FPGA-Based Real-Time SLAM

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 19, 2017

Advisor:
Professor R. James Duckworth

Contents

List of Figures

Abstract

This project created a proof of concept SLAM sensor suite capable of remotely observing and mapping areas by combining real-time stereo camera imagery with distance measurements and localization data to generate a 3D depth map and 2D floorplan of its environment. The system used a Xilinx Zynq SoC containing an embedded ARM processor and FPGA fabric, and implemented unique SLAM processing functionality dependent on both software and parallelized custom logic.

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 raw 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 the same type of system through Simultaneous Localization and Mapping (SLAM). Through performing onboard image processing on the sensor suite itself, valuable information can be extracted from the images, such as depth information or even human detection, before its wireless transmission.

This project sought to fill this gap in existing technology by creating a sensor suite that processed real-time stereo camera imagery onboard and combined it with localization and distance data to generate a 3D depth map and 2D floorplan of the sensor suite's environment. Field Programmable Gate Arrays (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 FPGA platform used to create this proof of concept sensor suite was an Avnet ZedBoard, which contained a Xilinx Zynq-7020 All-Programmable System on Chip (SoC). The Xilinx Zynq-7020 SoC consists of a dual-core ARM Cortex A9 processor coupled with Xilinx Artix-7 FPGA fabric. Having both an embedded processor and digital logic on the same chip allowed the sensor suite to be designed in a way that used each for their unique advantages. The embedded processor was mainly used to communicate with the necessary peripherals using Xilinx's built-in peripheral drivers, whereas the digital logic was used to parallelize the data processing and interface with memory.

The sensors used in this project included a pair of MT9V034 monochrome camera modules. These cameras were mounted on a custom-made stereo camera printed circuit board

developed during the initial testing stages of the project. The MT9V034 camera module is a global-shutter image sensor capable of capturing WVGA imagery at 60 frames per second. A Hokuyo URG-04LX scanning laser rangefinder was also used to estimate the distance of objects closest to the device. This rangefinder has a 240° field of view and 99% distance precision. In order to geographically reference data with digital compass readings, the magnetometer on the Digilent PmodNAV Inertial Measurement Unit (IMU) was also used.

The first stage of data processing consisted of a dual image buffer controller used for triggering image captures and reading stereo camera imagery into local memory on the Zynq-7020 processor. Using an image processing technique known as disparity mapping, the stereo camera imagery was converted into 3D depth maps. A portion of this data was then stored in local memory for correlation with data from the 2D scanning laser rangefinder. Distance data from the rangefinder and compass data from the magnetometer were simultaneously pre-processed on the ARM Cortex A9 processor, and then written to the system's programmable logic to undergo further coordinate-axis transformation.

Through its data processing stages, the sensor suite produced a real-time 3D depth map in addition to a compass-referenced 2D map of the objects closest to the device. These continuously updated outputs were viewable through the use of an attached VGA display, and were selectable based on user inputs. The overall result of this implementation was a proof of concept SLAM sensor suite that could serve as an improvement to a simple imaging sensor on a remote robotic platform.

Existing robotic surveillance systems use camera modules to transmit real-time video streams of their surroundings. In addition to outputting a simple video stream, the proof of concept sensor suite developed through this project extracted depth information from stereo cameras and used it to create a real-time depth map. Simultaneously, the sensor suite created a compass-referenced 2D floorplan of its environment. The final product sensor suite proved that more situational awareness was available than was being provided by existing solutions, and that FPGAs were a viable tool for this type of application.

For future work, we recommend incorporating IMU displacement data in order to create visualizations of the entire path traversed by the device. In addition, incorporating human detection would be an improvement that could be added to the image processing portion of this project. A human detection algorithm could be combined with the existing 2D and 3D depth information in order to create an all-encompassing sensor suite for situational awareness analysis.

1 Introduction

In recent years, improvements in embedded processing technology have allowed for the creation of robotic situational awareness platforms for remotely observing dangerous or inaccessible areas. The market for such devices is a new and expanding field, and faces large demand from the military and first responders. This field is being piloted by throwable and remotely drivable platforms such as the Endeavor Robotics 110 FirstLook and the Bounce Imaging Explorer, shown in Figure 1.

(a) Endeavor Robotics 110 FirstLook [12]

(b) Bounce Imaging Explorer [7]

Figure 1: Robotic Situational Awareness Devices

These devices contain simple wireless video streaming technology, allowing for remote visual surveillance and situational analysis. Although a video stream is an effective strategy for simple observation, this method of gathering information has room for improvement. This project investigated the extraction of information such as object positioning and localization from a camera-based sensor suite in real time, allowing for more comprehensive situational observation. One method of performing this process is Simultaneous Localization and Mapping, or SLAM.

SLAM is the technique of mapping an unknown environment with respect to an agent, and can be performed using a wide variety of sensors and computational methods. SLAM is

a common area of research in the fields of image processing and high-speed computing, and has been applied mainly to autonomous vehicles. Most current SLAM implementations rely on the use of a sensor suite connected to a computer or System on Chip (SoC) computing device.

One type of technology useful for performing the high speed data processing necessary for SLAM is a Field Programmable Gate Array, or FPGA. FPGAs consist of digital logic gates that are designed to be user-configured, allowing for the creation of completely customized digital hardware. FPGAs are especially useful for parallelized data processing, posing potential real-time advantages over standard computing or microcontroller technology. Although FPGA technology is highly applicable to performing SLAM-like tasks, there are currently few existing products that use FPGAs for this purpose.

This project explored the viability of an FPGA-based real-time SLAM sensor suite as a replacement for standard video cameras on existing situational awareness systems. This sensor suite utilized data from stereo camera modules, a scanning laser rangefinder, and an Inertial Measurement Unit (IMU) to create a real-time depth-augmented video feed and compass-referenced 2D floorplan of the device's field of view.

The following chapters detail the creation of this device, beginning with an exploration of relevant technology and prior work. The overall system design and processing methods used are examined at a high level in the following section. Next, the implementation requirements of each individual sensor are explored in more detail. Methods for processing and combining sensor data for the production of a 2D floorplan and 3D depth visualization are then detailed. Each of these methods are then verified through comprehensive testing, and a finalized design is demonstrated. Lastly, conclusions and recommendations for future improvements are presented.

2 Background

The initial research portion of this project focused on learning more about different remote situational awareness products and how they may have been improved using existing technology, eventually leading to a set of overall project goals.

2.1 Remote Situational Awareness Products

Many situational awareness devices consist of remote-controlled throwable robots with wireless video-streaming capability. One such example is the 110 FirstLook by Endeavor Robotics, seen in Figure 1a. This device is a throwable, rugged robot that streams real-time video of its surroundings, and is used to investigate dangerous locations and hazardous material while keeping its operator out of harm's way. The 110 FirstLook has four day and night cameras, and also supports two-way audio. The device is remotely controlled by a tablet operator control unit, and is currently in use for military applications [12].

Similar to the 110 FirstLook, the Bounce Image Explorer is a throwable camera ball that wirelessly transmits a 360° real-time video stream of its surroundings, and can be seen in Figure 1b. The Explorer processes input from six monochrome WVGA camera modules, and outputs a video stream that can be accessed from a tablet or smartphone. This device is currently in a trial phase with United States Law Enforcement [7].

A more commercial remote situational awareness device is the Serveball Squito™ [19]. Squito is a wireless, throwable, 360° panoramic camera that implements target detection to produce a stabilized output video stream. This device is shown in Figure 2 below.

Figure 2: Serveball's Squito [19]

Squito utilizes a microprocessor receiving input from a fiber optic camera interface, as well as orientation and position sensors, in order to transmit a real-time stabilized video of its surroundings. The image in Figure 3 shows the input from the Squito's four camera inputs on the left, and a corresponding stitched output on the right. The device is still in the prototype stage, and has received interest from the first responder community.

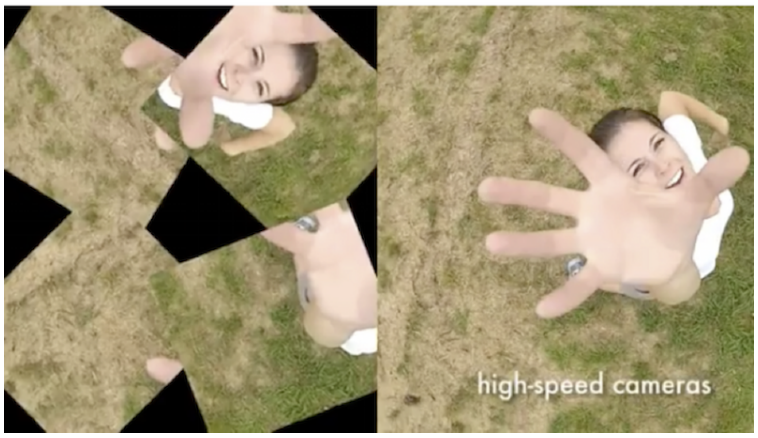


Figure 3: Serveball's Squito Input and Output [19]

The 110 FirstLook, Explorer, and Squito are all examples of remote situational awareness products that contain basic controls and real time video streaming outputs. Using additional image processing in the form of Simultaneous Localization and Mapping, the outputs of each of these devices could be improved to allow for more comprehensive situational awareness.

2.2 Simultaneous Localization and Mapping

As mentioned in the previous chapter, Simultaneous Localization and Mapping is the technique of mapping an unknown environment with respect to a localized agent. SLAM is especially useful for autonomous systems and remote observation, as it can be used for situational analysis and response. Recent research in the field of SLAM has focused on making these systems more portable.

One application of such a system was a proof of concept of camera-based SLAM implementation for feature identification presented by Andrew Davison of Oxford University [9]. This system was handheld, and relied on a computer using a 2.2 GHz Pentium processor connected to a single camera and laser rangefinder. This system implemented edge detection on a known environment, and produced a real-time video output containing a 3D feature localization plot. An output frame from the device is shown in Figure 4.

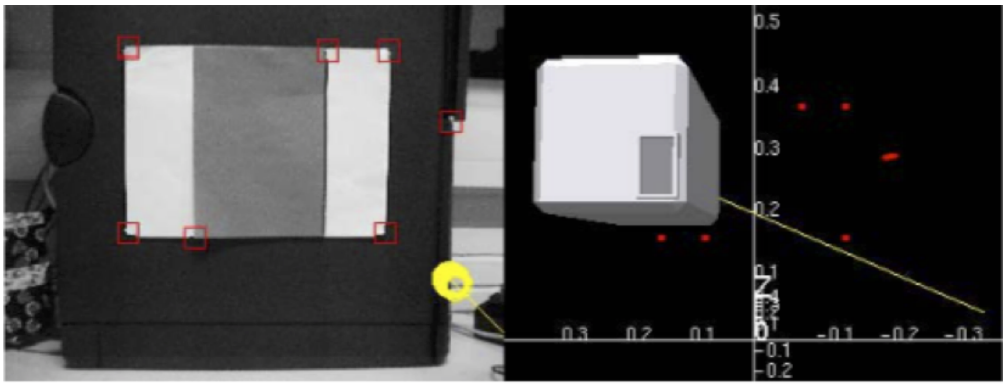


Figure 4: Real-Time SLAM with a Single Camera [9]

The left image in Figure 4 shows six points of a paper target that were input to the system as prior knowledge, along with successfully marked identifying features (marked as red squares), and another identifying feature that was not marked for measurement (marked by a yellow circle). The frame on the right is a localization plot that displayed the relative positions of all red squares detected by the device.

Along with identifying features of interest, SLAM image processing techniques are also used for depth estimation. The process of estimating depth from imagery is known as dis-

parity mapping. Disparity mapping algorithms are used to calculate the similarities between stereo camera image pairs, and to convert said similarities to relative depth measurements. Disparity mapping is useful for situational awareness because it is used to determine the exact locations of all objects within a sensor suite's field of view.

