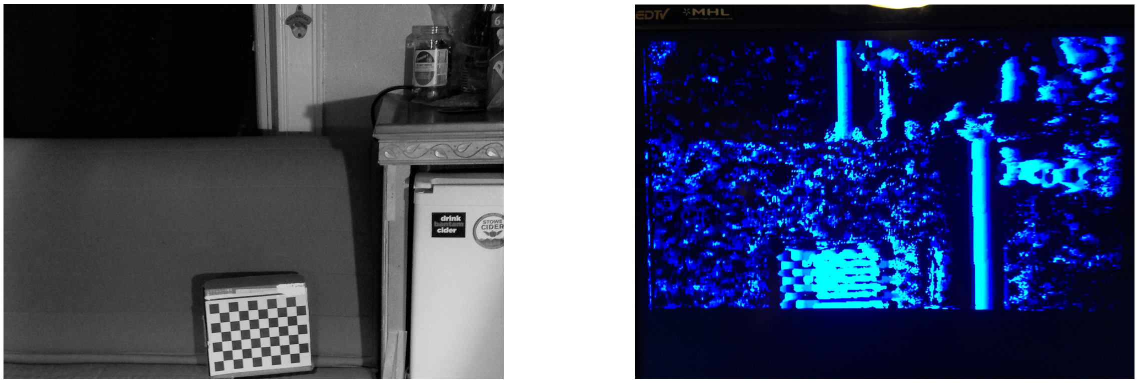


Figure 66: Disparity Final Implementation

The overall state machine used to create the final disparity implementation still followed the same next-state logic as that of the original implementation, shown in Figure 59. Advances in speed of the overall algorithm were therefore mostly associated with parallel calculations during the computation of the Sum of Absolute Differences. As discussed in Section 5.4.3, the left and right search images passed to the disparity algorithm were also windowed to $0.6 \times VGA$ resolution, further reducing computation time.

In order to verify the output of the hardware disparity implementation, a VGA display driver was used to display the disparity algorithm's output in real time, as shown in Figure 67. With the successful implementation of this output mode, it was then possible to concentrate on combining the disparity and camera controller modules into a final design that incorporated the IMU and Rangefinder modules.



(a) Device View (b) Resultant Disparity

Figure 67: Disparity Algorithm Output

5.6 Full System Operation

After fully testing each of the individual sensors used in this project and developing a working disparity algorithm, the individual modules created for each test were combined. The result of this finalized implementation was a proof of concept SLAM implementation capable of generating compass-referenced 2D floorplans of its surroundings, as well as 3D depth maps of its entire field of view. The output modes of this device were user-selectable, and could be viewed on an external VGA display. An overall system block diagram of the final implementation of this project is shown in Figure 68.

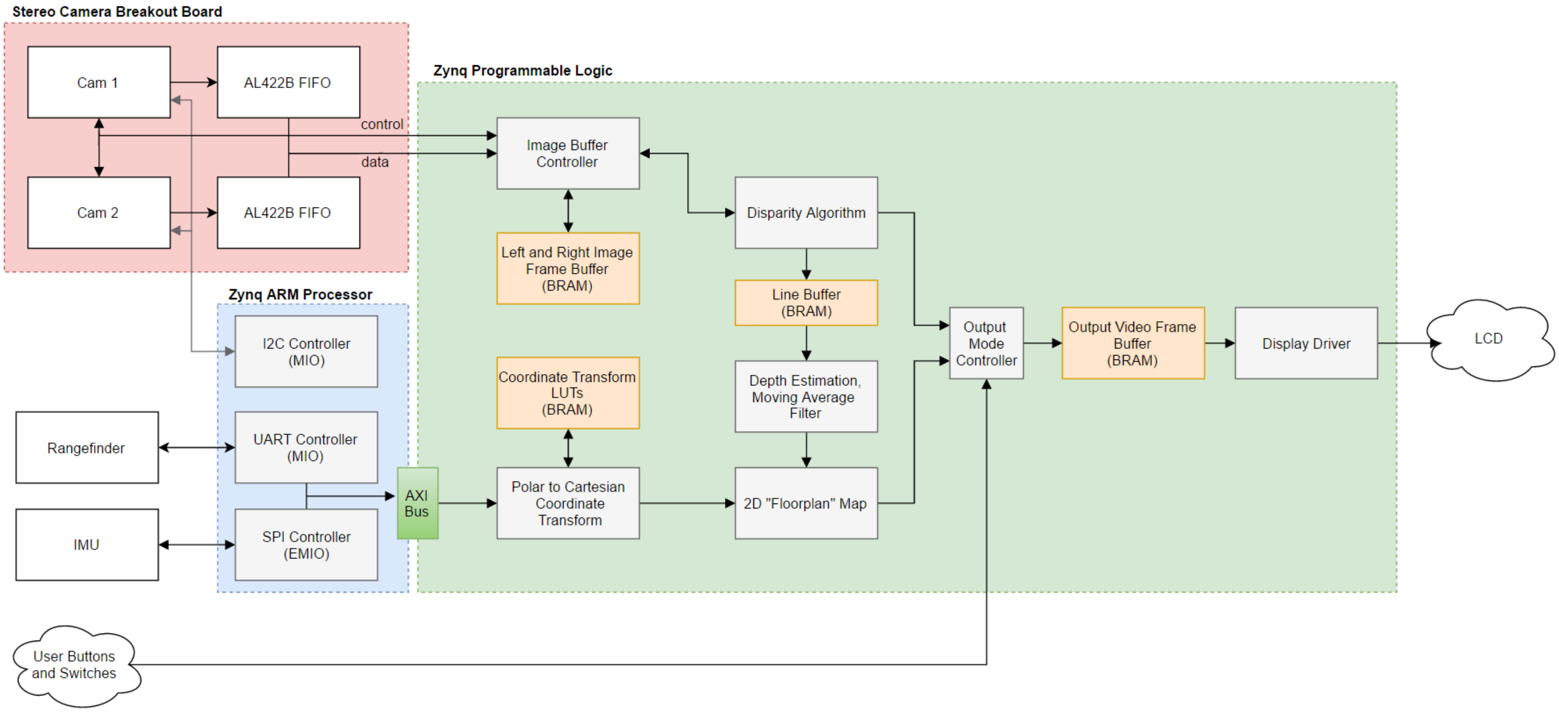


Figure 68: Final System Block Diagram

This implementation was broken down into several major components. At the top level, several blocks were used to connect the Zynq7 ARM Cortex A9 processing system to customized programmable logic, as well as to peripherals such as the rangefinder and IMU by using an AXI peripheral controller, SPI, and SPI/GPIO, respectively. The programmable hardware version of this implementation may be found in Figure 69.

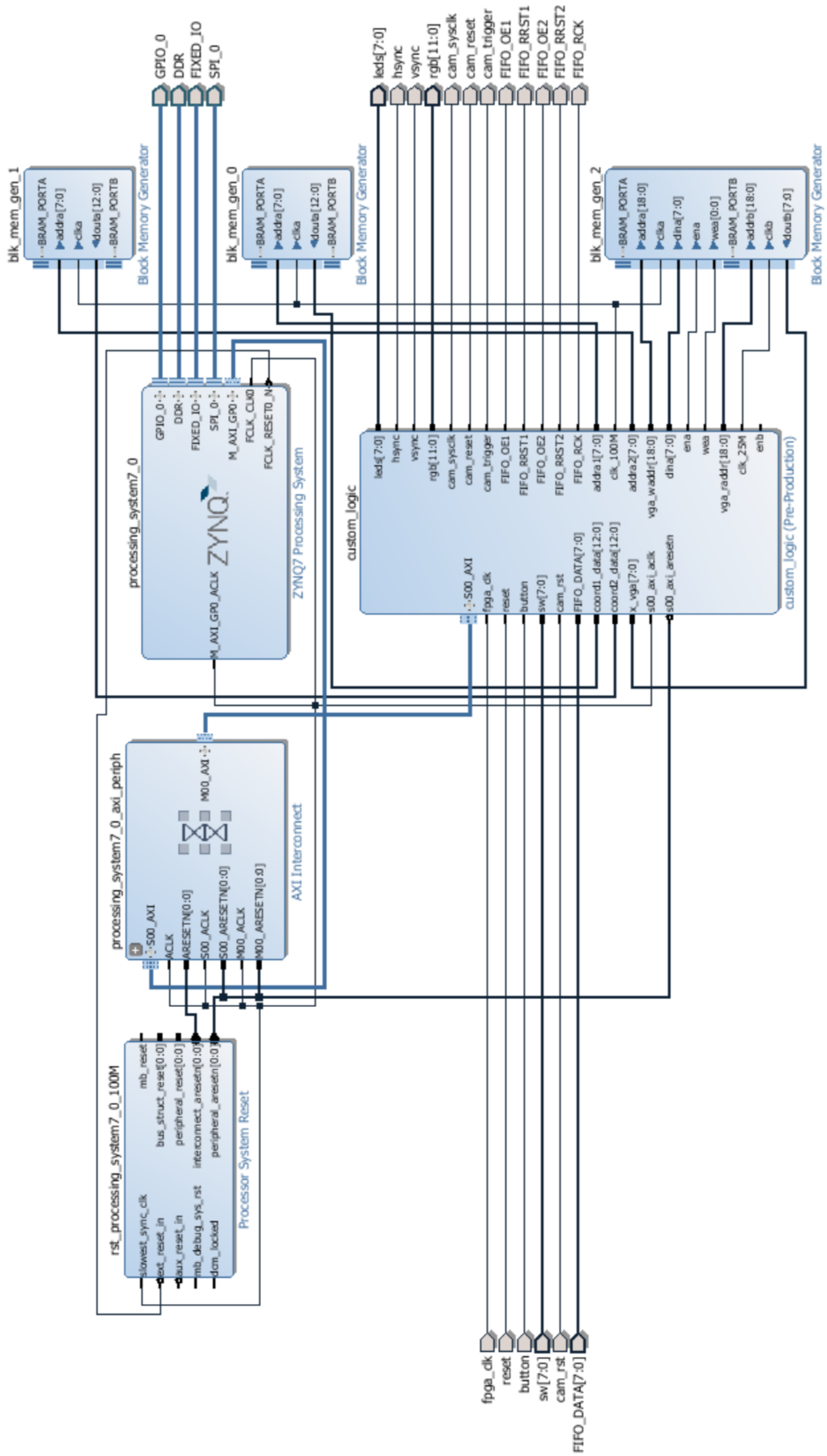


Figure 69: Implemented Design

The final hardware implementation of the project can be seen in Figure 70. This implementation supported several output modes based on the positions of the user switches, including a 3D disparity mode, camera image mode, rangefinder output mode, and combined 2D "floorplan" mode. Note that the VGA outputs of each mode were continuously updated.

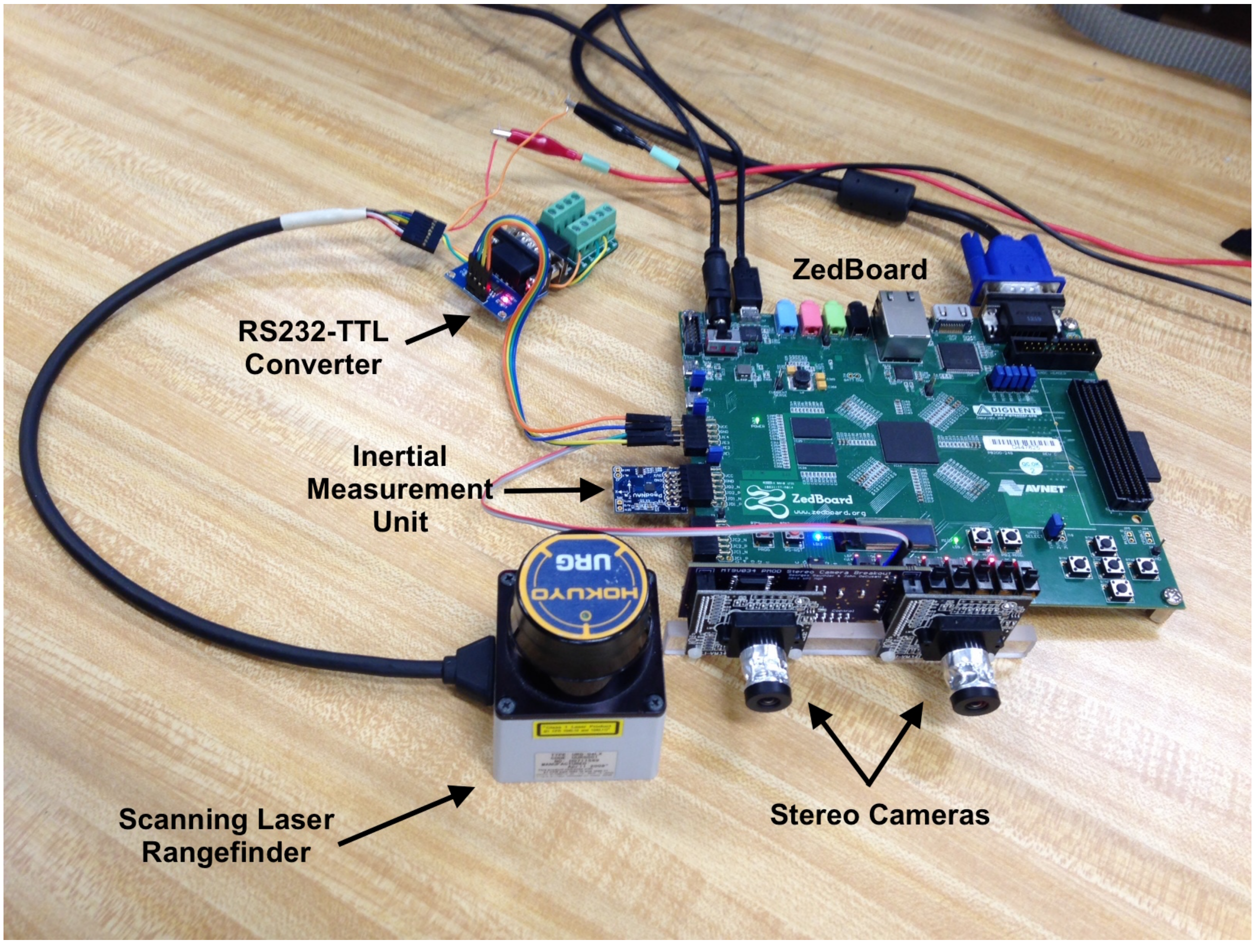


Figure 70: System Hardware

In order to create a fully integrated hardware-software interface that would allow for communication between the Zynq FPGA fabric and ARM processor, a customized IP core needed to be created. This issue was resolved by placing all programmable logic for the rangefinder, IMU, and camera interface into a user-generated IP core. named "custom logic", as shown in Figure 69. The IP core was connected directly to the stereo camera breakout board, and accepted rangefinder and IMU data from the programmable software via an AXI interface. This customized IP core also accepted user inputs through the ZedBoard's switches, and supported outputs to the ZedBoard's LEDs and VGA interface.