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 disparity image from this implementation is shown in Figure 5 [20]. Using their stereo disparity implementation, the researchers were able to generate real-time video showing the relative locations of objects within the device's field of view. The relative distance from the device to a given object was displayed using a color gradient, with nearer objects shown in brighter colors. Based on this depth information, it was also possible for the researchers to detect objects located within the field of view of the stereo imaging system, as shown in Figure 5. This implementation was extremely applicable to situational awareness systems, as it allowed for the localization of objects and creation of 2D slices of an area in real-time using only two camera sensors.

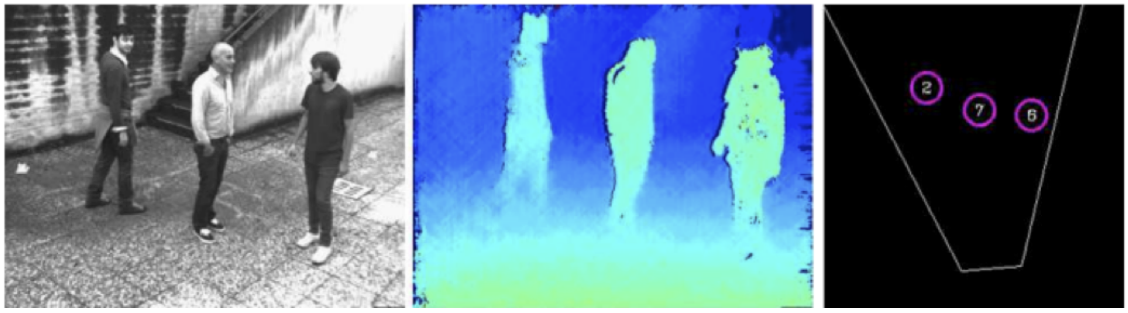


Figure 5: From Left to Right: Original Image, Disparity Map, Object Detection Results [20]

A major concern with calculating disparity in real time is image processing speed. In the case of the implementation shown above, the parallelized data processing capabilities of FPGA hardware were used to address these concerns. The successful creation of this system demonstrated that FPGAs are useful for complex image processing applications [20].

2.3 The Zynq Evaluation and Development Board (ZedBoard)

The FPGA platform used in this project was the Zynq Evaluation and Development Board, or ZedBoard. The ZedBoard is a low-cost development board containing a Xilinx Zynq-7020 All-Programmable SoC, and is shown in Figure 6.

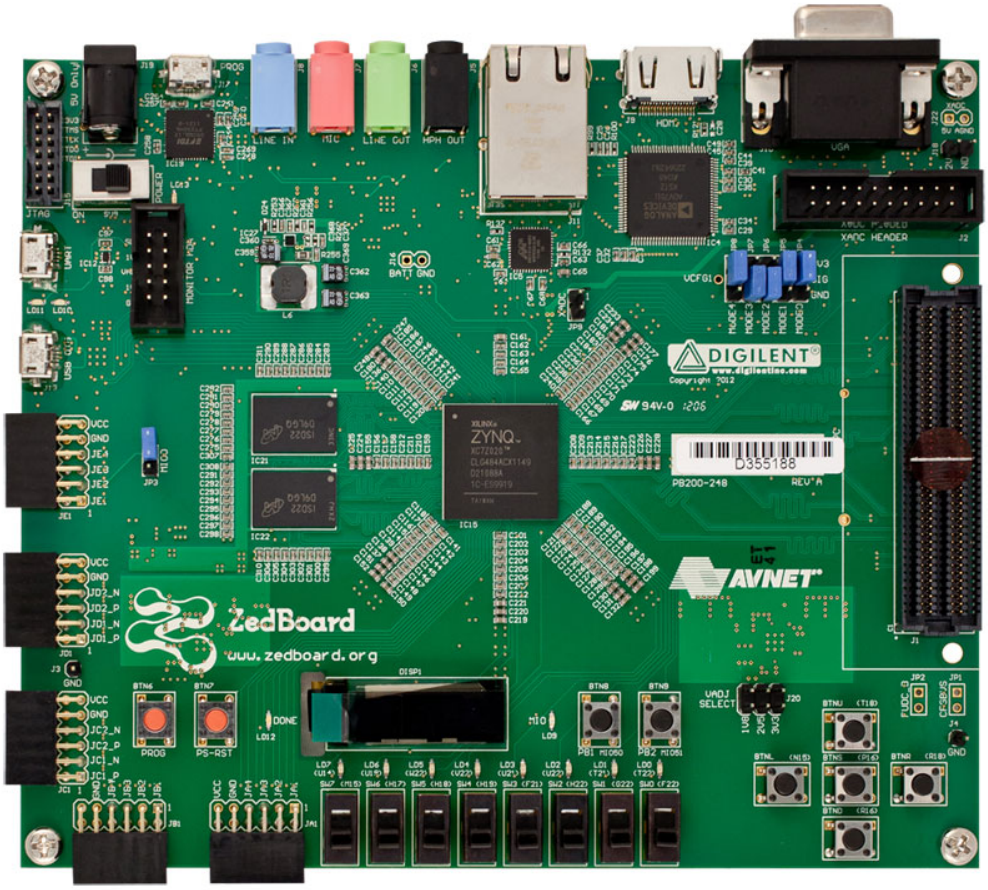


Figure 6: The Avnet ZedBoard [36]

The Xilinx Zynq-7020 SoC consists of a dual-core ARM Cortex A9 processor coupled with Xilinx Artix-7 FPGA fabric. The ARM Cortex A9 processor uses a dedicated 33.3333 MHz clock source, while the onboard 100 MHz oscillator supplies the Programmable Logic (PL) clock. The Zynq-7020 SoC contains 85,000 programmable logic cells with 140 36K Block RAM modules. The ZedBoard also features 5 Pmod IO ports, 8 LEDs, 8 switches, 7 push buttons, a USB UART port, and a VGA port [6]. These are shown in the ZedBoard's

block diagram in Figure 7.

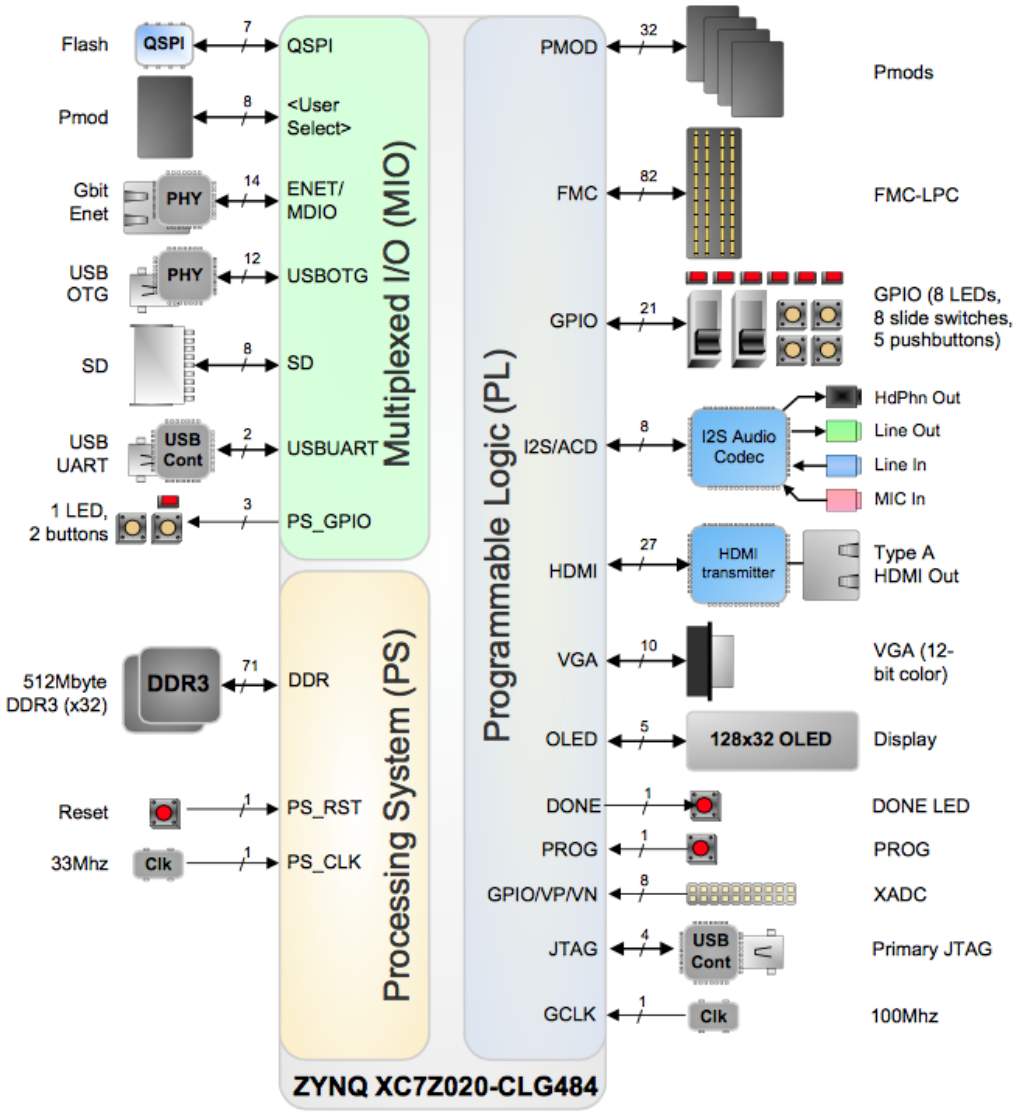


Figure 7: ZedBoard Block Diagram [6]

As our research progressed, it became evident that FPGAs were a viable solution for implementing real-time situational awareness algorithms on a compact scale. For the purposes of this project, we decided to interface the ZedBoard with a stereo camera pair to gather disparity depth information on an area, and supplement that data with digital compass and rangefinder readings to produce detailed maps of the sensor suite's surroundings in real time. The implementation scheme for this device is detailed in the following chapter.

3 System Design

The system used in this project was designed to take advantage of the hardware-software interfaces of the Zynq SoC. An overall block diagram of the system is shown in Figure 8 below. The design was broken down into several functional blocks. These include the sensor hardware shown in red, programmable logic (PL) shown in green, and programmable software (PS) shown in blue.

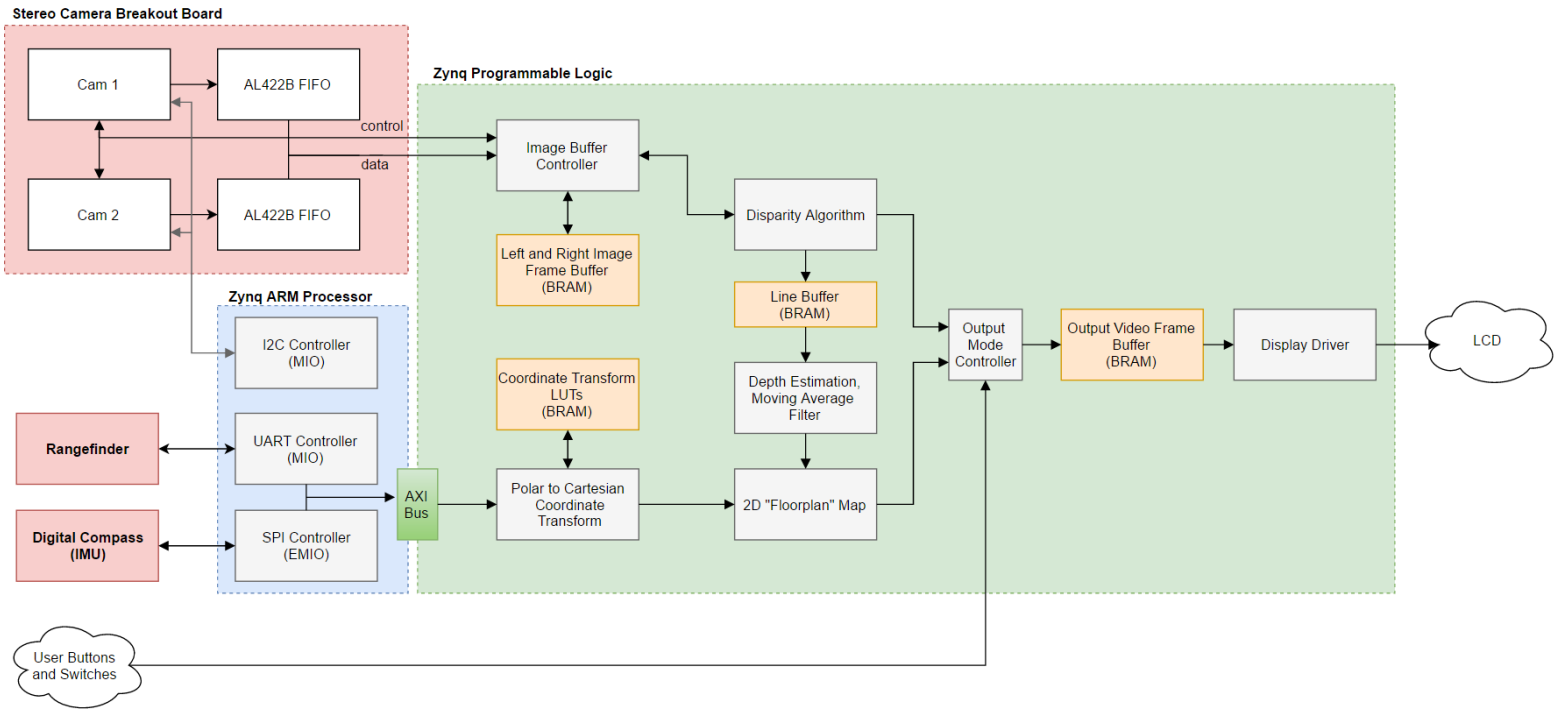


Figure 8: System Block Diagram

The majority of the design was implemented in PL rather than in PS in order to maximize the amount of calculations performed in parallel. For ease of integration, communication with the rangefinder and digital compass was handled using the ARM processor of the Zynq SoC, as pre-configured SPI and UART peripheral drivers were available through Xilinx. Data obtained through these interfaces was then pre-processed before being passed to the FPGA fabric, allowing for rangefinder distance data to be rotated based on the digital compass heading. A customized Advanced eXtensible Interface (AXI) peripheral bus was

then used to pass said data from the ARM processor to the programmable logic. Within the programmable logic, a pair of lookup tables were used to convert pre-rotated rangefinder distance data from Polar to Cartesian coordinates to create a 2D "floorplan" of the sensor suite's environment.

Along with pre-processing compass and rangefinder data, the PS was used to initialize each camera module for manual trigger mode per an I^2C peripheral driver at startup. Following this initialization sequence the PS continuously gathered new data samples from the rangefinder and compass. Simultaneously, a separate PL module triggered new image captures from the stereo camera pair. Since the camera modules immediately output their image data after a new trigger sequence, a customized stereo camera printed circuit board was designed with attached image buffer integrated circuits (ICs) for temporarily storing image data. This intermediate buffering stage allowed the programmable logic to read image data into local memory using separate clock domains, easing the overall timing requirements of the image processing pipeline. After stereo camera image data had been read into on-chip memory, a disparity module was then used to calculate the relative offset between objects contained in the stereo image pair. The resultant disparity image was stored in a video frame buffer for VGA display. A portion of this data was also stored in a separate memory buffer for correlation with data from the 2D scanning laser rangefinder.

In order to correlate disparity data with 2D depth information from the scanning laser rangefinder, several horizontal lines of pixel data from the disparity algorithm were passed through a moving average filter. This filter was used to smooth the disparity depth data, as well as to account for differences in the field of view of each sensor. After this processing stage, the 2D disparity and laser rangefinder data was combined directly. Depending on the status of the ZedBoard's user switches, this 2D floorplan map was then passed to the output video buffer for external display.

The system implementation is explored further in the next chapter.

4 System Implementation

After deciding that the proof of concept sensor suite required 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 theoretical implementation of the overall system.

4.1 Rangefinder Operation

A rangefinder is a device that estimates the distance of the objects closest to it. Because this sensor suite was intended to traverse unknown locations and create a 2-dimensional map, data accuracy, precision, and reliability were vital design requirements.

4.1.1 Selection

The project's rangefinder selection depended on the following criteria: field of vision, maximum sensible depth, accuracy, precision, and cost. Many of the rangefinders limited by the project's budgetary restrictions were severely lacking in at least 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 9, 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 was perfect for our application [15].

Figure 9: URG-04LX Scanning Laser Rangefinder [14]

4.1.2 Communication

The URG-04LX rangefinder used the Recommend Standard (RS) 232C protocol via Universal Asynchronous Receiver/Transmitter (UART) communication. RS-232 is a form of differential serial data transmission which recognizes a digital logic high from -3V to -25V, and a digital logic low from +3V to +25V [3].

Since the ZedBoard's Peripheral Module (Pmod) connectors supported UART communication, the rangefinder was communicated with using a Pmod IO connector. The ZedBoard's Pmod connectors used the Transistor-Transistor Logic (TTL) protocol, 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 [23]. Since TTL has different logic levels than RS-232, these protocols were incompatible and did not recognize each other. Figure 10 shows a timing diagram of both RS-232 and TTL communication protocols.

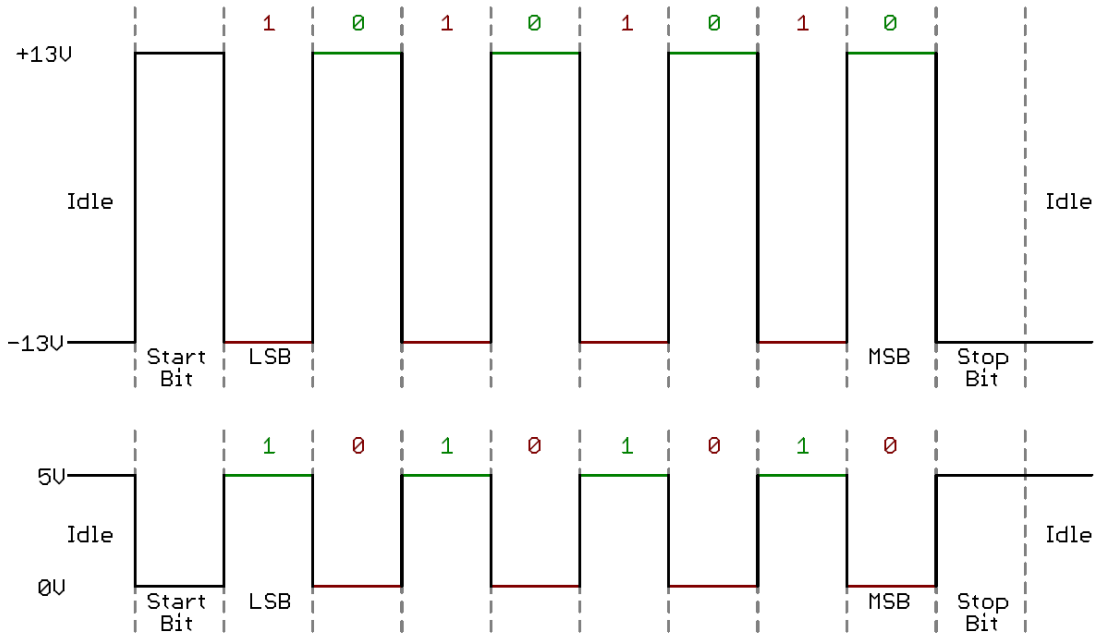


Figure 10: Timing Diagram of RS-232 (top) and TTL Communication Protocols [23]

To address these logic-level issues, an external RS-232 to TTL converter board was needed to allow the rangefinder to 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 11 shows the RS-232 to TTL converter attached to the RS-232 breakout board.

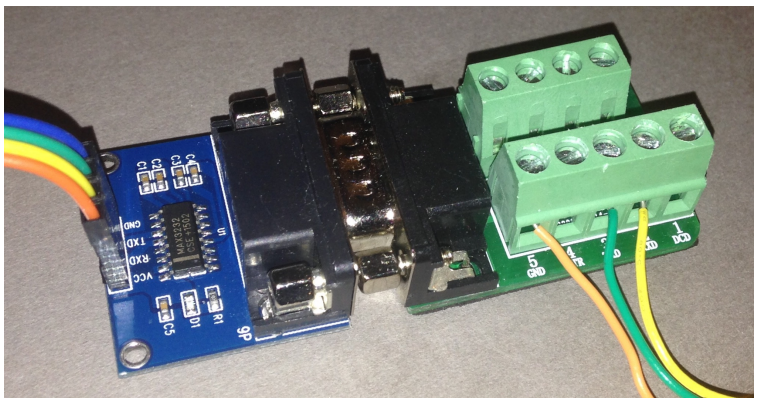


Figure 11: RS-232 to TTL Converter with RS-232 Breakout Board

Although the ZedBoard's Pmod connectors were sufficient for UART communication with

the URG-04LX, the power specifications were not compatible; the Pmod connectors output 3.3V but the rangefinder required 5V [6, 15]. Thus, the rangefinder was powered externally by a lab bench power supply.

4.1.3 Commands

The rangefinder defaulted to a communication speed of 19.2 kbps (19200 baud) and recognized four different commands: the version command, the communication speed setting command, the laser illumination command, and the distance data acquisition command [16]. The version command and the communication speed setting commands were both not used for the purpose of this project. The laser illumination command was used as a debugging tool to turn the laser on and off. The laser's status was displayed using a status LED on the rangefinder, so turning the laser on and off allowed for simple visual confirmation of successful communication. The distance data acquisition command was the primary command used in this project, as it was used for gathering new distance data from the rangefinder.

The distance data acquisition command consisted of five different pieces that controlled 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 was the step of the area from where the data reading started, and the end point was where the data reading stopped. The data reading started at the start point and traversed counterclockwise until the end point. Figure 12 shows a top-down view of the rangefinder's field of vision with the steps labeled.

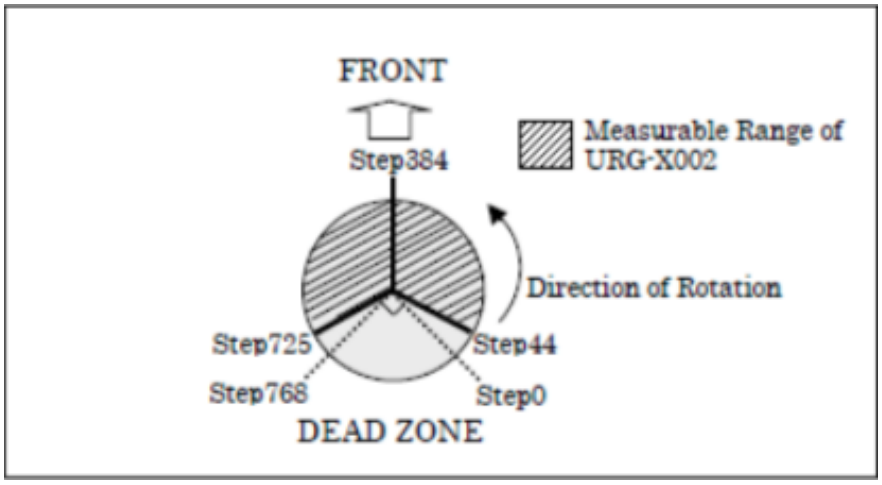


Figure 12: Top-Down View of Rangefinder Field of Vision [16]

For this project, the beginning point was set to '000' and the end point to '768' to obtain the device's maximum coverage of 270°.

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

Putting all of these pieces together, the data acquisition command 'G00076801\n' was obtained and transmitted from the ZedBoard to the rangefinder to request one scan of data. With the rangefinder's operation understood, an interface was created for connecting the ZedBoard to the rangefinder.