The 3D disparity output mode may be found in Figure 71a. In this mode, a windowed 384x288 pixel depth map was continuously updated to reflect the camera's current field of view. Figure 71b shows a modified version of this output consisting of depth information from a centrally-located horizontal line of pixels from this depth map that may then be correlated with data from the scanning laser rangefinder.

(a) Full Output

(b) Single Line Output

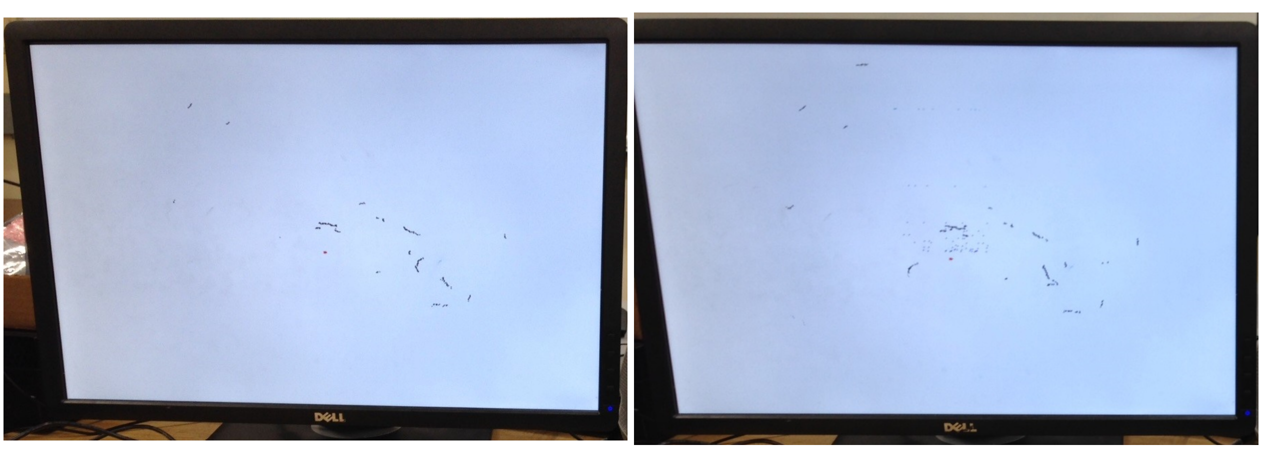
Figure 71: Disparity Output Modes

In order to aid in hardware debugging, an additional output mode was also included for showing the current camera images being used by the disparity algorithm, as shown in Figure 72. Note that the images captured were monochrome, and were arbitrarily mapped to VGA colors due to a lack of grayscale color space.

Figure 72: Raw Camera Data Mode

The normal rangefinder output mode is shown in Figure 73a. In this mode, objects found by the scanning laser rangefinder were displayed in black. The entire scan was also referenced to the device's central location, shown in red. All rangefinder data was pre-processed by the programmable software to include a compass offset from the IMU, with due north representing the center of the top of the VGA display.

A final output mode was also included to incorporate disparity data with the 2D "floor-plan" produced by the rangefinder. By combining data from both sensors, the stereo cameras were able to account for situations where the scanning laser rangefinder was out of range due to its limitations on viewing distance. This output mode is shown in Figure 73b.



(a) Rangefinder Output

(b) Combined Output

Figure 73: 2D "Floorplan" Output Modes

In this chapter, the functionality of each sensor was tested so that the design accurately reflects their behavior. Once each sensor's behavior was properly accounted for, their implementations were combined to create a functional SLAM sensor suite.

6 Conclusions

This project demonstrated a proof of concept SLAM sensor suite that could be used as a cost-effective replacement for the simple camera image sensors available on existing remote mapping robotic products. Through the use of a stereo camera pair, scanning laser rangefinder, inertial measurement unit, and Zynq7020 All Programmable SoC, an all-in-one sensor suite was developed that was capable of capturing 2D and 3D data on its surroundings. This sensor suite was unique in that it was able to process all of its data locally, without the need for additional processing power from the end user.

On the input stage of the sensor platform, the scanning laser rangefinder and IMU module were connected directly to the dual-core ARM Cortex A9 processor of the Zynq SoC, and were communicated with using low-level peripheral controls. Data collected from each sensor was passed to the FPGA fabric of the Zynq SoC, where a coordinate-axis transform was used to localize rangefinder data based on IMU readings and prepare it for output via VGA display. In addition, a custom made printed circuit board was used to connect two camera modules and image buffer ICs to the FPGA fabric of the Zynq SoC. After acquiring stereo camera image data, a Sum of Absolute Differences block matching algorithm was used to convert said data to depth measurements on a pixel by pixel basis. This depth information was then exported as-is in the form of a 3D depth map, or in slices combined with rangefinder and IMU data to form a 2D floorplan of the area being observed by the sensor suite. The outputs of this platform demonstrated that FPGA-based data processing was a viable replacement for the simple imaging sensors of existing remote mapping products.

At the time of submission, the rangefinder, camera, and IMU interfaces and processing stages described were fully implemented and tested in both hardware and various forms of simulation. This project was completed over a shortened timeline, and a larger portion of this time was dedicated to debugging hardware issues than originally expected. Due to a limited project budget of $250, a customized stereo camera interface PCB with attached frame buffer ICs was developed as a low-cost alternative to existing commercial products.

Several issues were encountered with the physical hardware of this interface, and this resulted in lost development time.

Issues were also encountered with finding a proper method of communicating with the scanning laser rangefinder, as the rangefinder and ZedBoard Zynq evaluation platform used two different logic voltage levels in their UART communications. After successfully resolving these issues by using a RS232-TTL logic-level converter, development began on integrating data from the inertial measurement unit into the rangefinder interface. After several weeks of unsuccessful communications and debugging with the original IMU intended for use in the project, it was determined that the unit was non-functional. A replacement unit relying on a different IMU sensor was eventually acquired in the final week of development, and little time was left for integrating IMU data into the sensor suite's overall implementation.

Although the issues mentioned resulted in large delays in development, the proof of concept SLAM sensor suite created through this project serves as an excellent platform for future development. The implications of the hardware issues faced in this project were quickly realized, and some recommendations for future work may be based off of original project goals that were modified as a result of time constraints.

6.1 Future Work

For this sensor suite to be utilized to its maximum potential, additional image processing algorithms should be implemented to allow for human recognition and object detection. In a first responder situation, human recognition could be used to provide potentially life-saving information about where people are located. With additional IMU data completely integrated, this sensor suite also has the capability to produce a sophisticated 2D map with both the sensor suite's displacement and rotational data, and can even be combined with the stereo cameras' depth and human detection information. This functionality would allow for a near-complete understanding of the environment around the device.

In order for the sensor suite's information to be useful to first responders, it needs to be accessible via wireless transmission. With wireless data transmission, not only is the system unrestricted by a physical connection, but the data can also be accessible by numerous devices at the same time. Since the sensor suite also processes its data locally, the information transmitted would be accessible to a wide range of low-power electronics without any major processing requirement.

To reduce overall size and cost of the system, the completed design could use an integrated printed circuit board containing a Zynq chip, an onboard IMU, and mounted stereo camera hardware. With the creation of a customized sensor board, this project could truly serve its purpose as a durable replacement sensor suite for a wide range of robotic platforms currently used for remote observation and mapping.

References

[1] Analog Devices, *ADIS16IMU1/PCBZ Breakout Board Wiki-Guide*. 2016. Available from: https://wiki.analog.com/resources/eval/user-guides/inertial-mems/imu/adis16imu1-pcb.

[2] Analog Devices, *Low Profile, Low Noise Six Degrees of Freedom Inertial Sensor*. Rev. C datasheet 2012. Available from: http://www.analog.com/media/en/technical-documentation/data-sheets/ADIS16375.pdf.

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

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

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

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

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

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

[9] Digilent. *Pmod NAV*. Available from: https://reference.digilentinc.com/reference/pmod/pmodnav/start.

[10] 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.

[11] Thomas Finley. *Two's Complement*. Available from: http://www.tfinley.net/notes/cps104/twoscomp.html. 2000.

[12] Hokuyo Automatic Co. *Scanning Laser Range Finder*. Available from: https://www.hokuyo-aut.jp/02sensor/07scanner/urg_04lx.html. 2009.

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

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

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

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

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

[18] 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.

[19] STMicroelectronics. *LPS25HD MEMS pressure sensor*. DocID027112 Rev 4 datasheet, 2016.

[20] STMicroelectronics. *LSM9DS1 iNEMO inertial module: 3D accelerometer, 3D gyroscope, 3D magnetometer*. DocID025715 Rev 3 datasheet, 2015.

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

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

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

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] Wikipedia. *USB On-The-Go*. 2017. Available from: https://en.wikipedia.org/wiki/USB_On-The-Go.

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

[30] Xilinx. *AR# 47511: Zynq-7000 AP SoC, SPI - In Master Mode on MIO, the SPI Controller Resets Itself when the SS0 Signal Asserts*. 2013. Available from: http://www.xilinx.com/support/answers/47511.html.

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

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

[33] Xilinx. *7 Series FPGAs Memory Resources User Guide*. 2016. Available from: https://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf.

Appendix

A 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`.

B Component Selection

Component	Part Number	Supplier	Cost
FPGA	ZedBoard	Borrowed	N/A
IMU	PmodNAV	Digilent	$45
Rangefinder	URG-04LX	Borrowed	N/A
RS232 to TTL Converter	MAX232CSE	uxcell	$7
RS232 Breakout Board	Swellder DB9	VIKINS Tech	$7
Stereo Cameras†	MT9V034	Mouser	$146

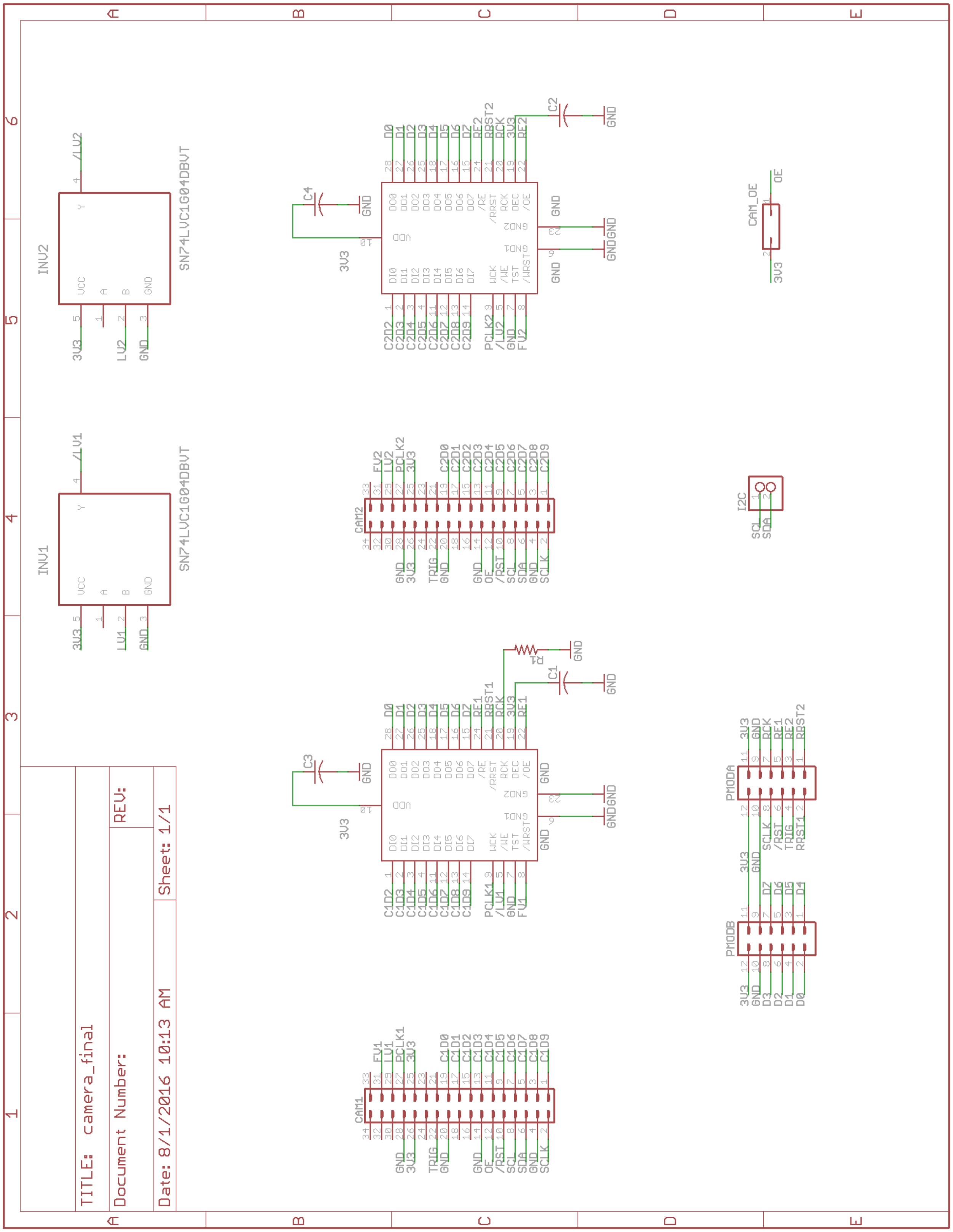
† Note that we originally planned to purchase a Flir Lepton thermal camera module and accompanying breakout board to support two stereo OV7670 camera modules. After experimenting with the OV7670 camera module on our FPGA board, we began to realize that these camera modules are highly limited due to their 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 field of view, 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 4.8.1.