4.2 Rangefinder Interface

As discussed in the previous section, the rangefinder was connected to the ZedBoard via a Pmod connector so that UART communication was possible. Since a UART controller was needed to drive communication with the rangefinder, the Zynq7 Processing System was implemented via Xilinx's Zynq7 Processing System Intellectual Property (IP) core.

4.2.1 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) and allows access to both on-chip and external memory interfaces, PL clocks, many I/O peripherals, and even extended I/O peripherals [34]. Despite all of this functionality the processing system was easy to customize and featured a simple user interface. The user interface was used to change the processing system's activated features. Figure 13 shows the processing system customization window.

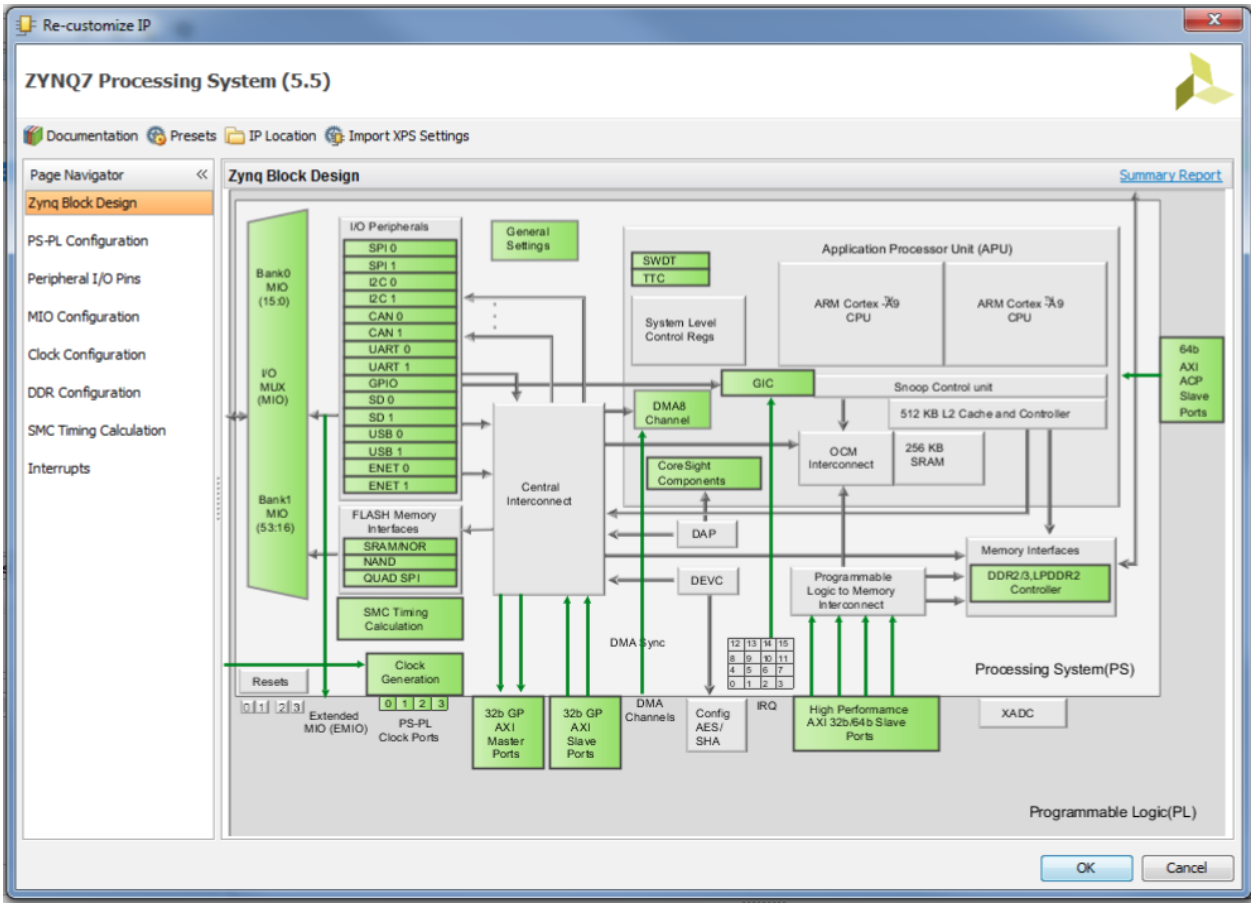


Figure 13: Zynq7 Processing System Customization Window [34]

There were two options for UART: UART0 and UART1. The functionality of UART0 and UART1 were nearly identical, except that UART1 had the capability of being routed

to the ZedBoard's USB UART port, which was not compatible with the rangefinder [6]. So, UART0 was arbitrarily chosen and the signals were routed 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 the rangefinder's default communication speed of 19200 baud [16]. This was done in the processing system's customization window under PS-PL Configuration on the sidebar in Figure 13. Figure 14 shows the PS-PL Configuration window.

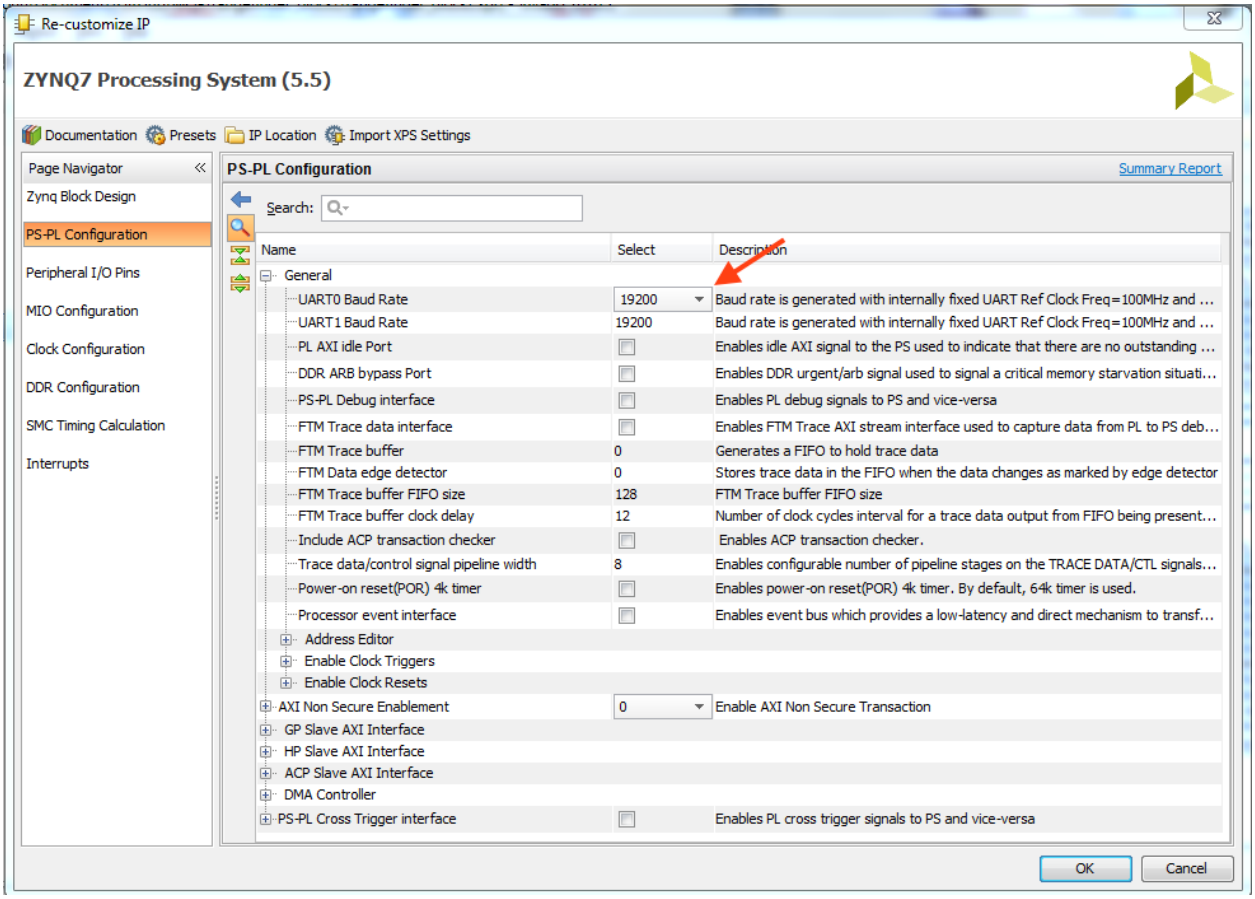


Figure 14: Zynq7 Processing System PS-PL Configuration Window

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

4.2.2 Designing with the Xilinx SDK

The Programmable Software (PS) was coded in the Xilinx SDK. To launch the SDK, the hardware design was exported so that the PS had a platform to be coded on. This was done by generating a bitstream. The bitstream compiled all the project's hardware customization into a *.bit* file which was 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 15.

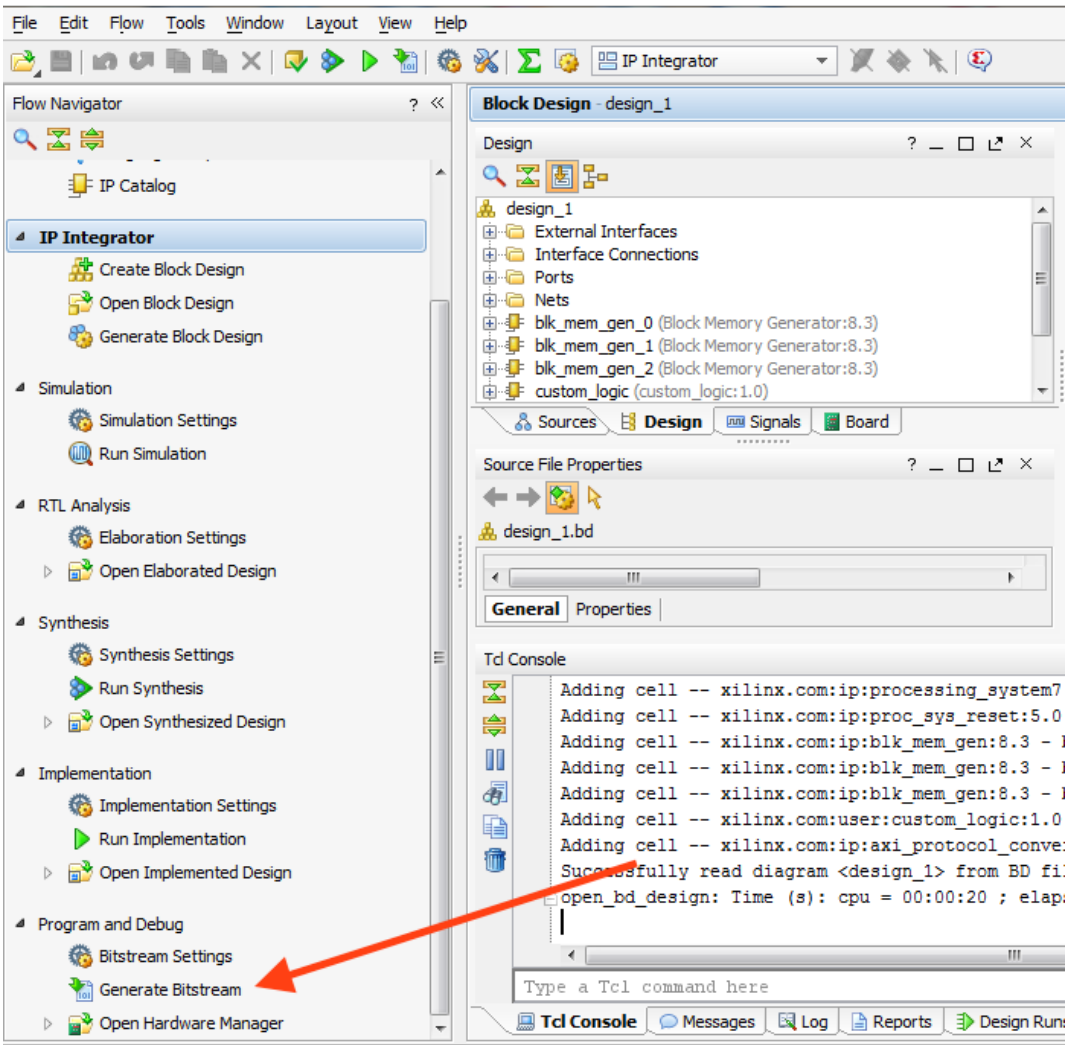


Figure 15: Generating a Bitstream in Vivado

Once the bitstream was generated, the design was exported to the SDK. 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 launched, there was a hardware platform project in the Project Explorer tab. This file contained the hardware platform that was exported from Vivado and was used to program the FPGA on the ZedBoard. To begin programming the PS, an Application Project was created with the hardware platform by choosing File → New → Application Project, entering a project name, and choosing Next. Note that the hardware platform exported from Vivado was selected in the Hardware Platform and was 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 16. The programmable software was edited in the project's source code file.

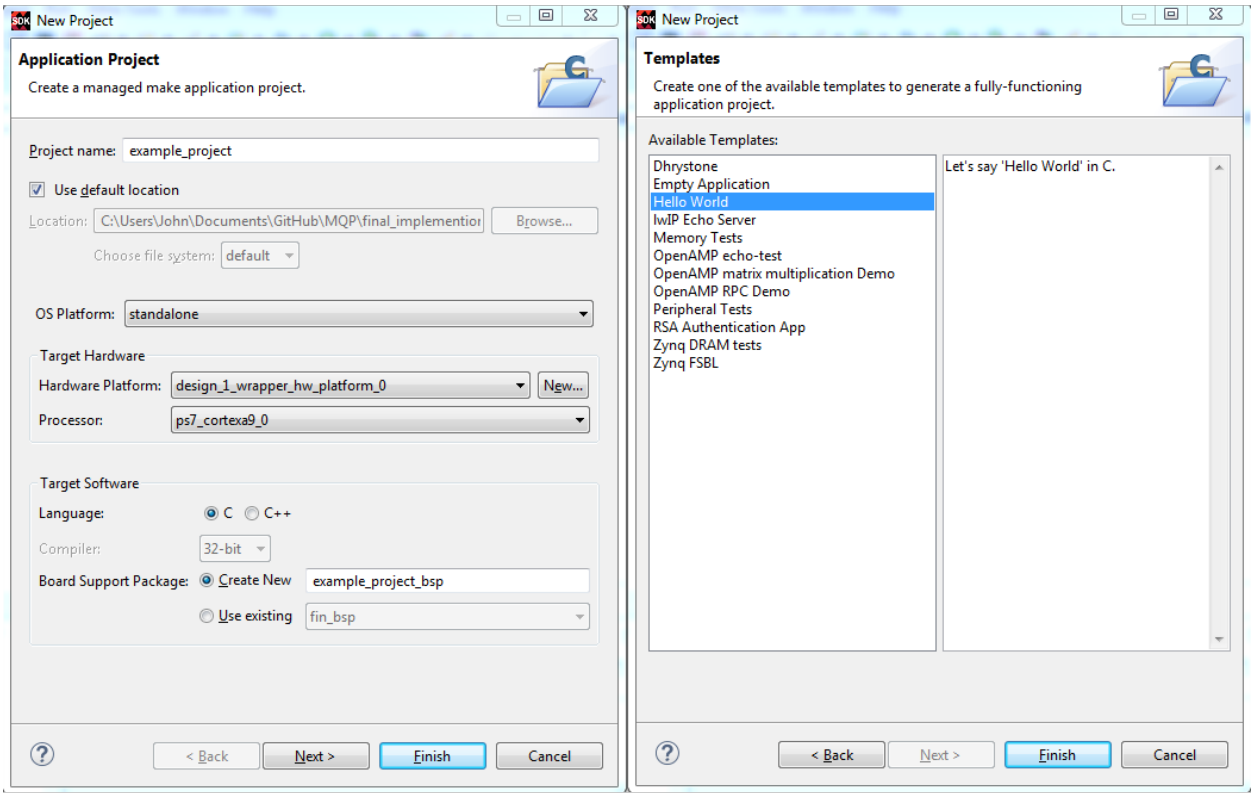


Figure 16: Creating a New Application Project in the Xilinx SDK

To upload the design to the ZedBoard, the FPGA fabric of the SoC was programmed

first in order to configure the PL with the hardware platform. This was 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 then choosing Run As → Launch on Hardware (GDB).

With an understanding of the rangefinder's interface, the data processing strategy was implemented. This will be explored in the following section.

4.3 Rangefinder Data Processing

The rangefinder's implementation into the system had two distinct pieces: communication and data processing. Since the ZedBoard SoC contains both an ARM processor and FPGA fabric, the design was optimized by utilizing both parts of the SoC for their unique advantages. As mentioned in Chapter 3, Xilinx offers a pre-configured UART peripheral driver through the ARM processor. As such, the Programmable Software (PS) was responsible for all of the communication with the rangefinder, in addition to pre-processing the data before it was written to the Programmable Logic (PL), where the FPGA was used to parallelize the data processing, block memory manipulation, and output VGA logic.

4.3.1 Programmable Software

To begin the data transaction, the distance data acquisition command was transmitted by the PS when it received signals from the PL. Once the command was sent, there were two different pieces of information that the software captured: the distance data and the step count. The rangefinder transmitted the distance data half a data point (one character) at a time. The first character was stored in a buffer until the second character was received. Once the second character was received, the distance data buffer was updated to hold both characters, the step count was incremented, and then both were written to the memory register connected to the PL, described further in Section 4.4.

4.3.2 Programmable Logic

When the PL receives the distance data and step count, the data begins its decoding process since the rangefinder encodes each data point before transmitting it. The decoded data point was expressed by 12 bits, in order to cover the rangefinder's maximum coverage distance of 4095 millimeters.[1] The rangefinder encoded the data by separating the 12-bit data into two 6-bit pieces, and then adding 30_{16} to each. The resultant data point was comprised of two ASCII characters [4]. The decoding process which took place in the PL was the inverse of encoding, where 30_{16} was subtracted from each character and then they were merged together MSB first[16].

Since the rangefinder provided the distance away from an object and the angle at which it was detected, the data was essentially expressed in the polar coordinate system [31]. In order to output the data on a VGA screen, the data was converted from polar to rectangular coordinates. This was accomplished by using the step number, since it corresponded to an angle around the rangefinder's detection circle, as shown in Figure 12. Note the angular rotation per step was calculated by using Equation 1. Accordingly, each step around the rangefinder's field of view corresponded to a change of 0.3515625°.

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

The transformation from polar to rectangular coordinates required basic trigonometry. 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, a LUT with a multiplier was implemented instead of a CORDIC function. The values in the LUT were used to extract the horizontal and vertical components from the polar coordinate. The LUT only held 256 values, which corresponded to the amount of rangefinder steps in one quadrant of the

[1] 2^{12} - 1 = 4095

rangefinder's detection circle. Each step value corresponded to two addresses in the LUT: one scale factor for the detected object's horizontal distance away from the device, and one scale factor for the object's vertical distance away from the device

The LUT was configured 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 the rangefinder's detection circle. Note that multiplying by 4096 equates to a 12-bit left shift, and was used to decrease error due to rounding in later data manipulation.

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

This coordinate-axis transformation required both scale factors from the LUT at the same time, but it was only possible to access one address in the BRAM at a time. To avoid read conflicts, the 256-address LUT was split into two 129-address LUTs, where one LUT corresponded to $0° \leq \theta \leq 45°$, and the other corresponded to $45° \leq \theta \leq 90°$.[2] The code for the coefficient files can be found in Appendix D.iii.

Once the LUT was customized, the BRAM IP Wizard window specifies the BRAM's latency. In this case, there was a latency of 2 clock cycles between the addresses being calculated and the data from the LUT being valid. Once the LUT data was valid, each was multiplied by the decoded polar coordinate data point by being wired to a Multiplier IP block. This multiplier carried a latency of 4 clock cycles in this case. This transformation converted the data from polar to rectangular coordinates. After this step, the data was localized to the device's location and then right-shifted so that the data was scaled properly and fit onto to a VGA screen with a 640x480 pixel resolution. After this step, each data point's x- and y-location accurately reflected the distance data localized to the device. The rangefinder's data processing module is found in Appendix D.ii.

[2] 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.

With proper x- and y-coordinates, each data point was stored in memory. Another BRAM IP was created for this purpose. Since the VGA resolution is 640x480, the BRAM required 307,200 addresses.[3] This BRAM IP was customized to function in the write-first, dual port configuration. One port was used as a write-only port with the 100 MHz logic clock, and the other was used as a read-only port using the 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.

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

The data was output to the VGA screen from the BRAM module. To control the VGA logic, Digilent's VHDL VGA controller module, found in Appendix D.ii, was implemented. In addition, the ZedBoard's VGA pins were configured in a constraints file (.xdc) to support 12-bit color resolution [6]. Similar to Equation 3, Equation 4 was used to calculate the read address of the VGA BRAM IP.

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

Since the peripheral communication took place in the ARM processor and the data processing in the FPGA fabric, data needed to pass between the two on the SoC. The creation of the PS-PL communication is discussed in the next section.

4.4 PS-PL Communication

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

[3] $640 \times 480 = 307,200$ addresses.

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 made it perfect for PS-PL communication. These AXI busses are integrally involved in most Xilinx IP cores and contain many wires with sophisticated logic. Instead of attempting to interface with AXI by creating these signals, a custom IP AXI Peripheral was created that abstracted away the complications of AXI communication.

4.4.2 Creating Custom IP

For this project, a custom IP module was created to serve as an AXI Peripheral. As an AXI Peripheral, this IP block was able to communicate with any other Xilinx IP blocks that use AXI. For the purpose of this project, the custom IP block uses its AXI bus to communicate with the Zynq7 Processing System. The custom IP was created in Vivado under Tools $\rightarrow$ Create and Package IP. The IP was set up as an AXI Peripheral, with four 32-bit wide memory registers attached to the AXI bus.

Once the bare IP AXI peripheral created, the custom IP's auto-generated files were edited to allow for memory registers in the design's custom logic to be wired to 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 custom logic. The IP was customized for the use of four memory 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 AXI output register data was wired to a register in the custom logic that held the data to be sent to the PS. 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 D.ii. In the PS, a Xilinx built-in memory access function was used to read the data from the AXI output register, shown on line 130 of the programmable software in Appendix D.iv.

To read from the PS, the data stored in the AXI's input register was wired to a register in the custom logic. This was done in the AXI write address *case* statement on line 239 of our edited auto-generated custom IP code, shown in Appendix D.ii. In the PS, a pointer to the memory address of the AXI input register was read from to obtain the data from the custom logic, as seen on line 198 of the programmable software in Appendix D.iv.

In addition, other necessary I/O ports from the custom logic were created in the custom IP's top module. Our custom IP top module is found in Appendix D.ii. Advanced user logic was also implemented within the IP core through modular instantiation.

Once the rangefinder's implementation into the system was complete, the digital compass was interfaced to in order to rotate the rangefinder's data according to the sensor suite's compass heading.

4.5 Digital Compass Operation

A digital compass was implemented into the system to account for the sensor suite's rotation.This was accomplished through the use of a magnetometer. A magnetometer is a digital instrument that measures the direction of a magnetic field at a point in space. Magnetometers are commonly found in Inertial Measurement Units (IMUs).

4.5.1 Selection

This project required a sensitive IMU that was able to provide accurate and repeatable compass directions. Due to the time limitations and budgetary restrictions of this project, the PmodNAV IMU was selected. The PmodNAV provided 10-degree of freedom functionality through its on-chip LSM9DS1 3-axis magnetometer, 3-axis accelerometer, and 3-axis gyroscope, and the LPS25HB barometer [22, 21]. The PmodNAV is shown in Figure 17.