C 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 [24]

D Stereo Camera Schematic



E Code

E.i MT9V034 and Al422b Test Code

Top Module:

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////
// Created by Georges Gauthier
// July 09 2016
// Test module for controlling the Leopardboard LI-VM34LP camera breakout
/////////////////////////////////////////////////////////////////////////
 module mt9v034_top(
                input sysclk, // 100MHz fpga clk
                input reset, // oddr2 reset
                input cam_rst, // button for camera RESET_BAR
                input trigger, // button for camera trigger
                input SW_cam_oe, // switch for camera output enable
                input cam_LV, // line valid in from camera
                output LOCKED, // oddr2 LOCKED led
                output cam_sysclk, // sysclk out to camera
                output cam_reset, // reset_bar out to camera
                output cam_trigger, // trigger out to camera
                output cam_oe, // output enable out to camera
                output i2c_ready, // LED indicator for i2c bus ready
                output [6:0] cathodes, // 7seg cathodes
                output [3:0] anodes, // 7seg anodes
                input MICRO_SW, // SW2, used to trigger a new FIFO dump over UART from the
                    mcs
                input mcs_reset, // Microblaze reset
                output MICRO_LED0, // LED used to indicate if mcs is reading from FIFO
                input [7:0] FIFO_DATA, // DO[7:0] from AL422b fifo
                output FIFO_WE, // Write enable to fifo (LV inverted)
                output FIFO_OE, // read enable to fifo (active low)
                output FIFO_RRST, // read reset to fifo (active low)
                output FIFO_RCK, // rck to fifo (1MHz)
                output UART_Tx // UART send pin from mcs
                );

wire clk_20Hz_unbuf, clk_20Hz;
wire clk_10kHz;
wire clk_1MHz, clk_1MHz_unbuf;
wire clk_24MHz;
wire clk_100MHz;

// 24MHz clock for driving MT9V034's SYSCLK
// 100mhz out for FIFO
// note you can't connect sysclk to a dcm and other things
dcm CLK_24MHz
   (
    .CLK_IN1(sysclk),
    .CLK_OUT1(clk_100MHz),
         .CLK_OUT2(clk_24MHz),
    .RESET(reset),
    .LOCKED(LOCKED)
         );

// further divide the dcm clock to other freqs
clk_div clks(
         .reset(reset), // synchronous reset
    .clk_24M(clk_24MHz), // 24MHz camera SCLK
         .clk_fifo(clk_1MHz_unbuf), // 1MHz FIFO RCK
    .clk_debounce(clk_20Hz_unbuf), // 20Hz clock pulse for debouncing stuff
         .anodes(clk_10kHz) // 10k 7Seg anode driver
    );
```

```

// clock buffer for 1MHz fifo rck
BUFG clk_1M (
      .O(clk_1MHz),
      .I(clk_1MHz_unbuf)
);
// clock buffer for 20Hz button debouncing
BUFG clk_20H (
      .O(clk_20Hz),
      .I(clk_20Hz_unbuf)
);

// forward the camera sysclk out using a dedicated clocking route
ODDR2 #(
      .DDR_ALIGNMENT("NONE"), // Sets output alignment to "NONE", "C0" or "C1"
      .INIT(1'b0),    // Sets initial state of the Q output to 1'b0 or 1'b1
      .SRTYPE("SYNC") // Specifies "SYNC" or "ASYNC" set/reset
   ) clkfwd0 (
      .Q(cam_sysclk),     // 1-bit DDR output data
      .C0(clk_24MHz),  // 1-bit clock input
      .C1(~clk_24MHz), // 1-bit clock input
      .CE(1'b1),      // 1-bit clock enable input
      .D0(1'b0), // 1-bit data input (associated with C0)
      .D1(1'b1), // 1-bit data input (associated with C1)
      .R(1'b0),   // 1-bit reset input
      .S(1'b0)   // 1-bit set input
   );

// forward the fifo read clk out using a dedicated clocking route
ODDR2 #(
      .DDR_ALIGNMENT("NONE"), // Sets output alignment to "NONE", "C0" or "C1"
      .INIT(1'b0),    // Sets initial state of the Q output to 1'b0 or 1'b1
      .SRTYPE("SYNC") // Specifies "SYNC" or "ASYNC" set/reset
   ) clkfwd1 (
      .Q(FIFO_RCK),     // 1-bit DDR output data
      .C0(clk_1MHz),  // 1-bit clock input
      .C1(~clk_1MHz), // 1-bit clock input
      .CE(1'b1),      // 1-bit clock enable input
      .D0(1'b0), // 1-bit data input (associated with C0)
      .D1(1'b1), // 1-bit data input (associated with C1)
      .R(1'b0),   // 1-bit reset input
      .S(1'b0)   // 1-bit set input
   );

// 7seg display controls
wire [15:0] displayVal;
seven_seg segs(
    .values(displayVal), // values to be written to the four seven segment LEDs
         .CLK(clk_24MHz), // 24MHz clock
         .en(clk_10kHz), // 10kHz counter enable used for setting the segment refresh rate
    .cathodes(cathodes),
    .anodes(anodes)
    );

// debounce trigger button input
debounce deb(
    .clk(clk_20Hz),
    .btn(trigger),
    .btn_val(cam_trigger)
    );

// debounce output enable switch
btnlatch sw_oe(
    .clk(clk_20Hz),
    .btn(SW_cam_oe),
    .btn_val(cam_oe)
    );

```

```verilog
// 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:

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

```
////////////////////////////////////////////////////////////////////////////
module fifo_read(
        input reset_pointer, // from microblaze, signal to assert fifo_rrst
        input get_data, // from microblaze (sent to trigger new read from FIFO to FPGA
            buffer)
        input [9:0] pixel_addr, // from microblaze, 0-751
        input [7:0] fifo_data, // 8 bit data in from fifo
        input fifo_rck, // 1MHz clock signal generated by FPGA
        output reg fifo_rrst, // fifo read reset (reset read addr pointer to 0)
        output reg fifo_oe, // fifo output enable (allow for addr pointer to increment)
        output reg buffer_ready, // to microblaze, signal that buffer is ready to read from
        output [7:0] pixel_value, // 8-bit value from internal buffer
        output [15:0] current_line // value to seven segment displays
   );

parameter [1:0] ready = 2'b00;
parameter [1:0] read = 2'b01;
parameter [1:0] done = 2'b10;
parameter [1:0] init = 2'b11;

reg [1:0] state = ready;
reg [1:0] prev_state, next_state = ready;

reg [7:0] pixel_line [0:751]; // implemented in BRAM
reg [9:0] pixel = 10'b00_0000_0000;
reg [15:0] num_lines = 16'h0000;

always @(posedge fifo_rck)
        state <= next_state;

always @(state,get_data,pixel)
        case(state)
                ready:
                        begin
                                if(get_data)
                                        next_state = init;
                                else
                                        next_state = ready;

                                prev_state = ready;
                        end
                init:
                        begin
                                next_state = read;
                                prev_state = init;
                        end
                read:
                        begin
                                if(pixel == 751)
                                        next_state = done;
                                else
                                        next_state = read;

                                prev_state = read;
                        end
                done:
                        begin
                                next_state = ready;
                                prev_state = done;
                        end
        endcase

always @(posedge fifo_rck)
begin
                if(reset_pointer)
                        begin
                        fifo_rrst <= 1'b0;
                        num_lines <= 16'h0000;
```

```
                        end
                else if(state==ready) // allow for MCS to read from pixel_line
                        begin
                        //pixel_value [7:0] <= pixel_line[pixel_addr];
                        fifo_rrst <= 1'b1; // make sure read addr doesn't get reset
                        end
                else if(state == init) // prepare to read new data from the AL422 into pixel_line
                        begin
                        pixel <= 10'b00_0000_000;
                        num_lines <= num_lines + 1'b1;
                        buffer_ready <= 1'b0;
                        fifo_oe <= 1'b0; // allow for read pointer to increment
                        end
                else if(state == read) // read data in from the AL422
                        begin
                        if(next_state == done)
                                fifo_oe <= 1'b1; // turn off read enable
                        if(prev_state != init) // one cycle delay between init and valid data
                        begin
                                pixel_line[pixel] <= fifo_data;
                                pixel <= pixel + 1'b1;
                        end
                        end
                else if(state == done)
                        begin
                        buffer_ready <= 1'b1;
                        end
end

// display number of lines written on 7seg display
assign current_line = (num_lines);
// allow for MCS to read stored pixel line at given addr if state==ready
assign pixel_value [7:0] = pixel_line[pixel_addr];

endmodule
```

MicroBlaze Code:

```
/*
 * 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;
```

```c

    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){
                print("\n\rReading from FIFO...\n\r");

                GPO1 |= (RRST); // reset FIFO position to 0th index
                GPO1 &=~ (GETDATA); // make sure we're not trying to read data
                XIOModule_DiscreteWrite(&gpo,1,GPO1);
                pixel_position = 0;
                GPO1 &=~(RRST);
                XIOModule_DiscreteWrite(&gpo,1,GPO1);
                GPO1 |= (GETDATA); // make sure we're not trying to read data
                XIOModule_DiscreteWrite(&gpo,1,GPO1);

                u32 pixelsRead = 0;

                while(row<480){
                    // make sure read_sw hasn't been turned off
                    GPI2 = XIOModule_DiscreteRead(&gpi,2);
                    if ((GPI2&SW_READ)==0) break;

                    u8 i=0;
                    // check to see if BUF_READY is good to go
                    GPI2 = XIOModule_DiscreteRead(&gpi,2);
                    // wait until it is
                    while((GPI2&BUF_READY)==0){
                        if(i==0){
                            i++;
                            //print("\n\t buffer not ready \n\r");
                        }
                        GPI2 = XIOModule_DiscreteRead(&gpi,2);
                    }
                    GPO1 &=~ (GETDATA); // make sure we're not trying to read data
                    XIOModule_DiscreteWrite(&gpo,1,GPO1);

                    while (pixel_position < 752){
                        // update pixel position for FPGA buffer
```

```
                        XIOModule_DiscreteWrite(&gpo,2,pixel_position);

                        // make sure read_sw hasn't been turned off
                        GPI2 = XIOModule_DiscreteRead(&gpi,2);
                        if ((GPI2&SW_READ)==0) break;

                        // read value at pixel position from FPGA buffer
                        data = XIOModule_DiscreteRead(&gpi,1);

                        //print the value
                        xil_printf("%d\n\r",data);

                        // increment to the next pixel position
                        pixel_position++;
                        pixelsRead ++;
                    }
                    // signal to the FPGA that we want more data!
                    GPO1 |= (GETDATA);
                    XIOModule_DiscreteWrite(&gpo,1,GPO1);
                    pixel_position = 0;
                    row++;
                    //xil_printf("Row: %d",row);

                }
                //xil_printf("%d Pixels Read by MCS",pixelsRead);
            } else {
                GPO1 &=~(GETDATA);
                GPO1 |= RRST;
                XIOModule_DiscreteWrite(&gpo,1,GPO1);
                print("\n\rReset for new sequence\n\r");
            }
        }

        prevState = swState; // update prev switch position
    }
    cleanup_platform();
    return 0;
}
```

Matlab Image Parser:

```
% 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)
```

E.ii Custom IP

Top Module:

```
`timescale 1 ns / 1 ps

	module custom_logic_v1_0 #
	(
		// Users to add parameters here

		// User parameters ends
		// Do not modify the parameters beyond this line


		// Parameters of Axi Slave Bus Interface S00_AXI
		parameter integer C_S00_AXI_DATA_WIDTH	= 32,
		parameter integer C_S00_AXI_ADDR_WIDTH	= 4
	)
	(
		// Users to add ports here

	//physical pins
	input wire fpga_clk,
	input wire reset,
	input wire button,
	input wire [7:0] sw,
	output wire [7:0] leds,
	output wire hsync,
	output wire vsync,
	output wire [11:0] rgb,

	// cameras
	input wire cam_rst, // button for camera RESET_BAR
	output wire cam_sysclk, // sysclk out to camera
	output wire cam_reset, // reset_bar out to camera
	output wire cam_trigger, // trigger out to camera
	input wire [7:0] FIFO_DATA, // DO[7:0] from AL422b fifo
	output wire FIFO_OE1, // read enable to fifo (active low)
	output wire FIFO_RRST1, // read reset to fifo (active low)
	output wire FIFO_OE2, // read enable to fifo (active low)
	output wire FIFO_RRST2, // read reset to fifo (active low)
	output wire FIFO_RCK, // rck to fifo (1MHz)

	//rangefinder BRAM
	output wire [7:0] addra1,
	input wire [12:0] coord1_data,
	output wire clk_100M,
	output wire [7:0] addra2,
	input wire [12:0] coord2_data,
	//output wire clk_100M2,

	//vga map BRAM
	output wire [18:0] vga_waddr,
	//output wire clk_100M3,
	output wire [7:0] dina,
	output wire ena,
	output wire wea,

	output wire [18:0] vga_raddr,
	output wire clk_25M,
	input wire [7:0] x_vga,
	output wire enb,

		// User ports ends
		// Do not modify the ports beyond this line
```

```

		// Ports of Axi Slave Bus Interface S00_AXI
		input wire  s00_axi_aclk,
		input wire  s00_axi_aresetn,
		input wire [C_S00_AXI_ADDR_WIDTH-1 : 0] s00_axi_awaddr,
		input wire [2 : 0] s00_axi_awprot,
		input wire  s00_axi_awvalid,
		output wire  s00_axi_awready,
		input wire [C_S00_AXI_DATA_WIDTH-1 : 0] s00_axi_wdata,
		input wire [(C_S00_AXI_DATA_WIDTH/8)-1 : 0] s00_axi_wstrb,
		input wire  s00_axi_wvalid,
		output wire  s00_axi_wready,
		output wire [1 : 0] s00_axi_bresp,
		output wire  s00_axi_bvalid,
		input wire  s00_axi_bready,
		input wire [C_S00_AXI_ADDR_WIDTH-1 : 0] s00_axi_araddr,
		input wire [2 : 0] s00_axi_arprot,
		input wire  s00_axi_arvalid,
		output wire  s00_axi_arready,
		output wire [C_S00_AXI_DATA_WIDTH-1 : 0] s00_axi_rdata,
		output wire [1 : 0] s00_axi_rresp,
		output wire  s00_axi_rvalid,
		input wire  s00_axi_rready
	);

    //processing system
    wire [27:0] data_enable_step;
    wire transmit;

// Instantiation of Axi Bus Interface S00_AXI
	custom_logic_v1_0_S00_AXI # (
		.C_S_AXI_DATA_WIDTH(C_S00_AXI_DATA_WIDTH),
		.C_S_AXI_ADDR_WIDTH(C_S00_AXI_ADDR_WIDTH)
	) custom_logic_v1_0_S00_AXI_inst (
	    .data_enable_step(data_enable_step),
	    .transmit(transmit),
		.S_AXI_ACLK(s00_axi_aclk),
		.S_AXI_ARESETN(s00_axi_aresetn),
		.S_AXI_AWADDR(s00_axi_awaddr),
		.S_AXI_AWPROT(s00_axi_awprot),
		.S_AXI_AWVALID(s00_axi_awvalid),
		.S_AXI_AWREADY(s00_axi_awready),
		.S_AXI_WDATA(s00_axi_wdata),
		.S_AXI_WSTRB(s00_axi_wstrb),
		.S_AXI_WVALID(s00_axi_wvalid),
		.S_AXI_WREADY(s00_axi_wready),
		.S_AXI_BRESP(s00_axi_bresp),
		.S_AXI_BVALID(s00_axi_bvalid),
		.S_AXI_BREADY(s00_axi_bready),
		.S_AXI_ARADDR(s00_axi_araddr),
		.S_AXI_ARPROT(s00_axi_arprot),
		.S_AXI_ARVALID(s00_axi_arvalid),
		.S_AXI_ARREADY(s00_axi_arready),
		.S_AXI_RDATA(s00_axi_rdata),
		.S_AXI_RRESP(s00_axi_rresp),
		.S_AXI_RVALID(s00_axi_rvalid),
		.S_AXI_RREADY(s00_axi_rready)
	);

	// Add user logic here

    mqp_top mqp_top
    (
        //physical pins
        .fpga_clk(fpga_clk),
        .reset(reset),
        .button(button),
        .sw(sw),
```

```
        .leds(leds),
        .hsync(hsync),
        .vsync(vsync),
        .rgb(rgb),

        // cameras
        .cam_rst(cam_rst), // button for camera RESET_BAR
        .cam_sysclk(cam_sysclk), // sysclk out to camera
        .cam_reset(cam_reset), // reset_bar out to camera
        .cam_trigger(cam_trigger), // trigger out to camera
        .FIFO_DATA(FIFO_DATA), // DO[7:0] from AL422b fifo
        .FIFO_OE1(FIFO_OE1), // read enable to fifo (active low)
        .FIFO_RRST1(FIFO_RRST1), // read reset to fifo (active low)
        .FIFO_OE2(FIFO_OE2), // read enable to fifo (active low)
        .FIFO_RRST2(FIFO_RRST2), // read reset to fifo (active low)
        .FIFO_RCK(FIFO_RCK), // rck to fifo (1MHz)

        //processing system
        .data_enable_step(data_enable_step),
        .transmit(transmit),

        //rangefinder BRAM controllers
        .addra1(addra1),
        .coord1_data(coord1_data),
        .clk_100M(clk_100M),
        .addra2(addra2),
        .coord2_data(coord2_data),
        //.clk_100M2(clk_100M2),

        //vga BRAM controller
        .vga_waddr(vga_waddr),
        //.clk_100M3(clk_100M3),
        .dina(dina),
        .ena(ena),
        .wea(wea),
        .vga_raddr(vga_raddr), // check size on this
        .clk_25M(clk_25M),
        .x_vga(x_vga),
        .enb(enb)
    );

      // User logic ends

        endmodule
```