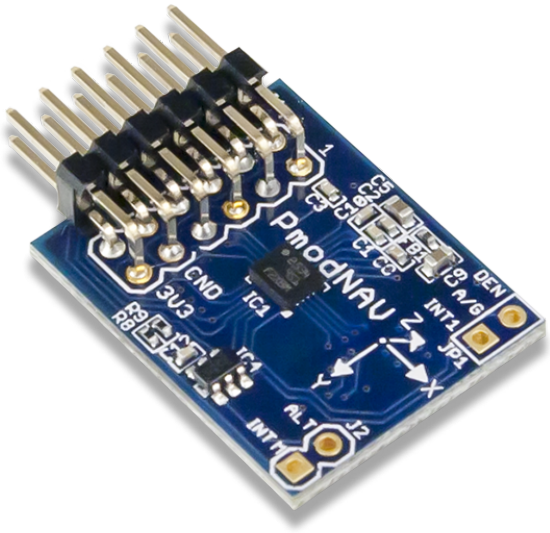


Figure 17: The PmodNAV 10-Axis IMU [10]

The PmodNAV's was easy to integrate into this design, as it was fixed in position connected to the ZedBoard and its communication was directly compatible, requiring no intermediary hardware.

4.5.2 Communication

The PmodNAV supported two communication protocols: Serial Peripheral Interface (SPI) and Inter-Integrated Circuit (I^2C) [22]. However the magnetometer on the LSM9DS1 was not addressable by the I^2C bus, so SPI communication was implemented. The magnetometer's SPI protocol is shown in Figure 18.

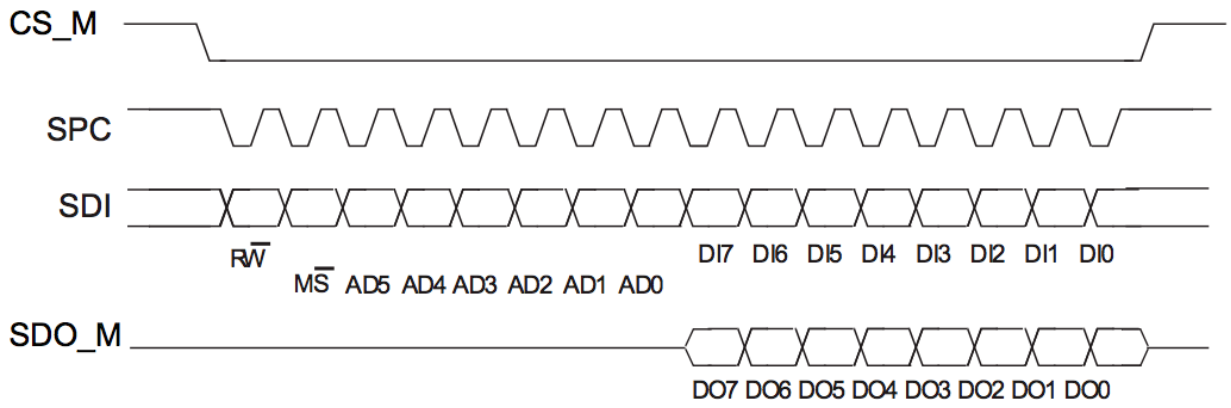


Figure 18: Magnetometer SPI Read and Write Protocol [22]

The CS_M line was the magnetometer active-low chip select. The SPC line is the clock controlled by the master. SDI and SDO_M are the data input and data output lines, respectively. They were driven at the falling edge of SPC and were captured at the rising edge [22].

In an SPI sequence, the first bit sent from the master is the read/write bit, $R\overline{W}$. This bit was set to 1 to read from the compass, and 0 for a write to the compass. For a read sequence, the DO bits represented the data read from the memory address specified by the AD bits. The DI bits were not written to. For a write sequence, the DI bits were the data being written to the memory address, and the DO line was ignored. When the $M\overline{S}$ bit was set to 1 the memory address signified by the address bits $AD5$ down to $AD0$ was auto-incremented, allowing for multiple reads or writes to be completed in the same SPI transaction [22]. Figure 19 shows a multiple-byte SPI read protocol.

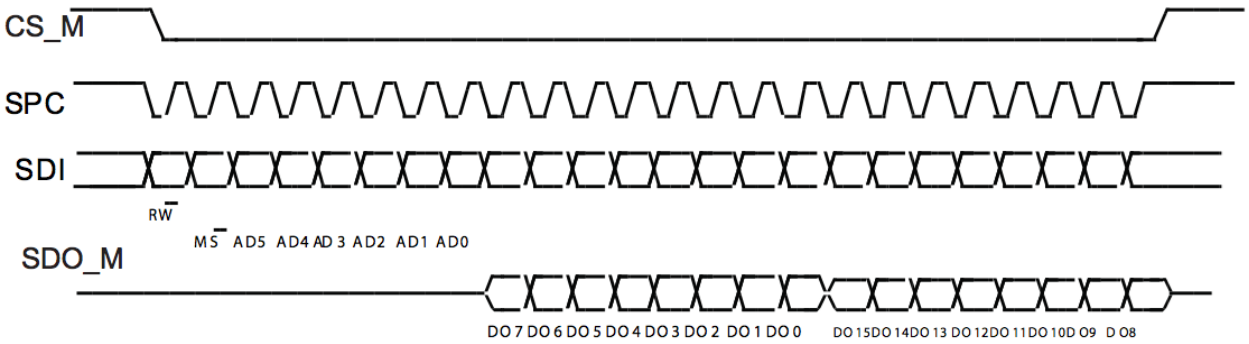


Figure 19: Magnetometer Multiple-Byte SPI Read Protocol [22]

4.5.3 Register Settings

The magnetometer on the LSM9DS1 needed slight adjustments in its register setting in order to operate properly. The adjustments were done by writing to the magnetometer's control registers.

One register that was written to was the magnetometer's control register 1, CTRL_REG_1_M, 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 [22].

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 [22].

Due to time constraints, the magnetometer on the PmodNAV was the only slave that was interfaced to. The accelerometer, gyroscope, and barometer were not used so the above control registers did not need to be changed to allow the other sensors to be read from.

4.5.4 Read Registers

With the magnetometer turned on and its settings properly adjusted, its data registers were read from. As such, the $R\overline{W}$ bit was set to 1. There were two different read sequences used to obtain compass data.

One sequence that was used was a read from the status register, STATUS_REG_M, which is at address 27_{16}. The status register signifies which of the magnetometer's registers hold data that has not yet been read. This address was read from until the two least significant data bits read 11_2, signifying that new x- and y-axis magnetometer data was available [22]. Once new x- and y-axis data was available, the corresponding data registers were read from.

The next command was used to read the available x- and y-axis magnetometer data. This data came in 16-bit resolution. Due to the SPI transfer protocol shown in Figure 18, 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 were consecutive,

allowing 32 bits of data to be obtained in one cascading read in the format shown in Figure 19. The cascading read was performed 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 [22].

4.6 Compass Implementation

The PmodNAV was connected to the ZedBoard via one of its Pmod connectors. The PmodNAV was controlled by a Pmod connected to the ARM Processor of the SoC so that the compass data was able to be directly combined with the rangefinder's data before it was written to the FPGA. However, only one of the ZedBoard's five Pmod connectors is directly connected to the ARM Processor, and it was already used for communication with the rangefinder. One Pmod connected to the FPGA fabric was re-routed to be controlled by the ARM Processor through capability provided by the Zynq7 Processing System.

4.6.1 Re-Customizing the Zynq7 Processing System

The Zynq7 Processing System was easily re-customized to account for the PmodNAV's specifications. 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) to allow one of the ZedBoard's FPGA-controlled Pmods to be controlled by the ARM Processor. Since both SPI0 and SPI1 support EMIO capability, SPI0 was arbitrarily chosen over SPI1. This process is shown in Figure 20.

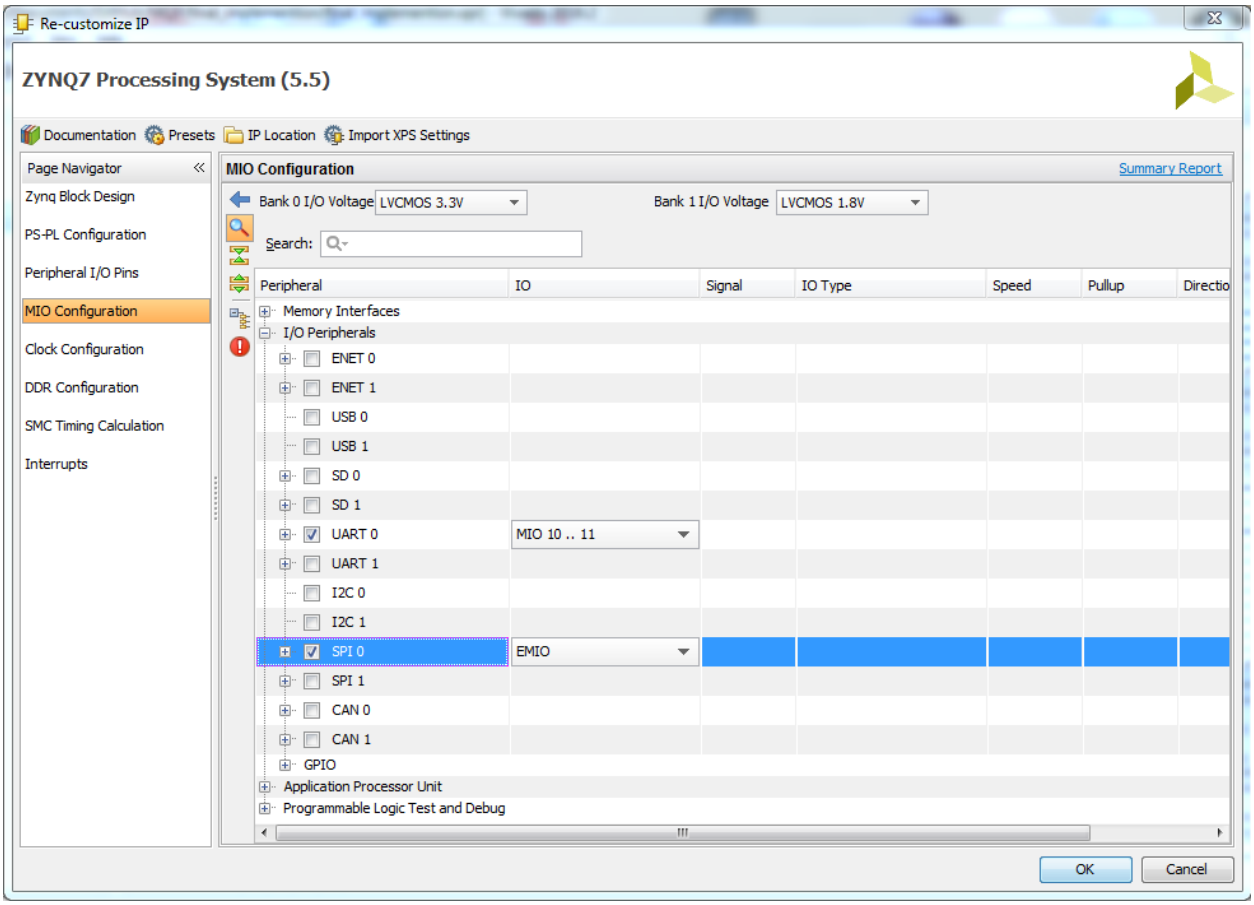


Figure 20: Re-Customizing the Zynq7 Processing System to Add SPI

4.7 Compass Data Processing

Once the Zynq7 Processing System was re-customized to support communication with the PmodNAV, the Xilinx SDK was relaunched. All of the magnetometer's data was processed in the Programmable Software on the ARM Processor to rotate the rangefinder's data according to the sensor suite's compass heading.

4.7.1 Programmable Software

The SPI communication in the SDK was implemented by following example code provided within the SPIPS driver in 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 magnetometer initialization and communication from Section 4.5.3 was implemented. In the programmable software, the axis data was read into an unsigned 8-bit variable. The data points were rearranged and then stored into a buffer of type *int* for each axis. The magnetometer's axis data was signed and expressed in Two's Complement format [22].[4] Combining a data point 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 each data point's sign bit. If each data point was greater than or equal to 8000_{16} then the sign bit must be '1', indicating a negative number. If necessary, the data was turned negative by subtracting $FFFF_{16}$ and adding 1.

To perform the necessary complex math on the data, the header file *math.h* was linked and 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 21. In addition, *math.h* still was included by using *#include* in the project's source code file.

[4] 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 [13].

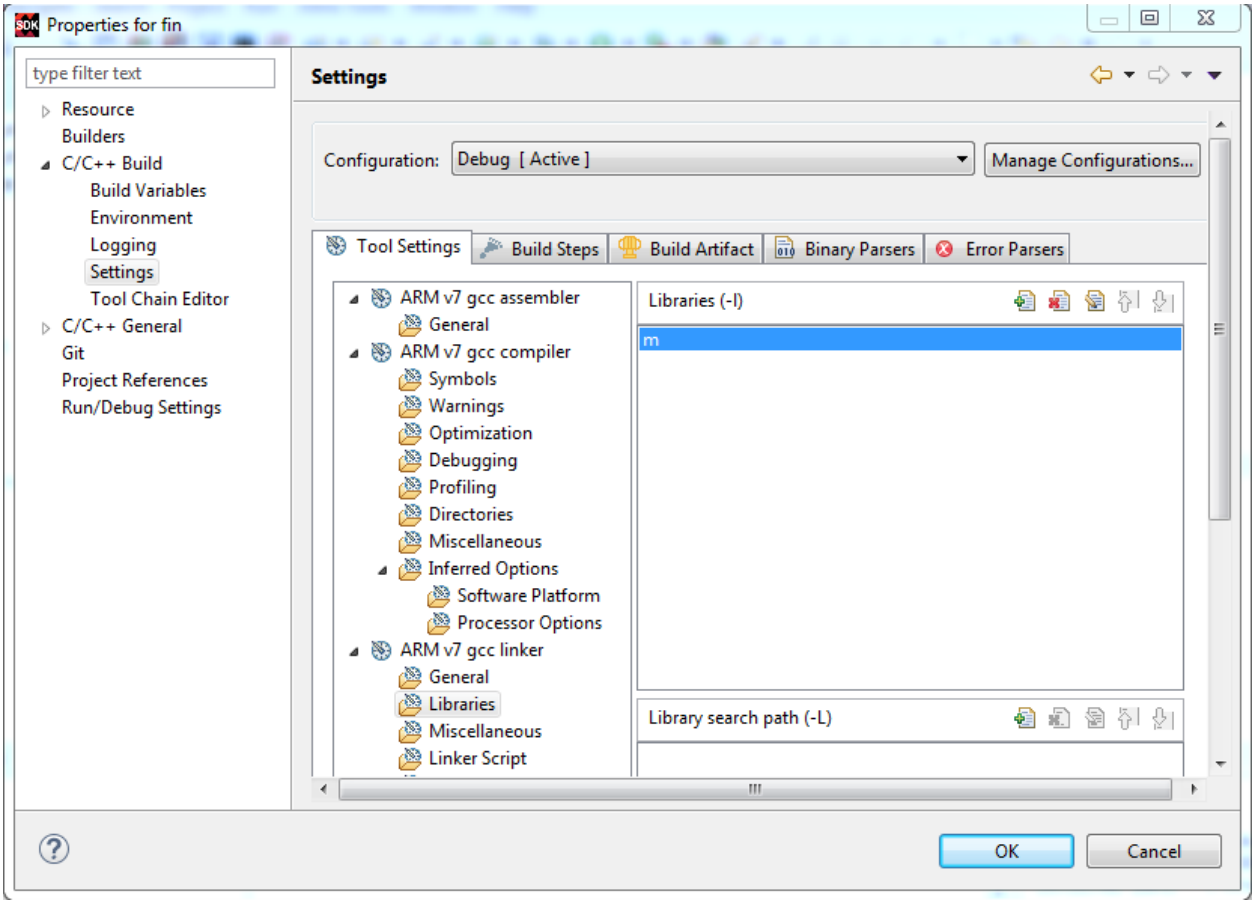


Figure 21: Linking *math.h* into the Application Project

The complex math was used to convert the magnetometer's axis data, in terms of milli-gauss per bit, to a compass heading in degrees. The equation that was used for this transformation is shown in Equation 5. Note that the arctangent function from *math.h* outputs data in radians, so the resultant angle was multiplied by $\frac{180}{\pi}$ to convert it to degrees.

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

To convert the compass heading to a rangefinder step offset, Equation 6 was used. This calculation was possible because there were a total of 768 rangefinder steps around the rangefinder's 270° field of vision [16].

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

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 sensor suite faced due North the step offset equated to 0, so the rangefinder's data was not rotated. When the ZedBoard faced due South the step offset equated to 512, which rotated 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 performed to determine how the chosen module was operated.

4.8.1 Camera Selection

Due to our limited project budget of $250, we focused on finding camera modules that were low-cost and simple to communicate with. This ruled out many low-cost camera modules that relied 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 based on several categories, 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 was 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. Global shutter cameras contain specialized circuitry for acquiring image data from all pixels simultaneously, thus reducing motion-based artifacts and making them ideal for taking images in a sensor suite that is susceptible to motion. The MT9V034 also used a parallel data interface and relied on an external clock and shutter trigger, which made 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 began by reverse engineering the camera module breakout boards purchased. The MT9V034 camera breakouts used were purchased through Leopard Imaging Inc. Although these camera module breakout boards were 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 were ideal for our application [18, 27]. Once the schematics of each camera module breakout were known, a basic control interface for each camera was then designed.

4.8.2 Camera Signaling

By default, the MT9V034 camera module continuously gathered image data at 60Hz while supplied with an external clock signal and output enable signal [27]. In this operating mode, several output signals from the camera module were used to transmit image data. Each image, or frame, was broken up into individual "lines" which corresponded to a line of pixels that stretched the width of the frame. Since our camera module captured images at 752x480 pixel resolution, one frame contained 480 lines of 752 pixels each. The camera modules broke up image data by frame and line, and camera data pins FRAME_VALID and LINE_VALID were toggled to indicate the transmission of a frame or line. The timing diagram shown in Figure 22 shows the operation of these pins while transmitting an image.

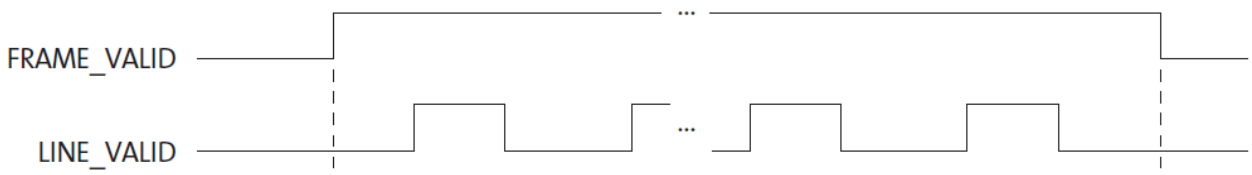


Figure 22: Frame and Line Valid [27]

Since the MT9V034 camera module transmitted pixel data in parallel and each pixel contained 10 bits of resolution, 10 pins were used to transmit pixel values. Pixel data was transmitted in correspondence with LINE_VALID and output clock signal PIXCLK. When LINE_VALID was asserted, the pixel data pins were updated with values corresponding to pixels 0-751 of the given line. Values for each pixel were written out on the falling edge of the camera's PIXCLK pin, which allowed for pixel values to be read on each rising PIXCLK edge. A full LINE_VALID data transmission sequence therefore contained 752 PIXCLK cycles, which corresponded to the 752 pixels that made up the given line. A timing diagram of this data transmission scheme is shown in Figure 23.

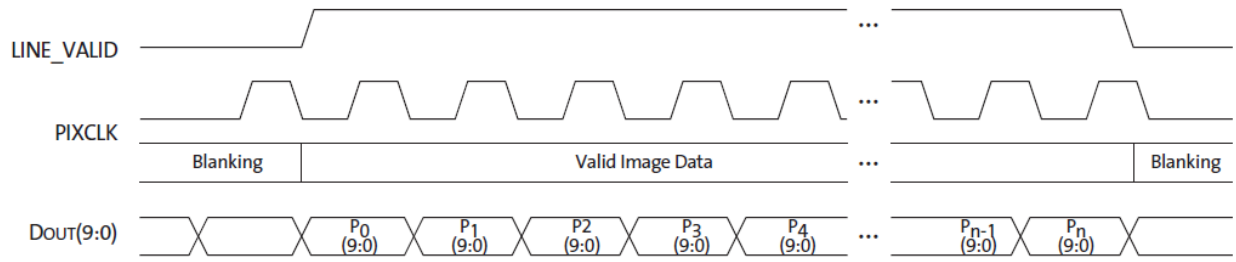


Figure 23: Line Data Transfer [27]

The default camera data transmission scheme was also examined using an oscilloscope, as shown in Figure 24, with channels 1-4 corresponding to camera PCLK, FRAME_VALID, LINE_VALID, and Data[0], respectively. In the case of Figure 24, the camera was initially powered off, resulting in an inactive PCLK signal during the beginning of the recording.

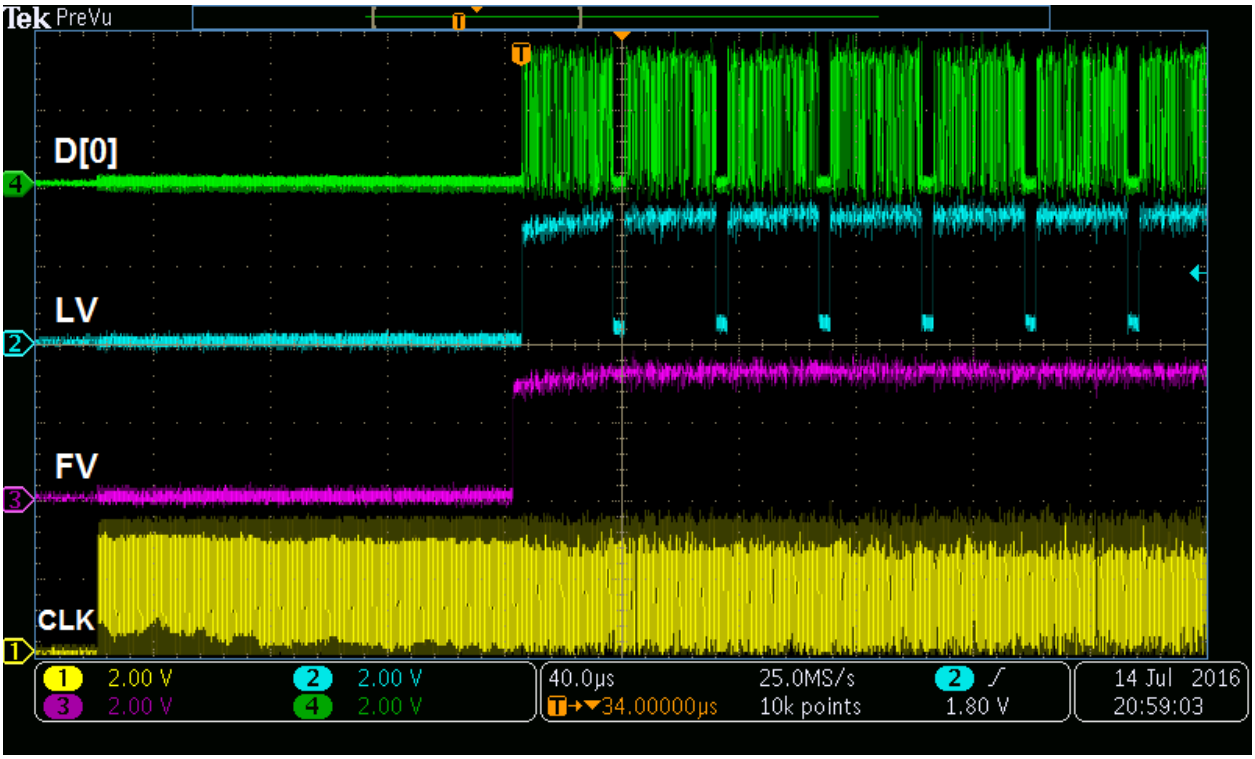


Figure 24: Camera Data Transfer

4.8.3 I²C Control

The MT9V034 Camera module's mode of operation was 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. The I^2C interface used contained a Serial Data Line (SDA) and Serial Clock Line (SCL) that were normally pulled to 5V. When one connected I^2C device wished to communicate with another, it pulled the SDA line low while leaving the SCL line high. The master device then began clocking the SCL line, and SDA was used to transfer 7 bits representing the address of the desired slave device, along with an 8th bit that represented whether a read or write operation was being performed. An example of this transfer is shown in Figure 25. After this initial transfer, a second 8 bit sequence representing a specific register within the slave device was then transmitted.