Instantiated File:

```

`timescale 1 ns / 1 ps

        module custom_logic_v1_0_S00_AXI #
        (
                // Users to add parameters here

                // User parameters ends
                // Do not modify the parameters beyond this line

                // Width of S_AXI data bus
                parameter integer C_S_AXI_DATA_WIDTH    = 32,
                // Width of S_AXI address bus
                parameter integer C_S_AXI_ADDR_WIDTH    = 4
        )
        (
                // Users to add ports here
                output reg [27:0] data_enable_step,
        input wire transmit,
```

```
		// User ports ends
		// Do not modify the ports beyond this line

		// Global Clock Signal
		input wire  S_AXI_ACLK,
		// Global Reset Signal. This Signal is Active LOW
		input wire  S_AXI_ARESETN,
		// Write address (issued by master, acceped by Slave)
		input wire [C_S_AXI_ADDR_WIDTH-1 : 0] S_AXI_AWADDR,
		// Write channel Protection type. This signal indicates the
		// privilege and security level of the transaction, and whether
		// the transaction is a data access or an instruction access.
		input wire [2 : 0] S_AXI_AWPROT,
		// Write address valid. This signal indicates that the master signaling
		// valid write address and control information.
		input wire  S_AXI_AWVALID,
		// Write address ready. This signal indicates that the slave is ready
		// to accept an address and associated control signals.
		output wire  S_AXI_AWREADY,
		// Write data (issued by master, acceped by Slave)
		input wire [C_S_AXI_DATA_WIDTH-1 : 0] S_AXI_WDATA,
		// Write strobes. This signal indicates which byte lanes hold
		// valid data. There is one write strobe bit for each eight
		// bits of the write data bus.
		input wire [(C_S_AXI_DATA_WIDTH/8)-1 : 0] S_AXI_WSTRB,
		// Write valid. This signal indicates that valid write
		// data and strobes are available.
		input wire  S_AXI_WVALID,
		// Write ready. This signal indicates that the slave
		// can accept the write data.
		output wire  S_AXI_WREADY,
		// Write response. This signal indicates the status
		// of the write transaction.
		output wire [1 : 0] S_AXI_BRESP,
		// Write response valid. This signal indicates that the channel
		// is signaling a valid write response.
		output wire  S_AXI_BVALID,
		// Response ready. This signal indicates that the master
		// can accept a write response.
		input wire  S_AXI_BREADY,
		// Read address (issued by master, acceped by Slave)
		input wire [C_S_AXI_ADDR_WIDTH-1 : 0] S_AXI_ARADDR,
		// Protection type. This signal indicates the privilege
		// and security level of the transaction, and whether the
		// transaction is a data access or an instruction access.
		input wire [2 : 0] S_AXI_ARPROT,
		// Read address valid. This signal indicates that the channel
		// is signaling valid read address and control information.
		input wire  S_AXI_ARVALID,
		// Read address ready. This signal indicates that the slave is
		// ready to accept an address and associated control signals.
		output wire  S_AXI_ARREADY,
		// Read data (issued by slave)
		output wire [C_S_AXI_DATA_WIDTH-1 : 0] S_AXI_RDATA,
		// Read response. This signal indicates the status of the
		// read transfer.
		output wire [1 : 0] S_AXI_RRESP,
		// Read valid. This signal indicates that the channel is
		// signaling the required read data.
		output wire  S_AXI_RVALID,
		// Read ready. This signal indicates that the master can
		// accept the read data and response information.
		input wire  S_AXI_RREADY
	);

reg [31:0] extra_data_enable_step;

	// AXI4LITE signals
```

```verilog
	reg [C_S_AXI_ADDR_WIDTH-1 : 0] 	axi_awaddr;
	reg  	axi_awready;
	reg  	axi_wready;
	reg [1 : 0] 	axi_bresp;
	reg  	axi_bvalid;
	reg [C_S_AXI_ADDR_WIDTH-1 : 0] 	axi_araddr;
	reg  	axi_arready;
	reg [C_S_AXI_DATA_WIDTH-1 : 0] 	axi_rdata;
	reg [1 : 0] 	axi_rresp;
	reg  	axi_rvalid;

	// Example-specific design signals
	// local parameter for addressing 32 bit / 64 bit C_S_AXI_DATA_WIDTH
	// ADDR_LSB is used for addressing 32/64 bit registers/memories
	// ADDR_LSB = 2 for 32 bits (n downto 2)
	// ADDR_LSB = 3 for 64 bits (n downto 3)
	localparam integer ADDR_LSB = (C_S_AXI_DATA_WIDTH/32) + 1;
	localparam integer OPT_MEM_ADDR_BITS = 1;
	//----------------------------------------------
	//-- Signals for user logic register space example
	//------------------------------------------------
	//-- Number of Slave Registers 4
	reg [C_S_AXI_DATA_WIDTH-1:0]	slv_reg0;
	reg [C_S_AXI_DATA_WIDTH-1:0]	slv_reg1;
	reg [C_S_AXI_DATA_WIDTH-1:0]	slv_reg2;
	reg [C_S_AXI_DATA_WIDTH-1:0]	slv_reg3;
	wire	 slv_reg_rden;
	wire	 slv_reg_wren;
	reg [C_S_AXI_DATA_WIDTH-1:0]	 reg_data_out;
	integer	 byte_index;

	// I/O Connections assignments

	assign S_AXI_AWREADY	= axi_awready;
	assign S_AXI_WREADY	= axi_wready;
	assign S_AXI_BRESP	= axi_bresp;
	assign S_AXI_BVALID	= axi_bvalid;
	assign S_AXI_ARREADY	= axi_arready;
	assign S_AXI_RDATA	= axi_rdata;
	assign S_AXI_RRESP	= axi_rresp;
	assign S_AXI_RVALID	= axi_rvalid;
	// Implement axi_awready generation
	// axi_awready is asserted for one S_AXI_ACLK clock cycle when both
	// S_AXI_AWVALID and S_AXI_WVALID are asserted. axi_awready is
	// de-asserted when reset is low.

	always @( posedge S_AXI_ACLK )
	begin
	  if ( S_AXI_ARESETN == 1'b0 )
	    begin
	      axi_awready <= 1'b0;
	    end
	  else
	    begin
	      if (~axi_awready && S_AXI_AWVALID && S_AXI_WVALID)
	        begin
	          // slave is ready to accept write address when
	          // there is a valid write address and write data
	          // on the write address and data bus. This design
	          // expects no outstanding transactions.
	          axi_awready <= 1'b1;
	        end
	      else
	        begin
	          axi_awready <= 1'b0;
	        end
	    end
	end
```

```

	// Implement axi_awaddr latching
	// This process is used to latch the address when both
	// S_AXI_AWVALID and S_AXI_WVALID are valid.

	always @( posedge S_AXI_ACLK )
	begin
	  if ( S_AXI_ARESETN == 1'b0 )
	    begin
	      axi_awaddr <= 0;
	    end
	  else
	    begin
	      if (~axi_awready && S_AXI_AWVALID && S_AXI_WVALID)
	        begin
	          // Write Address latching
	          axi_awaddr <= S_AXI_AWADDR;
	        end
	    end
	end

	// Implement axi_wready generation
	// axi_wready is asserted for one S_AXI_ACLK clock cycle when both
	// S_AXI_AWVALID and S_AXI_WVALID are asserted. axi_wready is
	// de-asserted when reset is low.

	always @( posedge S_AXI_ACLK )
	begin
	  if ( S_AXI_ARESETN == 1'b0 )
	    begin
	      axi_wready <= 1'b0;
	    end
	  else
	    begin
	      if (~axi_wready && S_AXI_WVALID && S_AXI_AWVALID)
	        begin
	          // slave is ready to accept write data when
	          // there is a valid write address and write data
	          // on the write address and data bus. This design
	          // expects no outstanding transactions.
	          axi_wready <= 1'b1;
	        end
	      else
	        begin
	          axi_wready <= 1'b0;
	        end
	    end
	end

	// Implement memory mapped register select and write logic generation
	// The write data is accepted and written to memory mapped registers when
	// axi_awready, S_AXI_WVALID, axi_wready and S_AXI_WVALID are asserted. Write strobes are used to
	// select byte enables of slave registers while writing.
	// These registers are cleared when reset (active low) is applied.
	// Slave register write enable is asserted when valid address and data are available
	// and the slave is ready to accept the write address and write data.
	assign slv_reg_wren = axi_wready && S_AXI_WVALID && axi_awready && S_AXI_AWVALID;

	always @( posedge S_AXI_ACLK )
	begin
	  if ( S_AXI_ARESETN == 1'b0 )
	    begin
	      slv_reg0 <= 0;
	      slv_reg1 <= 0;
	      slv_reg2 <= 0;
	      slv_reg3 <= 0;
	    end
```

```
      else begin
        if (slv_reg_wren)
          begin
            case ( axi_awaddr[ADDR_LSB+OPT_MEM_ADDR_BITS:ADDR_LSB] )
              2'h0:
                for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1; byte_index = byte_index+1 )
                  if ( S_AXI_WSTRB[byte_index] == 1 ) begin
                    // Respective byte enables are asserted as per write strobes
                    // Slave register 0
                    slv_reg0[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];
                  end
              2'h1:
                for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1; byte_index = byte_index+1 )
                  if ( S_AXI_WSTRB[byte_index] == 1 ) begin
                    // Respective byte enables are asserted as per write strobes
                    // Slave register 1
                    extra_data_enable_step[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];
                  end
              2'h2:
                for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1; byte_index = byte_index+1 )
                  if ( S_AXI_WSTRB[byte_index] == 1 ) begin
                    // Respective byte enables are asserted as per write strobes
                    // Slave register 2
                    slv_reg2[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];
                  end
              2'h3:
                for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1; byte_index = byte_index+1 )
                  if ( S_AXI_WSTRB[byte_index] == 1 ) begin
                    // Respective byte enables are asserted as per write strobes
                    // Slave register 3
                    slv_reg3[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];
                  end
              default : begin
                          slv_reg0 <= slv_reg0;
                          slv_reg1 <= slv_reg1;
                          slv_reg2 <= slv_reg2;
                          slv_reg3 <= slv_reg3;
                        end
            endcase
          end
      end
    end

    // Implement write response logic generation
    // The write response and response valid signals are asserted by the slave
    // when axi_wready, S_AXI_WVALID, axi_wready and S_AXI_WVALID are asserted.
    // This marks the acceptance of address and indicates the status of
    // write transaction.

    always @( posedge S_AXI_ACLK )
    begin
      if ( S_AXI_ARESETN == 1'b0 )
        begin
          axi_bvalid  <= 0;
          axi_bresp   <= 2'b0;
        end
      else
        begin
          if (axi_awready && S_AXI_AWVALID && ~axi_bvalid && axi_wready && S_AXI_WVALID)
            begin
              // indicates a valid write response is available
              axi_bvalid <= 1'b1;
              axi_bresp  <= 2'b0; // 'OKAY' response
```

```
            end                   // work error responses in future
          else
            begin
              if (S_AXI_BREADY && axi_bvalid)
                //check if bready is asserted while bvalid is high)
                //(there is a possibility that bready is always asserted high)
                begin
                  axi_bvalid <= 1'b0;
                end
            end
        end
    end

    // Implement axi_arready generation
    // axi_arready is asserted for one S_AXI_ACLK clock cycle when
    // S_AXI_ARVALID is asserted. axi_awready is
    // de-asserted when reset (active low) is asserted.
    // The read address is also latched when S_AXI_ARVALID is
    // asserted. axi_araddr is reset to zero on reset assertion.

    always @( posedge S_AXI_ACLK )
    begin
      if ( S_AXI_ARESETN == 1'b0 )
        begin
          axi_arready <= 1'b0;
          axi_araddr  <= 32'b0;
        end
      else
        begin
          if (~axi_arready && S_AXI_ARVALID)
            begin
              // indicates that the slave has acceped the valid read address
              axi_arready <= 1'b1;
              // Read address latching
              axi_araddr  <= S_AXI_ARADDR;
            end
          else
            begin
              axi_arready <= 1'b0;
            end
        end
    end

    // Implement axi_arvalid generation
    // axi_rvalid is asserted for one S_AXI_ACLK clock cycle when both
    // S_AXI_ARVALID and axi_arready are asserted. The slave registers
    // data are available on the axi_rdata bus at this instance. The
    // assertion of axi_rvalid marks the validity of read data on the
    // bus and axi_rresp indicates the status of read transaction.axi_rvalid
    // is deasserted on reset (active low). axi_rresp and axi_rdata are
    // cleared to zero on reset (active low).
    always @( posedge S_AXI_ACLK )
    begin
      if ( S_AXI_ARESETN == 1'b0 )
        begin
          axi_rvalid <= 0;
          axi_rresp  <= 0;
        end
      else
        begin
          if (axi_arready && S_AXI_ARVALID && ~axi_rvalid)
            begin
              // Valid read data is available at the read data bus
              axi_rvalid <= 1'b1;
              axi_rresp  <= 2'b0; // 'OKAY' response
            end
          else if (axi_rvalid && S_AXI_RREADY)
            begin
```

```
                  // Read data is accepted by the master
                  axi_rvalid <= 1'b0;
                end
            end
        end

        // Implement memory mapped register select and read logic generation
        // Slave register read enable is asserted when valid address is available
        // and the slave is ready to accept the read address.
        assign slv_reg_rden = axi_arready & S_AXI_ARVALID & ~axi_rvalid;
        always @(*)
        begin
              // Address decoding for reading registers
              case ( axi_araddr[ADDR_LSB+OPT_MEM_ADDR_BITS:ADDR_LSB] )
                2'h0   : reg_data_out <= {31'b0, transmit};
                2'h1   : reg_data_out <= slv_reg1;
                2'h2   : reg_data_out <= slv_reg2;
                2'h3   : reg_data_out <= slv_reg3;
                default : reg_data_out <= 0;
              endcase
        end

        // Output register or memory read data
        always @( posedge S_AXI_ACLK )
        begin
          if ( S_AXI_ARESETN == 1'b0 )
            begin
              axi_rdata  <= 0;
            end
          else
            begin
              // When there is a valid read address (S_AXI_ARVALID) with
              // acceptance of read address by the slave (axi_arready),
              // output the read dada
              if (slv_reg_rden)
                begin
                  axi_rdata <= reg_data_out;     // register read data
                end
            end
        end

        // Add user logic here
    always @( posedge S_AXI_ACLK )
    begin
        data_enable_step = extra_data_enable_step[27:0];
    end

        // User logic ends

        endmodule
```

Rangefinder Data Processing:

```
`timescale 1ns / 1ps
// Rangefinder controller module
// Communicates with PS via AXI, BRAM lookup tables, GPIO, and VGA display controller
// Used to convert rangefinder data processed by the PS into a format that can be
// output for display and storage
module rangefinder(
    input clk, // 100MHz input clock
    input reset, // synchronous reset
    input button, // trigger new data capture (no longer used)
    input [8:0] device_x, // device x offset for display
    input [8:0] device_y, // device y offset for display
    output [7:0] leds, // LED indicators for debug
    input [15:0] data, // rangefinder data in (from PS/PL AXI)
```

```
    input enable, // PL data processing enable (from PS/PL AXI)
    input [10:0] step, // current step offset (0-768) from PS/PL AXI
    output reg [18:0] vga_waddr, // write address to VGA logic
    output reg transmit, // enable new data capture (to PS via AXI)
    output reg [7:0] dina, // VGA output buffer pixel data
    output reg wea, // VGA output buffer write enable

    output reg [7:0] addra1, // 0-45 degree ROM lookup table address
    input [12:0] coord1_data, // 0-45 degree ROM lookup table data

    output reg [7:0] addra2, // 45-90 degree ROM lookup table address
    input [12:0] coord2_data // 45-90 degree ROM lookup table address
    );

    reg [15:0] watchdog; // timeout to begin new data capture sequence
    reg [15:0] decoded; // stores 2 bytes of rangefinder data as (MSB -> LSB)-0x30
    reg [24:0] xmult, ymult; // rangefinder data point x,y location relative to (0,0)
    wire xneg, yneg; // flag for indicating if x,y data is positive or negative
    reg [9:0] xlocation; // rangefinder data point x location relative to device_x
    reg [8:0] ylocation; // rangefinder data point y location relative to device_y

    reg [2:0] offset_counter; // counter to wait for multiply operation in OFFSET state
    reg [1:0] address_counter; // counter to wait for multiply operation in ADDRESS state

    wire [18:0] vga_product; // current pixel row for output address

    wire [24:0] product1, product2; // x/y offset output from trig lookup / multiplier conversion

    // FSM states, next state logic
    parameter [2:0] IDLE = 0, CLEAR = 1, DECODE = 2, OFFSET = 3, SCALE = 4, ADDRESS = 5, WRITE = 6;
    reg [2:0] current_state, next_state;

    reg [10:0] row_count; // 0-480
    reg [10:0] col_count; // 0-640
    reg [8:0] step_count = 9'd0;
//------------------------------------------------------------
//data processing
    //sets flag to indicate negative horizontal value on circle
    assign xneg = (step >= 0 && step <= 128)  ? 1'b0 :
                  (step > 128 && step <= 384) ? 1'b0 :
                  (step > 384 && step <= 640) ? 1'b1 : //1'b1;
                  (step > 640 && step <= 768) ? 1'b1 :
                  (step > 768 && step <= 896) ? 1'b1 : 1'b0;

    //sets flag to indicate negative vertical value on circle
    assign yneg = (step >= 0 && step <= 128)  ? 1'b1 :
                  (step > 128 && step <= 384) ? 1'b0 :
                  (step > 384 && step <= 640) ? 1'b0 : //1'b1;
                  (step > 640 && step <= 768) ? 1'b1 :
                  (step > 768 && step <= 896) ? 1'b1 : 1'b1;

    // State machine overhead control
    always @ (posedge clk)
        if (reset)
            current_state <= IDLE;
        else
            current_state <= next_state;

    //next state logic
    always @ (current_state, offset_counter, address_counter, enable, row_count, col_count, button, step)
    begin
        case (current_state)
            // stays in IDLE until a new enable pulse
            IDLE:
                if(enable)
```

```
                    next_state = DECODE;
                else if (button) // was (button || step == 768)
                    next_state = CLEAR;
                else
                    next_state = IDLE;

            CLEAR: // clear pixel data from previous run
                if(row_count == 479 && col_count == 639)
                    next_state = IDLE;
                else
                    next_state = CLEAR;
            // stays in DECODE for one clock cycle
            DECODE:
                next_state = OFFSET;

            // stays in OFFSET for a multiply operation
            OFFSET:
                if(offset_counter == 4)
                    next_state = SCALE;
                else
                    next_state = OFFSET;

            // stays in SCALE for one clock cycle
            SCALE:
                next_state = ADDRESS;

            // stays in ADDRESS for a multiply operation
            ADDRESS:
                if(address_counter == 3)
                    next_state = WRITE;
                else
                    next_state = ADDRESS;

            // stays in ADDRESS for one clock cycle
            WRITE:
                next_state = IDLE;

            default:
                next_state = IDLE;
        endcase
    end

    //state machine
    always @ (posedge clk)
    begin
        //resets registers, waits for data to process
        if(current_state == IDLE)
        begin
            wea <= 1'b0;
            dina <= 8'h00;
            row_count <= 11'd0;
            col_count <= 11'd0;
            if(step_count == 9'd767)
                step_count <= 9'd0;
        end

        // clear previous set of data from BRAM
        else if(current_state == CLEAR)
        begin
            vga_waddr <= (640*row_count) + col_count;
            wea <= 1'b1;
            dina <= 8'h00;
            if(col_count < 639)
                col_count <= col_count + 1'b1;
            else if(col_count == 639 && row_count < 479)
                begin
                col_count <= 11'd0;
                row_count <= row_count + 1'b1;
```

```
                end

        end
        //calculates adresses for trig LUTs
        //decodes data by substracting 0x30 from both halves of data point
        else if(current_state == DECODE)
        begin
            step_count <= step_count + 1'b1;

            wea <= 1'b0;

            addra1 <= (step >= 0 && step <= 128)  ? 128-step :  //index 128 to 0  - Q1 - hz
                      (step > 128 && step <= 256) ? step-128 :  //index 1 to 128  - Q2 - hz
                      (step > 256 && step <= 384) ? 384-step :  //index 127 to 0  - Q2 - vr
                      (step > 384 && step <= 512) ? step-384 :  //index 1 to 128  - Q3 - vr
                      (step > 512 && step <= 640) ? 640-step :  //index 127 to 0  - Q3 - hz
                      (step > 640 && step <= 768) ? step-640 :  //index 1 to 128  - Q4 - hz
                      (step > 768 && step <= 896) ? 896-step :  //index 127 to 0  - Q4 - vr
                      //(step > 896 && step <= 1023) ? step - 896 :
                                                    step-896;   //index 0 to 128  - Q1 - vr

            addra2 <= (step >= 0 && step <= 128)  ? step :      //index 128 to 256 - Q1 - vr
                      (step > 128 && step <= 256) ? 256-step :  //index 255 to 128 - Q2 - vr
                      (step > 256 && step <= 384) ? step-256 :  //index 129 to 256 - Q2 - hz
                      (step > 384 && step <= 512) ? 512-step :  //index 255 to 128 - Q3 - hz
                      (step > 512 && step <= 640) ? step-512 :  //index 129 to 256 - Q3 - vr
                      (step > 640 && step <= 768) ? 768-step :
                      (step > 768 && step <= 896) ? step-768 :
                      //(step > 896 && step <= 1023) ? 1023 - step :
                                                    1023-step;   //index 255 to 128 - Q4 -
                                                        vr


            decoded <= {data[7:0]-8'h30, data[15:8]-8'h30};
        end

        //drops upper bit of each data point
        //calculates horizontal and vertical distance for each data point
        else if(current_state == OFFSET)
        begin
            if(offset_counter == 4)
            begin
                if((step > 256 && step <= 512) || (step > 768 && step <= 1023))
                begin
                    xmult <= product2;
                    ymult <= product1;
                end

                else
                begin
                    xmult <= product1;
                    ymult <= product2;
                end
            end

            offset_counter <= offset_counter + 1'b1;
        end

        //scales data and localizes to device
        else if(current_state == SCALE)
        begin
            offset_counter <= 3'b0;

            if(xneg)
                xlocation <= device_x - xmult[23:16];
            else
                xlocation <= device_x + xmult[23:16];

```

```
            if(yneg)
                ylocation <= device_y + ymult[23:16];
            else
                ylocation <= device_y - ymult[23:16];
        end

        //calculates address for VGA BRAM
        else if(current_state == ADDRESS)
        begin
            if(address_counter == 3)
                vga_waddr <= vga_product + xlocation;

            address_counter <= address_counter + 1'b1;
        end

        //writes to BRAM
        else if(current_state == WRITE)
        begin
            address_counter <= 2'b00;
            dina <= 8'hFF;
            wea <= 1'b1;
        end
    end

    // convert data point, trig lookup to x or y offset
    mult_gen_0 trig_mult_1
    (
        .CLK(clk),
        .A({decoded[13:8], decoded[5:0]}),
        .B(coord1_data),
        .P(product1)
    );
    // convert data point, trig lookup to x or y offset
    mult_gen_0 trig_mult_2
    (
        .CLK(clk),
        .A({decoded[13:8], decoded[5:0]}),
        .B(coord2_data),
        .P(product2)
    );

    // calculates 640*row_count portion of VGA write address
    mult_gen_2 vga_multiplier
    (
        .CLK(clk),
        .A(ylocation),
        .P(vga_product)
    );

    //watchdog counter watches for missed data transfers
    always @(posedge clk, posedge reset)
    begin
        if(reset || transmit)
            watchdog <= 0;
        else if (current_state == IDLE)
            watchdog <= watchdog + 1'b1;
        else
            watchdog <= watchdog;
    end

    // trigger a new rangefinder data aquisition sequence when the entire screen has been
        cleared
    always @ (posedge clk)
    begin
        if((row_count == 479 && col_count == 639) || step_count == 9'd767)
            transmit <= 1'b1;
        else
            transmit <= 1'b0;
```

```
    end

    // write current_state, next_state, and I/O to LEDs for debug
    assign leds[7:0] = {next_state[2:0],current_state[2:0],enable,button};

endmodule
```

VGA Controller Module by Digilent:

```
------------------------------------------------------------------------
-- vga_controller_640_60.vhd
------------------------------------------------------------------------
-- Author : Ulrich Zoltan
--          Copyright 2006 Digilent, Inc.
------------------------------------------------------------------------
-- Software version : Xilinx ISE 7.1.04i
--                    WebPack
-- Device             : 3s200ft256-4
------------------------------------------------------------------------
-- This file contains the logic to generate the synchronization signals,
-- horizontal and vertical pixel counter and video disable signal
-- for the 640x480@60Hz resolution.
------------------------------------------------------------------------
--  Behavioral description
------------------------------------------------------------------------
-- Please read the following article on the web regarding the
-- vga video timings:
-- http://www.epanorama.net/documents/pc/vga_timing.html

-- This module generates the video synch pulses for the monitor to
-- enter 640x480@60Hz resolution state. It also provides horizontal
-- and vertical counters for the currently displayed pixel and a blank
-- signal that is active when the pixel is not inside the visible screen
-- and the color outputs should be reset to 0.

-- timing diagram for the horizontal synch signal (HS)
-- 0                         648    744           800 (pixels)
-- -------------------------|______|-----------------
-- timing diagram for the vertical synch signal (VS)
-- 0                                  482    484  525 (lines)
-- -----------------------------------|______|-------

-- The blank signal is delayed one pixel clock period (40ns) from where
-- the pixel leaves the visible screen, according to the counters, to
-- account for the pixel pipeline delay. This delay happens because
-- it takes time from when the counters indicate current pixel should
-- be displayed to when the color data actually arrives at the monitor
-- pins (memory read delays, synchronization delays).
------------------------------------------------------------------------
--  Port definitions
------------------------------------------------------------------------
-- rst                - global reset signal
-- pixel_clk          - input pin, from dcm_25MHz
--                    - the clock signal generated by a DCM that has
--                    - a frequency of 25MHz.
-- HS                 - output pin, to monitor
--                    - horizontal synch pulse
-- VS                 - output pin, to monitor
--                    - vertical synch pulse
-- hcount             - output pin, 11 bits, to clients
--                    - horizontal count of the currently displayed
--                    - pixel (even if not in visible area)
-- vcount             - output pin, 11 bits, to clients
--                    - vertical count of the currently active video
--                    - line (even if not in visible area)
-- blank              - output pin, to clients
```

```vhdl
--                    - active when pixel is not in visible area.
-----------------------------------------------------------------------------
-- Revision History:
-- 09/18/2006(UlrichZ): created
-----------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

-- simulation library
library UNISIM;
use UNISIM.VComponents.all;

-- the vga_controller_640_60 entity declaration
-- read above for behavioral description and port definitions.
entity vga_controller_640_60 is
port(
   rst         : in std_logic;
   pixel_clk   : in std_logic;

   HS          : out std_logic;
   VS          : out std_logic;
   hcount      : out std_logic_vector(10 downto 0);
   vcount      : out std_logic_vector(10 downto 0);
   blank       : out std_logic
);
end vga_controller_640_60;

architecture Behavioral of vga_controller_640_60 is

-----------------------------------------------------------------------------
-- CONSTANTS
-----------------------------------------------------------------------------

-- maximum value for the horizontal pixel counter
constant HMAX  : std_logic_vector(10 downto 0) := "01100100000"; -- 800
-- maximum value for the vertical pixel counter
constant VMAX  : std_logic_vector(10 downto 0) := "01000001101"; -- 525
-- total number of visible columns
constant HLINES: std_logic_vector(10 downto 0) := "01010000000"; -- 640
-- value for the horizontal counter where front porch ends
constant HFP   : std_logic_vector(10 downto 0) := "01010001000"; -- 648
-- value for the horizontal counter where the synch pulse ends
constant HSP   : std_logic_vector(10 downto 0) := "01011101000"; -- 744
-- total number of visible lines
constant VLINES: std_logic_vector(10 downto 0) := "00111100000"; -- 480
-- value for the vertical counter where the front porch ends
constant VFP   : std_logic_vector(10 downto 0) := "00111100010"; -- 482
-- value for the vertical counter where the synch pulse ends
constant VSP   : std_logic_vector(10 downto 0) := "00111100100"; -- 484
-- polarity of the horizontal and vertical synch pulse
-- only one polarity used, because for this resolution they coincide.
constant SPP   : std_logic := '0';

-----------------------------------------------------------------------------
-- SIGNALS
-----------------------------------------------------------------------------

-- horizontal and vertical counters
signal hcounter : std_logic_vector(10 downto 0) := (others => '0');
signal vcounter : std_logic_vector(10 downto 0) := (others => '0');

-- active when inside visible screen area.
signal video_enable: std_logic;

begin
```

```vhdl

    -- output horizontal and vertical counters
    hcount <= hcounter;
    vcount <= vcounter;

    -- blank is active when outside screen visible area
    -- color output should be blacked (put on 0) when blank in active
    -- blank is delayed one pixel clock period from the video_enable
    -- signal to account for the pixel pipeline delay.
    blank <= not video_enable when rising_edge(pixel_clk);

    -- increment horizontal counter at pixel_clk rate
    -- until HMAX is reached, then reset and keep counting
    h_count: process(pixel_clk)
    begin
       if(rising_edge(pixel_clk)) then
          if(rst = '1') then
             hcounter <= (others => '0');
          elsif(hcounter = HMAX) then
             hcounter <= (others => '0');
          else
             hcounter <= hcounter + 1;
          end if;
       end if;
    end process h_count;

    -- increment vertical counter when one line is finished
    -- (horizontal counter reached HMAX)
    -- until VMAX is reached, then reset and keep counting
    v_count: process(pixel_clk)
    begin
       if(rising_edge(pixel_clk)) then
          if(rst = '1') then
             vcounter <= (others => '0');
          elsif(hcounter = HMAX) then
             if(vcounter = VMAX) then
                vcounter <= (others => '0');
             else
                vcounter <= vcounter + 1;
             end if;
          end if;
       end if;
    end process v_count;

    -- generate horizontal synch pulse
    -- when horizontal counter is between where the
    -- front porch ends and the synch pulse ends.
    -- The HS is active (with polarity SPP) for a total of 96 pixels.
    do_hs: process(pixel_clk)
    begin
       if(rising_edge(pixel_clk)) then
          if(hcounter >= HFP and hcounter < HSP) then
             HS <= SPP;
          else
             HS <= not SPP;
          end if;
       end if;
    end process do_hs;

    -- generate vertical synch pulse
    -- when vertical counter is between where the
    -- front porch ends and the synch pulse ends.
    -- The VS is active (with polarity SPP) for a total of 2 video lines
    -- = 2*HMAX = 1600 pixels.
    do_vs: process(pixel_clk)
    begin
       if(rising_edge(pixel_clk)) then
          if(vcounter >= VFP and vcounter < VSP) then
```

```
            VS <= SPP;
         else
            VS <= not SPP;
         end if;
      end if;
   end process do_vs;

   -- enable video output when pixel is in visible area
   video_enable <= '1' when (hcounter < HLINES and vcounter < VLINES) else '0';

end Behavioral;
```

E.iii 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
```

E.iv Programmable Software

Programmable Software for the Zynq7 Processing System:

```
I#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"
#include "xil_io.h"
#include "xbasic_types.h"
#include "xparameters.h"
#include "xiicps.h"
#include "xgpiops.h"
#include "xspips.h"     /* SPI device driver */
#include <math.h>

//I2C config params
#define IIC_DEVICE_ID XPAR_XIICPS_0_DEVICE_ID
#define IIC_SCLK_RATE 100000
#define PAGE_SIZE 16
// camera I2C addresses
#define CAM1_ADDR 0x48
#define CAM2_ADDR 0x58
// camera I2C write addresses
#define REG_LOCK 0xFE
#define CAMERA_CTL 0x07
// camera I2C write values
#define MANUAL_TRIG 0x0198
#define LOCKED 0xDEAD
#define UNLOCKED 0xBEEF
// I2C address size
typedef u8 AddressType;

// Zynq SPI device ID
#define SPI_DEVICE_ID       XPAR_XSPIPS_0_DEVICE_ID
// SPI addresses to read to / write from
#define READ 0x80 /*OR*/
#define WRITE 0x7F /*AND*/
#define CTL1 0x20
#define CTL2 0x21
#define CTL3 0x22
#define STATUS_ADDRESS 0x27
#define DATA_ADDRESS 0x28
#define OFFSET_ADDRESS_XLOW 0x05
#define OFFSET_ADDRESS_XHIGH 0x06
#define OFFSET_ADDRESS_YLOW 0x07
#define OFFSET_ADDRESS_YHIGH 0x08
#define CASCADE 0x60

#define PI 3.14159265
#define DEGREES_PER_STEP 0.3515625  // 360 degrees/1024 steps = 0.3515625
#define MAGNETOMETER_SENSITIVITY 0.00014    // +/- 4 gauss

// create a new SPI instance
static XSpiPs SpiInstance;
// create a new GPIO instance
static XGpioPs Gpio;
static XIicPs IicInstance;      /* The instance of the IIC device. */

int xOffset, yOffset;
int xData, yData;

typedef enum
{
    WAIT = 0,
    TX_COMMAND = 1,
    RX_DATA = 2
```

```
}UART_STATE;

// function prototype for camera initialization fxn
void init_cams();

void delay(int cycles);
void IMU_init();
int getIMUdata();
double getCompassHeading();
double getStepOffset(double compassHeading);

int main()
{
    init_platform();

    char echo[11];
    char status[2];
    char databuf[65][24];
    char databufbuf[3];
    char linefeed[4];

    echo[10] = '\0';
    databufbuf[2] = '\0';

    Xuint32 *baseaddr_p = (Xuint32 *)XPAR_CUSTOM_LOGIC_S00_AXI_BASEADDR;

    u32 data_enable_step;
    u32 write;
    u32 stepbuf;

    char *echotestnl = "G00076801\n";
    char *echotestlf = "G00076801\r";

    UART_STATE STATE = WAIT;

    // setup LED output
    XGpioPs_Config * ConfigPtr = XGpioPs_LookupConfig(XPAR_PS7_GPIO_0_DEVICE_ID);
    XGpioPs_CfgInitialize(&Gpio, ConfigPtr, ConfigPtr->BaseAddr);
    XGpioPs_SetDirectionPin(&Gpio, 7, 1);
    // initialize cameras using I2C
    init_cams();

    int Status;
    // initialize SPI
    XSpiPs_Config *SpiConfig;
    SpiConfig = XSpiPs_LookupConfig(SPI_DEVICE_ID);
    // initialize the spi hardware with the device config
    Status = XSpiPs_CfgInitialize(&SpiInstance, SpiConfig, SpiConfig->BaseAddress);
    // Set the Spi device as a master
    XSpiPs_SetOptions(&SpiInstance, XSPIPS_MASTER_OPTION | XSPIPS_CLK_PHASE_1_OPTION |
        XSPIPS_FORCE_SSELECT_OPTION);
    // Set the SPI clock prescaler
    XSpiPs_SetClkPrescaler(&SpiInstance, XSPIPS_CLK_PRESCALE_256);
    // initialize the imu
    IMU_init();

    int deviceStepOffset = 0;


    // Continuously run rangefinder state machine after initialization is complete
    while(1)
    {
        switch(STATE)
        {
            // Wait until the PL indicates that it wants new rangefinder data
            case WAIT:
            {
                //blocks until the flag is set (from PL) to initiate data transfer
```

```
            while(Xil_In32(baseaddr_p) == 0);
            while(getIMUdata()==0);
            deviceStepOffset = (int) round(getStepOffset(getCompassHeading()));

            STATE = TX_COMMAND;
            break;
        }
        // Call to the rangedfinder for a new round of data acquisition
        case TX_COMMAND:
        {
            printf("G00076801\n"); // data acquisition command
            STATE = RX_DATA;
            break;
        }
        // Process incoming data from the rangefinder byte by byte
        case RX_DATA:
        {
            int rx_index = 0;   // indexes rows
            int rx_line = 0;    // indexes columns - counts which data line is being received

            // receives echo
            for(rx_index = 0; rx_index < 10; rx_index++)
            {
                echo[rx_index] = inbyte();  // blocking - inbyte is polled
            }

            // makes sure the echo command is successful
            // if not, it repeats the tx_command state
            if(strcmp(echo, echotestnl) != 0 && strcmp(echo, echotestlf) != 0)
            {
                STATE = WAIT;
                break;
            }

            // receives status
            for(rx_index = 0; rx_index < 2; rx_index++)
            {
                status[rx_index] = inbyte();    // blocking - inbyte is polled
            }

            int iteration = 0;

            // receives 24 data blocks
            stepbuf = deviceStepOffset; //deviceStepOffset;
            for(rx_line = 0; rx_line < 24; rx_line++)
            {
                iteration = 0;
                // receives 65 bytes per block
                for(rx_index = 0; rx_index < 65; rx_index++)
                {
                    iteration++;
                    databuf[rx_line][rx_index] = inbyte();  // blocking - inbyte is polled
                    write = 0;
                    databufbuf[1-(iteration%2)] = databuf[rx_line][rx_index];

                    // process data two chars at a time
                    if(iteration%2 == 0)
                    {
                        //'enables' data in memory when the data is not an error code
                        if(!(strcmp(databufbuf[0], '0') == 0 && strcmp(databufbuf[1], 'C') <= 0))
                            write = 1;
                        stepbuf++;
                    }

                    //sends information to the programmable logic for each data point (2
```

```
                             chars)
                        //in the order of {data, enable, step}
                        //data_enable_step = (400 << 20) | (400 << 12) | (write << 11) |
                            stepbuf;
                        data_enable_step = (databufbuf[1] << 20) | (databufbuf[0] << 12) | (
                            write << 11) | stepbuf;
                        *(baseaddr_p+1) = data_enable_step;

                    }
                }

                iteration = 0;
                // account for 4 chars of data after step data has been sent
                for(rx_index = 0; rx_index < 4; rx_index++)
                {
                    linefeed[rx_index] = inbyte();  // blocking - inbyte is polled
                }

                echo[0] = '\0';

                STATE = WAIT;
                break;
            }
        }
    }
    cleanup_platform();
    return 0;
}

void init_cams()
{
    // number of bytes to be written from write buffer
    u8 ByteCount = 4;
    // Create and fill write buffer with ByteCount bytes of data
    u8 WriteBuffer[sizeof(AddressType) + PAGE_SIZE];
    WriteBuffer[0] = (u8) (CAM1_ADDR); // camera 1 I2C address
    WriteBuffer[1] = (u8) (CAMERA_CTL); // camera control register
    WriteBuffer[2] = (u8) (MANUAL_TRIG >> 8); // 16 bit write value (upper byte)
    WriteBuffer[3] = (u8) (MANUAL_TRIG); // 16 bit write value (lower byte)

    XIicPs_Config *ConfigPtr;
    int Status;

    // Look up Zynq-specific IIC device configuration
    ConfigPtr = XIicPs_LookupConfig(IIC_DEVICE_ID);
    // Initialize said device-specific iic configuration so the driver is ready for use
    Status = XIicPs_CfgInitialize(&IicInstance, ConfigPtr, ConfigPtr->BaseAddress);
    // Run a self test to make sure the driver works (will return XST_SUCCESS if working)
    Status = XIicPs_SelfTest(&IicInstance);
    // Set iic SCLK rate (bus now ready for use)
    XIicPs_SetSClk(&IicInstance, IIC_SCLK_RATE);

    // Initialize Cam1
    XIicPs_MasterSendPolled(&IicInstance, WriteBuffer, ByteCount,CAM1_ADDR);
    while(XIicPs_BusIsBusy(&IicInstance));

    // Initialize Cam2
    WriteBuffer[0] = (u8) (CAM2_ADDR); // change write address to cam2
    XIicPs_MasterSendPolled(&IicInstance, WriteBuffer, ByteCount,CAM2_ADDR);
    while(XIicPs_BusIsBusy(&IicInstance));

    // Write LED to indicate finished initialization sequence
    XGpioPs_WritePin(&Gpio, 7, 0x01);
}

void delay(int cycles)
{
    int i = 0;
```

```c
    while(i<cycles)
        i++;
    return;
}

void IMU_init()
{
    u8 DataBuffer[2]; // addr + 8 bits write val
    // high power mode
    DataBuffer[0] = (u8) CTL1 & WRITE;
    DataBuffer[1] = (u8) 0x7C;
    // ONLY SET SLAVE SELECT ON THE 1ST TRANSFER
    XSpiPs_SetSlaveSelect(&SpiInstance,0x01);
    XSpiPs_PolledTransfer(&SpiInstance, DataBuffer, NULL, 2);
    delay(500);

    DataBuffer[0] = (u8) CTL2 & WRITE;
    DataBuffer[1] = (u8) 0x00;
    XSpiPs_PolledTransfer(&SpiInstance, DataBuffer, NULL, 2);
    delay(500);

    // turn on the device
    DataBuffer[0] = (u8) CTL3 & WRITE;
    DataBuffer[1] = (u8) 0x80;
    XSpiPs_PolledTransfer(&SpiInstance, DataBuffer, NULL, 2);
    delay(250);

    return;
}

int getIMUdata()
{

    int newDataReady = 0;

    // check status address
    u8 DataBuffer[2]; // addr + 8 bits read val
    DataBuffer[0] = (u8) STATUS_ADDRESS | READ;
    DataBuffer[1] = (u8) 0x00;
    XSpiPs_PolledTransfer(&SpiInstance, DataBuffer, &DataBuffer[0], 2);

    // if XY data's available, indicate so and grab it
    if ((DataBuffer[1] & 0x03) == 0x03) {

        // grabs x and y magnetometer environmental offset
        u8 XYoffset[5];
        XYoffset[0] = (u8) OFFSET_ADDRESS_XLOW | CASCADE | READ;
        XYoffset[1] = (u8) 0x00;
        XYoffset[2] = (u8) 0x00;
        XYoffset[3] = (u8) 0x00;
        XYoffset[4] = (u8) 0x00;
        XSpiPs_PolledTransfer(&SpiInstance, XYoffset, &XYoffset[0], 5);
        xOffset = (XYoffset[2]<<8) | XYoffset[1];
        yOffset = (XYoffset[4]<<8) | XYoffset[3];

        // grabs x and y magnetometer data
        u8 XYdata[5];
        XYdata[0] = (u8) DATA_ADDRESS | CASCADE | READ;
        XYdata[1] = (u8) 0x00;
        XYdata[2] = (u8) 0x00;
        XYdata[3] = (u8) 0x00;
        XYdata[4] = (u8) 0x00;
        XSpiPs_PolledTransfer(&SpiInstance, XYdata, &XYdata[0], 5);
        xData = (XYdata[2]<<8) | XYdata[1];
        yData = (XYdata[4]<<8) | XYdata[3];


        if(XYdata[2] >= 0x80)
```

```
            xData = (xData - 65535) + 1;
        if(XYdata[4] >= 0x80)
            yData = (yData - 65535) + 1;

        newDataReady = 1;
    }

    return newDataReady;
}

// translates magnetometer data into a compass heading
double getCompassHeading()
{
    //subtracts the environmental interference from the data
//  int xMagnetometer = xData - xOffset;
//  int yMagnetometer = yData - yOffset;

    //compass heading in degrees
    double compassHeading = atan2((double)yData, (double)xData) * (double)(180/PI);

    return compassHeading;
}

// converts compass heading to a step offset from north for rangefinder data
double getStepOffset(double compassHeading)
{
    if(compassHeading < 0)
        compassHeading = 360 + compassHeading;

    double stepOffset = compassHeading / DEGREES_PER_STEP;

    return stepOffset;
}
```

E.v Disparity Algorithm Implementation

Matlab Algorithm:

```
% 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);
        maxc = min(imgWidth, n + halfBlockSize);

        % Define the search boundaries as offsets from the template location.
        % Limit the search so that we don't go outside of the image.
        % 'mind' is the the maximum number of pixels we can search to the left.
        % 'maxd' is the maximum number of pixels we can search to the right.
        %
        % In the "Cones" dataset, we only need to search to the right, so mind
        % is 0.
        %
        % For other images which require searching in both directions, set mind
        % as follows:
        %   mind = max(-disparityRange, 1 - minc);
        mind = 0;
        maxd = min(disparityRange, imgWidth - maxc);

        % Select the block from the right image to use as the template.
        template = rightI(minr:maxr, minc:maxc);

        % Get the number of blocks in this search.
        numBlocks = maxd - mind + 1;

        % Create a vector to hold the block differences.
        blockDiffs = zeros(numBlocks, 1);

        % Calculate the difference between the template and each of the blocks.
        for (i = mind : maxd)

            % Select the block from the left image at the distance 'i'.
            block = leftI(minr:maxr, (minc + i):(maxc + i));

            % Compute the 1-based index of this block into the 'blockDiffs' vector.
            blockIndex = i - mind + 1;

            % Take the sum of absolute differences (SAD) between the template
            % and the block and store the resulting value.
            blockDiffs(blockIndex, 1) = sum(sum(abs(template - block)));
        end

        % Sort the SAD values to find the closest match (smallest difference).
        % Discard the sorted vector (the "~" notation), we just want the list
        % of indices.
        [temp, sortedIndeces] = sort(blockDiffs);

        % Get the 1-based index of the closest-matching block.
        bestMatchIndex = sortedIndeces(1, 1);

        % Convert the 1-based index of this block back into an offset.
        % This is the final disparity value produced by basic block matching.
        d = bestMatchIndex + mind - 1;

        % Store the calculated disparity value in the resultant img matrix
        DbasicSubpixel(m, n) = d;
    end

    % Update progress bar every 5th row.
    if (mod(m, 5) == 0)
        str = sprintf('  Image Row %d / %d (%.0f%%)\n', m, imgHeight, (m / imgHeight) * 100)
            ;
        waitbar(m/imgHeight,h,str)
    end

end

% close the progress bar
close(h);
```

```matlab

% 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'));
```

Verilog Algorithm:

```verilog
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Disparity Algorithm implementation
////////////////////////////////////////////////////////////////////////////////
module parallel_disparity(
         input clk, // Read clk signal
         input enable, // Enable new disparity calculation
         input sw, // Input from left/right image buffer controller
         input reset, // Reset disparity FSM
         input [7:0] ldata, // Left bram pixel data in
         input [7:0] rdata, // Right bram pixel data in
         output reg [16:0] laddr, // Left bram read address
         output reg [16:0] raddr, // Right bram read address
         output reg [18:0] result_addr, // Result bram write address
         output reg [7:0] result_data, // Result bram pixel data
         output result_wea, // Result bram write enable
         output [2:0] state_LED, // Current state indicator
         output [7:0] lineout, // Single line pixel data out
         input [6:0] lineaddr // Single line pixel data address
    );

// user-defined constants (image search parameters)
parameter WIDTH = 384 - 1; // output image width (0-indexed)
parameter HEIGHT = 288 - 1; // output image height (0-indexed)
parameter SEARCH_RANGE = 20-1; // disparity block comparison search range (0-indexed)
parameter HALF_BLOCK = 3; // half block size
parameter FOCAL_LENGTH = 6; // 6mm
parameter BASELINE = 63; //63mm

// calculated constants
```

```
parameter BLOCK_SIZE = (2*HALF_BLOCK) + 1; // block size
parameter BLOCK_SIZE0 = (2*HALF_BLOCK); // block size with zero based index
parameter FB = FOCAL_LENGTH*BASELINE; // focal length * baseline (done here to save computation space)

// search variables (incremented automatically)
reg [8:0] col_count = 9'b0;  // number of cols iterated through (m in matlab code)
reg [8:0] row_count = 9'b0; // number of rows iterated through (n in matlab code)
reg [8:0] minr = 9'b0, maxr = 9'b0, t_minc = 9'b0, t_maxc = 9'b0, b_minc = 9'b0, b_maxc = 9'b0; // current search block borders
reg [8:0] rcnt = 9'b0, dcnt=9'b0; // temporary counters based on above wires for search blocks
reg [2:0] cdcnt = 3'b0, rdcnt = 3'b0, ccnt = 3'b0;//temporary counters based on above wires for search blocks
reg [5:0] mind = 6'b0, maxd = 6'b0; // min/max disparity search bounds (limit SEARCH_RANGE to 63 blocks!)
reg [5:0] scnt = 6'b0; // number of disparity search comparisons performed
reg [5:0] numBlocks = 6'b0; // number of blocks within current search bounds
reg [5:0] blockIndex = 6'b0; // current block being searched in numBlocks
reg [5:0] index; // index of the min number in the disparity vector (disparity value)
reg [7:0] min; // value of min number in disparity vector
reg [1:0] pipe = 2'b00; // pipeline control for FSM
reg done; // will be 1 if disparity is 100% done, 0 otherwise (used for next_state == IDLE)

// temporary memory for template block, search block, and computational storage
// note that these blocks are broken down by row to allow for faster summations
// 7x7 template block
reg [7:0] template0 [0:BLOCK_SIZE0]; // template block row 0
reg [7:0] template1 [0:BLOCK_SIZE0]; // template block row 1
reg [7:0] template2 [0:BLOCK_SIZE0]; // template block row 2
reg [7:0] template3 [0:BLOCK_SIZE0]; // template block row 3
reg [7:0] template4 [0:BLOCK_SIZE0]; // template block row 4
reg [7:0] template5 [0:BLOCK_SIZE0]; // template block row 5
reg [7:0] template6 [0:BLOCK_SIZE0]; // template block row 6
// 7x7 search block
reg [7:0] block0 [0:BLOCK_SIZE0]; // search block row 0
reg [7:0] block1 [0:BLOCK_SIZE0]; // search block row 1
reg [7:0] block2 [0:BLOCK_SIZE0]; // search block row 2
reg [7:0] block3 [0:BLOCK_SIZE0]; // search block row 3
reg [7:0] block4 [0:BLOCK_SIZE0]; // search block row 4
reg [7:0] block5 [0:BLOCK_SIZE0]; // search block row 5
reg [7:0] block6 [0:BLOCK_SIZE0]; // search block row 6
// 7x7 storage for abs(template-block)
reg [7:0] SAD_diffs0 [0:BLOCK_SIZE0]; // block for holding abs(template-block) row 0
reg [7:0] SAD_diffs1 [0:BLOCK_SIZE0]; // block for holding abs(template-block) row 1
reg [7:0] SAD_diffs2 [0:BLOCK_SIZE0]; // block for holding abs(template-block) row 2
reg [7:0] SAD_diffs3 [0:BLOCK_SIZE0]; // block for holding abs(template-block) row 3
reg [7:0] SAD_diffs4 [0:BLOCK_SIZE0]; // block for holding abs(template-block) row 4
reg [7:0] SAD_diffs5 [0:BLOCK_SIZE0]; // block for holding abs(template-block) row 5
reg [7:0] SAD_diffs6 [0:BLOCK_SIZE0]; // block for holding abs(template-block) row 6
// 7x1 storage for sum(abs(template-block))
reg [10:0] temp0; // block for holding sum(abs(template-block)) row 0
reg [10:0] temp1; // block for holding sum(abs(template-block)) row 1
reg [10:0] temp2; // block for holding sum(abs(template-block)) row 2
reg [10:0] temp3; // block for holding sum(abs(template-block)) row 3
reg [10:0] temp4; // block for holding sum(abs(template-block)) row 4
reg [10:0] temp5; // block for holding sum(abs(template-block)) row 5
reg [10:0] temp6; // block for holding sum(abs(template-block)) row 6
// 20x1 for storing SAD value for ALL search blocks corresponding to a single template block
reg [14:0] SAD_vector [0:SEARCH_RANGE]; // block for holding sum(sum(abs(template-block))) - up to 9x9 block size
reg [10:0] line; // reg for holding [(focal length)*(baseline)]/index

// ~~~~~~~~~~~~~~ Disparity FSM ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
parameter [2:0] IDLE = 3'b000, // wait for next read sequence
                READ = 3'b001, // read data from FIFO
                                SEPARATE = 3'b010, // separate search image into rows
                                SAD = 3'b011, // perfom sum of absolute diff's
```

```
                                FINALIZE = 3'b100; // search for max vector values, store in disparity matrix
reg [2:0] current_state = IDLE,
          next_state = IDLE;

// state-machine flip-flops
always @ (posedge clk)
        if(reset) // synchronous to protect bram
                current_state <= IDLE;
        else
                current_state <= next_state;

// next state logic
always @(current_state,enable,ccnt,dcnt,pipe,done,rcnt,maxd)
        case(current_state)
        IDLE: // wait for new sequence enable
                if(enable)
                        next_state = READ;
                else
                        next_state = IDLE;
        READ: // previously stored logic now contained in imgbuf.v
                next_state = SEPARATE;
        SEPARATE: // isolate template and search block
                if(ccnt == (BLOCK_SIZE0) && rcnt == (BLOCK_SIZE0))
                        next_state = SAD;
                else
                        next_state = SEPARATE;
        SAD: // perform sum(sum(abs(template-search)))
                if(dcnt < maxd && pipe == 2'b11)
                        next_state = SEPARATE;
                else if (dcnt < maxd || pipe < 2'b11)
                        next_state = SAD;
                else
                        next_state = FINALIZE;
        FINALIZE: // Find disparity value from SAD vector and store in output buffer
                if(~done && pipe == 2'b11)
                        next_state = SEPARATE;
                else if(done && pipe == 2'b11)
                        next_state = IDLE;
                else
                        next_state = FINALIZE;
        default: next_state = IDLE;
        endcase

// FSM disparity Implementation
always @(posedge clk)
case(current_state)
        IDLE: // wait for next read sequence
        begin
            if(~sw)
                  row_count <= 9'b0;
                else
                  row_count <= 9'd144;
                col_count <= 9'b0;
                dcnt <= 9'b0;
                pipe <= 2'b00;
        end

        READ: // read in image data from buffers
        begin
                pipe <= 2'b00;
        end

        SEPARATE: // Read in new block data for next comparison
        begin
                // read in the template and search blocks IN PARALLEL as set by the following:
                // template block: (t_minc:t_maxc,minr:maxr)
```

```
    // search block: (b_minc:b_maxc,minr:maxr)
    // read in template image block
    if(ccnt <= (t_maxc-t_minc) && rcnt <= (maxr-minr)) // fully within template search bounds
            case(rcnt)
     0:  template0[ccnt] <= ldata;
     1:  template1[ccnt] <= ldata;
     2:  template2[ccnt] <= ldata;
     3:  template3[ccnt] <= ldata;
     4:  template4[ccnt] <= ldata;
     5:  template5[ccnt] <= ldata;
     6:  template6[ccnt] <= ldata;
             endcase
    else // outside tempate bounds
            case(rcnt)
     0:  template0[ccnt] <= 8'h00;
     1:  template1[ccnt] <= 8'h00;
     2:  template2[ccnt] <= 8'h00;
     3:  template3[ccnt] <= 8'h00;
     4:  template4[ccnt] <= 8'h00;
     5:  template5[ccnt] <= 8'h00;
     6:  template6[ccnt] <= 8'h00;
 endcase

    // read in search image block
    if(ccnt <= (b_maxc-b_minc) && rcnt <= (maxr-minr)) // fully within template search bounds
case(rcnt)
    0: block0[ccnt] <= rdata;
    1: block1[ccnt] <= rdata;
    2: block2[ccnt] <= rdata;
    3: block3[ccnt] <= rdata;
    4: block4[ccnt] <= rdata;
    5: block5[ccnt] <= rdata;
    6: block6[ccnt] <= rdata;
            endcase
    else // outside tempate bounds
            case(rcnt)
    0: block0[ccnt] <= 8'h00;
    1: block1[ccnt] <= 8'h00;
    2: block2[ccnt] <= 8'h00;
    3: block3[ccnt] <= 8'h00;
    4: block4[ccnt] <= 8'h00;
    5: block5[ccnt] <= 8'h00;
    6: block6[ccnt] <= 8'h00;
endcase

    // increment ccnt and rcnt to iterate through all pixels within blocks
    if(pipe == 2'b11) begin
        pipe <= 2'b00;
if(ccnt<(BLOCK_SIZE0))
    ccnt<=ccnt+1'b1;
else if(rcnt<(BLOCK_SIZE0) && ccnt==(BLOCK_SIZE0)) begin
    rcnt <= rcnt+1'b1;
    ccnt <= 3'b0;
end
    end
    else
      pipe <= pipe + 1'b1;

    // make sure pipe is clear for SAD
    if(next_state == SAD)begin
            pipe <= 2'b00;
            ccnt <= 3'b0;
            rcnt <= 9'b0;
            cdcnt <= 3'b0;
            rdcnt <= 3'b0;
    end
```

```
        end

        SAD:
        begin
        // ~~~~~~~~~~~~~~~~ abs(template-block) ~~~~~~~~~~~~~~~~
                if (pipe == 2'b00) begin
                    // abs0
                        if(template0[ccnt]>block0[ccnt])
                                SAD_diffs0[ccnt] <= template0[ccnt] - block0[ccnt];
                        else
                                SAD_diffs0[ccnt] <= block0[ccnt] - template0[ccnt];
                    // abs1
                    if(template1[ccnt]>block1[ccnt])
                SAD_diffs1[ccnt] <= template1[ccnt] - block1[ccnt];
            else
                SAD_diffs1[ccnt] <= block1[ccnt] - template1[ccnt];
            // abs2
            if(template2[ccnt]>block2[ccnt])
                SAD_diffs2[ccnt] <= template2[ccnt] - block2[ccnt];
            else
                SAD_diffs2[ccnt] <= block2[ccnt] - template2[ccnt];
            // abs3
            if(template3[ccnt]>block3[ccnt])
                SAD_diffs3[ccnt] <= template3[ccnt] - block3[ccnt];
            else
                SAD_diffs3[ccnt] <= block3[ccnt] - template3[ccnt];
            // abs4
            if(template4[ccnt]>block4[ccnt])
                SAD_diffs4[ccnt] <= template4[ccnt] - block4[ccnt];
            else
                SAD_diffs4[ccnt] <= block4[ccnt] - template4[ccnt];
            // abs5
            if(template5[ccnt]>block5[ccnt])
                SAD_diffs5[ccnt] <= template5[ccnt] - block5[ccnt];
            else
                SAD_diffs5[ccnt] <= block5[ccnt] - template5[ccnt];
            // abs6
            if(template6[ccnt]>block6[ccnt])
                SAD_diffs6[ccnt] <= template6[ccnt] - block6[ccnt];
            else
                SAD_diffs6[ccnt] <= block6[ccnt] - template6[ccnt];

            // increment through each index in all template and search rows
                        if(ccnt<(BLOCK_SIZE0))
                                ccnt<=ccnt+1'b1;
                        else
                                pipe <= 2'b01; // proceed to next stage of SAD
                end

                // ~~~~~~~~~~~~~~~~ sum(abs(template-block)) ~~~~~~~~~~~~~~~~
                if(pipe == 2'b01) begin
                        if(cdcnt < BLOCK_SIZE) begin // 0 .. block_size-1
                case(rdcnt) // sum(abs0 + abs1 + abs2 + abs3 + abs4 + abs5 + abs6) for all
                    columns within each row
                    0: temp0 <= SAD_diffs0[cdcnt] + SAD_diffs1[cdcnt] + SAD_diffs2[cdcnt] +
                        SAD_diffs3[cdcnt] + SAD_diffs4[cdcnt] + SAD_diffs5[cdcnt] +
                        SAD_diffs6[cdcnt];
                    1: temp1 <= SAD_diffs0[cdcnt] + SAD_diffs1[cdcnt] + SAD_diffs2[cdcnt] +
                        SAD_diffs3[cdcnt] + SAD_diffs4[cdcnt] + SAD_diffs5[cdcnt] +
                        SAD_diffs6[cdcnt];
                    2: temp2 <= SAD_diffs0[cdcnt] + SAD_diffs1[cdcnt] + SAD_diffs2[cdcnt] +
                        SAD_diffs3[cdcnt] + SAD_diffs4[cdcnt] + SAD_diffs5[cdcnt] +
                        SAD_diffs6[cdcnt];
                    3: temp3 <= SAD_diffs0[cdcnt] + SAD_diffs1[cdcnt] + SAD_diffs2[cdcnt] +
                        SAD_diffs3[cdcnt] + SAD_diffs4[cdcnt] + SAD_diffs5[cdcnt] +
                        SAD_diffs6[cdcnt];
                    4: temp4 <= SAD_diffs0[cdcnt] + SAD_diffs1[cdcnt] + SAD_diffs2[cdcnt] +
                        SAD_diffs3[cdcnt] + SAD_diffs4[cdcnt] + SAD_diffs5[cdcnt] +
```

```
                    SAD_diffs6[cdcnt];
            5: temp5 <= SAD_diffs0[cdcnt] + SAD_diffs1[cdcnt] + SAD_diffs2[cdcnt] + SAD_diffs3[cdcnt] + SAD_diffs4[cdcnt] + SAD_diffs5[cdcnt] + SAD_diffs6[cdcnt];
            6: temp6 <= SAD_diffs0[cdcnt] + SAD_diffs1[cdcnt] + SAD_diffs2[cdcnt] + SAD_diffs3[cdcnt] + SAD_diffs4[cdcnt] + SAD_diffs5[cdcnt] + SAD_diffs6[cdcnt];
                    endcase
                end else begin // avg accross block width when one sum is done temp[idx]=[sum[idx](0..7)/7]
                    case(rdcnt)
            0: temp0 <= (temp0/BLOCK_SIZE);
            1: temp1 <= (temp1/BLOCK_SIZE);
            2: temp2 <= (temp2/BLOCK_SIZE);
            3: temp3 <= (temp3/BLOCK_SIZE);
            4: temp4 <= (temp4/BLOCK_SIZE);
            5: temp5 <= (temp5/BLOCK_SIZE);
            6: temp6 <= (temp6/BLOCK_SIZE);
                        endcase
            end

                // iterate through each colum within each row
                if(cdcnt<BLOCK_SIZE)
                        cdcnt<=cdcnt+1'b1;
            // after finishing all sums, reduce to an average value
            else if(cdcnt == BLOCK_SIZE && rdcnt < BLOCK_SIZE0) begin
                rdcnt <= rdcnt + 1'b1;
                cdcnt <= 0;
            // when finished, reset for next stage of SAD
                end else begin
                        pipe <= 2'b10;
                        ccnt <= 3'b0;
                        rcnt <= 9'b0;
                        cdcnt <= 3'b0;
                        rdcnt <= 3'b0;
                end
        end

        // ~~~~~~~~~~~~~~~~ sum(sum(abs(template-block))) ~~~~~~~~~~~~~~~~
        if (pipe == 2'b10) begin // pipe = 2'b10
                if(ccnt<3'b001) begin
                        SAD_vector[blockIndex] <= temp0+temp1+temp2+temp3+temp4+temp5+temp6;
                        ccnt <= ccnt + 1'b1;
                end
                else begin
                        SAD_vector[blockIndex] <= SAD_vector[blockIndex]/(BLOCK_SIZE);
                        ccnt <= 3'b0;
                        pipe <= 2'b11;
                end
        end

// update SAD vector index (when full, proceed to finalization)
// this index represents the position of the current search block
// in relation to the template block (0-19 for a 20px disparity search)
        if(dcnt < maxd && pipe == 2'b11) begin
                dcnt <= dcnt + 1'b1; // number of searches performed
                blockIndex <= dcnt - mind; // index in SAD_vector
        end

        // update cols & rows processed by the algorithm after comparing
        // SEARCH_RANGE search blocks to the current template
        if (next_state == FINALIZE) begin
                scnt <= 6'b0;
                pipe <= 2'b00;

                // increment accross each row of pixels
```

```
                if(col_count < (WIDTH-(HALF_BLOCK+1'b1)))
        col_count <= col_count + 1'b1;
    // increment through all rows if NOT performing a line search
                else if (~sw && col_count == (WIDTH-(HALF_BLOCK+1'b1)) && row_count < HEIGHT) begin
        row_count <= row_count + 1'b1;
        col_count <= 9'b0;
            end
            // increment through two rows if performing a line search
        else if (sw && col_count == (WIDTH-(HALF_BLOCK+1'b1)) && row_count < 9'd145) begin
            if(row_count < 9'd144)
               row_count <= 9'd144;
            else
            row_count <= row_count + 1'b1;
        col_count <= 9'b0;
                end

                // full disparity search complete when all rows and cols have been processed
                if(~sw && col_count == (WIDTH-(HALF_BLOCK+1'b1)) && row_count == HEIGHT)
                   done <= 1'b1;
                // line disparity search complete when two most central rows and cols have been processed
                else if (sw && col_count == (WIDTH-(HALF_BLOCK+1'b1)) && row_count >= 9'd145)
                   done <= 1'b1;
                // if neither search is complete, allow for the FSM to keep cycling
                else
                   done <= 1'b0;
        end

        // reset for a new SAD sequence if there are more search blocks to compare to the template
        if (next_state == SEPARATE) begin
                ccnt <= 3'b0;
                rcnt <= 9'b0;
                cdcnt <= 3'b0;
                rdcnt <= 3'b0;
                pipe <= 2'b00;
        end
end

FINALIZE:
begin
        dcnt <= 9'b0;
        // search for index of min value in SAD_vector
        // this index corresponds to the pixel offset between the template
        // block and the closest matching search block
        if(scnt<numBlocks) begin
            // value at idx=0 will always be the lowest value to start...
                if(scnt == 6'b0) begin
                        min <= SAD_vector[0];
                        index <= 8'h00;
                        end
            // update this value and its index while incrementing through...
                else if(SAD_vector[scnt]<min) begin
                        min <= SAD_vector[scnt];
                        index <= scnt;
                        end
                scnt <= scnt + 1'b1;
        end

        // place disparity value in output image array after finding the index
        else begin
    if(sw == 1'b0) // use disparity values when in full search mode
        result_data <= index;
```

```
            else // use depth values when in line mode
                // in this case I sum 8 pixel values accross two lines for every output pixel
                // non-zero pixel value on any col other than 1st
                if(index > 0 && pipe > 2'b00)
                  line <= line + (FB/index);
                // non-zero pixel value on first column of pixels
                else if (index > 0)
                  line <= (FB/index);
                // zero pixel value on first row of pixels
                else if (row_count == 9'd144 && pipe <= 2'b00)
                  line <= 8'h00;
                // zero pixel value in any other location
                else
                  line = line;

            // count through 4 pixels for each iteration of line mode
            pipe <= pipe + 1'b1; //pipe <= 2'b11; // was 2'b11, changed to add a little extra time
                end
        end
endcase

// single line pixel buffer
line_bram linebuf (
  .clka(clk),    // input wire clka
  .wea(pipe == 2'b11 && sw && row_count == 9'd144),// input wire [0 : 0] wea
  .addra(col_count[8:2]),  // addra will increment every 4 pixels
  .dina(line>>3),    // dina is the sum of 8 depth values / 8
  .clkb(clk),    // input wire clkb
  .addrb(lineaddr),  // line address from top-level display logic
  .doutb(lineout)  // depth value to top-level display logic
);

// disparity value output buffer write enable
assign result_wea = (current_state == FINALIZE && scnt == (numBlocks)) ? 1'b1 : 1'b0;

// result address
always @(posedge clk)
    result_addr = ((WIDTH+1'b1)*row_count)+col_count;

// left image buffer read address (for template block)
always @(posedge clk)
    laddr = ((WIDTH+1'b1)*(minr+rcnt))+(t_minc+ccnt);

// right image buffer read address (for search block)
always @(posedge clk)
    raddr = ((WIDTH+1'b1)*(minr+rcnt))+(b_minc+ccnt);

// assign disparity block search bounds
always @(row_count,col_count,t_maxc,maxd,mind,dcnt)
begin
                minr = (0 > $signed(row_count - HALF_BLOCK)) ? 9'b0 : (row_count - HALF_BLOCK);
                maxr = ((HEIGHT) < (row_count + HALF_BLOCK)) ? HEIGHT : (row_count + HALF_BLOCK);
                t_minc = (0 > $signed(col_count - HALF_BLOCK)) ? 9'b0 : (col_count - HALF_BLOCK);
                t_maxc = ((WIDTH) < (col_count + HALF_BLOCK)) ? WIDTH : (col_count + HALF_BLOCK);
                b_minc =(0 > $signed(dcnt - HALF_BLOCK+col_count)) ? 9'b0 : (dcnt - HALF_BLOCK + col_count);
                b_maxc = ((WIDTH) < (dcnt + HALF_BLOCK)) ? WIDTH : (dcnt + col_count + HALF_BLOCK);
end

// assign disparity search bounds
always @(t_maxc,maxd,mind,current_state)
```

```
        begin
                mind = 6'b0; // or = max(-SEARCH_RANGE, 1-t_minc)
                if (current_state == READ)
                        maxd = SEARCH_RANGE;
                else if(current_state == FINALIZE)
                        maxd = (SEARCH_RANGE < ((WIDTH) - t_maxc)) ? SEARCH_RANGE : ((WIDTH)
                             - t_maxc);
                numBlocks = maxd - mind;
end

// current state indicator LED
assign state_LED = current_state;

endmodule
```