As an example, if the master device wished to write to slave device 0x40 at register 0x00, it transmitted 0x41 (address 0x40 and WRITE), followed by 0x00. If the slave device received this transmission, it acknowledged by pulling the SDA line low. If this acknowledgement was successfully delivered, the master then transmitted the value that it wished to write to the given slave address and register. If the operation was a read rather than a write, the slave then transmitted a value back to the master.

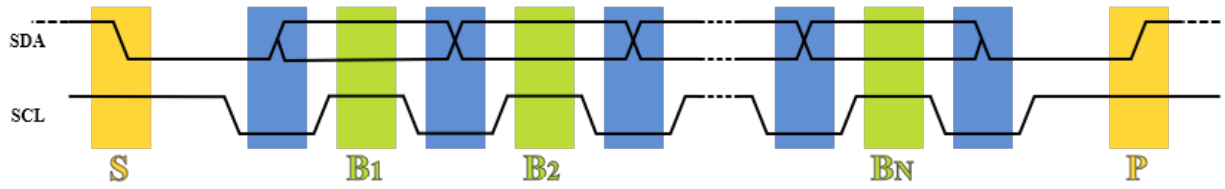


Figure 25: Example I^2C Data Transfer

Based on the LIVM34LP camera board schematic, each breakout board was configured so that its camera module was accessible at I^2C address 0x58 [18, 27]. Note that since both cameras came configured with the same I^2C bus address, a pullup resistor needed to be added to one of the cameras I^2C address lines so that both were individually accessible on a shared bus.

4.8.4 Image Buffering

Since each camera image contained 752x480 pixels with 10 bits of resolution per pixel, a full camera image consumed 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 was 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 needed to run at exactly the same clock speed. The same amount of vertical and horizontal blanking was also required to display each pixel in its correct location. If the image was transferred to a computer, the act of packaging the information so that it was interpreted by said computer would have placed severe limitations on the speed of the system. A proper solution to these timing issues was to buffer image data between the camera and the desired output source, which 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 seemed like a relatively small image size, creating a buffer object large enough for storing said image would have consumed an extremely large amount of FPGA resources. For reference, the Nexys3 FPGA evaluation board initially used for camera testing contained only 18kB of onboard Block RAM, and was not able to buffer an image of this size without the use of external memory[5]. 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 was created

[5]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 was able to connect directly to the camera module data output lines [5]. The AL422B FIFO module contained 3M-bits of RAM that were written to and read from in parallel, and supported separate input and output clock speeds between 1-50MHz [5]. This meant that the camera module was able to write pixel data to the FIFO as long as it operated at a speed between 1 and 50MHz, and the FPGA was able to independently read from the FIFO at any speed within the same range. Note that since this FIFO supported only 8-bits of parallel data input and output, the lowest two bits of camera pixel data were truncated. This was not a major issue, since the truncation corresponded to a 4/1024 reduction in the range of values that each pixel mapped to.

4.9 Disparity Algorithm

Some important properties of the stereo camera setup used in this project were taken advantage of in order to extract 3D depth information from 2D image data. Since both cameras captured imagery of the same scene from slightly different vantage points, depth information on the scene was extracted by calculating the pixel offsets, or disparity, between the same object's relative location in each image. Given this pixel offset, the distance from the camera pair to a given object was determined using simple geometry based on the focal length and baseline of the stereo camera pair. Each camera needed to have the same focal length, or distance from the image sensor to the lens of the camera. The baseline, or distance between image sensors of the stereo camera pair was equivalent to 63mm. Note that this number was chosen to reflect the average distance between a pair of human eyes [8].

4.9.1 Image Rectification

One simple method for 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 [8]. An example of shared epipolar lines between camera imagery is shown in Figure 26 below. Although an ideal stereo camera setup contains shared epipolar lines between camera images, raw image data from seemingly identical cameras will contain slight differences in object location based on the physical position of the camera modules, as well as minor differences in the lenses of each camera. Both input images are normally adjusted to share the same epipolar lines through a post-processing step known as image rectification [8].

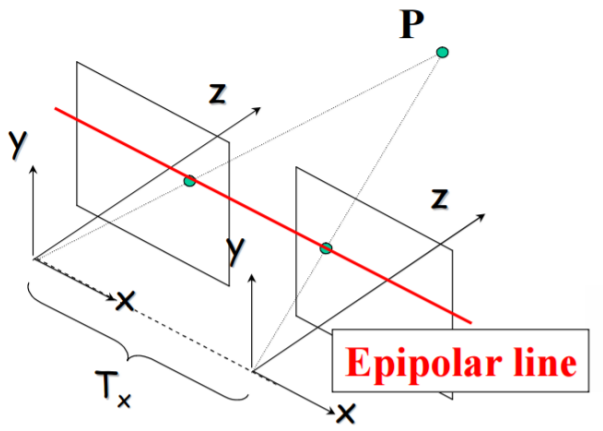


Figure 26: Horizontal Epipolar Lines [8]

A pictorial representation of the process of stereo image rectification is shown in Figure 27 below [24]. This specific rectification example was achieved using a 3x3 matrix coordinate transform based on parameters obtained from the external calibration process.

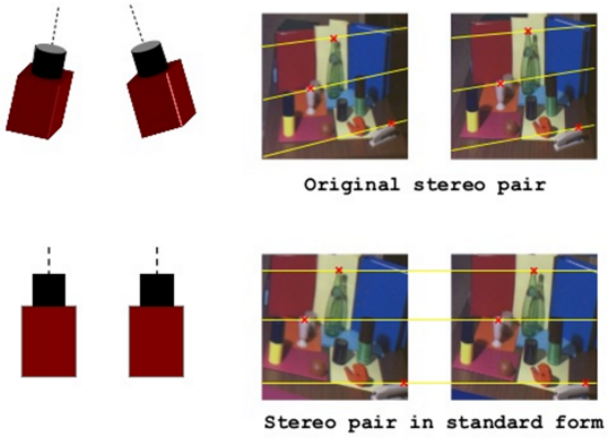


Figure 27: Stereo Image Rectification [24]

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 was 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 matched 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 28, 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 given example, the template and search blocks are relatively different, resulting in a high SAD value.

Figure 28: Sum of Absolute Differences [25]

Since the disparity algorithm used in this implementation calculated the sum of absolute differences for multiple search blocks, the resulting SAD values for each search block were then 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 indicated higher similarity between the template and search blocks. This comparison is demonstrated in Figure 29 below. In the case of Figure 29, higher match score values for each search block indicate lower SAD values.

In the project implementation, the SAD at multiple search points was 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 were stored in a vector. This pixel offset was equivalent to the disparity value for a given template and search block. The disparity d at a given point was then 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.

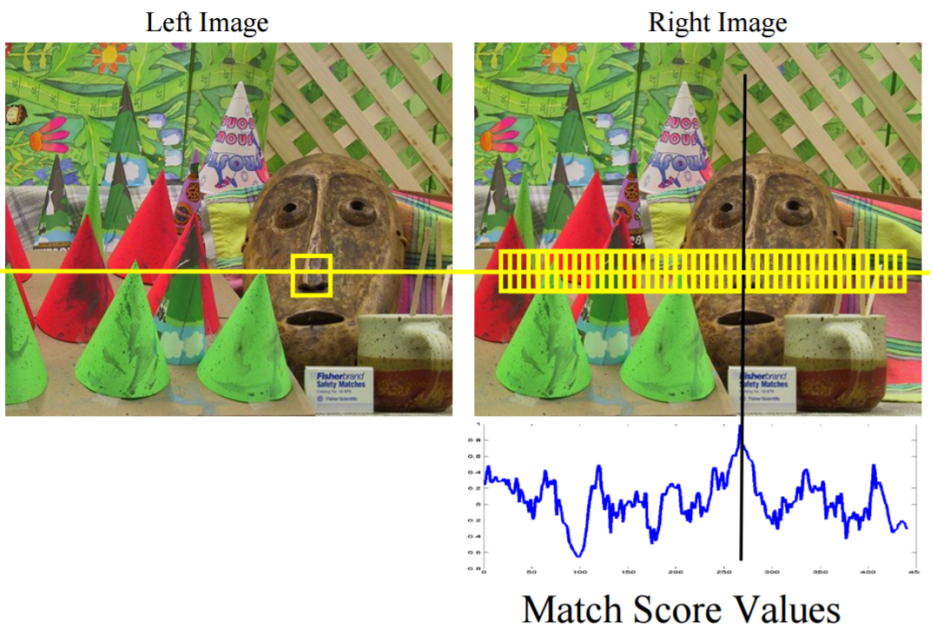


Figure 29: Block Matching Overview [8]

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

Pixel coloration values in a disparity image were based on the distance calculation shown in Equation 9, where each pixel was referenced to the disparity at a given template block's location. As an example, a disparity image created from a given pair of test images is shown in Figure 30 below.

Figure 30: Disparity Algorithm Output

4.10 Combined Implementation

The following sections describe the integration of each individual sensor's data into the final, combined implementation.

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 combined data stream relied on a moving average Finite Impulse Response (FIR) filter across a horizontal line of depth information from the disparity algorithm output. Note that although 2D rangefinder data was organized using a polar coordinate scheme, the output buffer used for displaying rangefinder data via VGA contained the same data in a Cartesian format. Cartesian rangefinder data was easily combined with averaged disparity depth information at the output stage, where both sensors' data was 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 needed to be taken into account. Since each camera had an approximate 55° field of view, and camera imagery was 752 pixels wide, the stereo camera interface had a deg:pixel ratio of $\frac{752}{55} = 13.67\frac{px}{deg}$. In contrast, output data from the rangefinder was divided into 768 steps over a 270° field of view. In order to correlate disparity and rangefinder data, the averaged disparity depth line was converted an equivalent number of rangefinder "steps" worth of data. The conversion factor for pixels of disparity depth to "steps" was calculated as shown in 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 needed to be 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 Compass Data

Device orientation was calculated using compass data from the IMU's magnetometer. This compass data was used to offset the rangefinder's step count in the programmable software. Changing the step count changed the start point location of the rangefinder sweep, thus rotating the distance data around the device. In turn, this changed the direction of the rangefinder's 240° "field of vision". For example, a compass reading corresponding to due West was a rotation of 90° counterclockwise from due North. By Equation 11, 90° equated to 256 rangefinder steps. So, the rangefinder's data essentially began at step 256 and ended at step 1024 or 0,[6] seen in Figure 12.

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

In this chapter, the behavior of each sensor was explored in depth, and an overall system designed was created. Once this design was completed, each sensor's behavior was thoroughly tested before integration.

[6] step 768 + 256 step offset = 1024. Since there were 1024 rangefinder steps in a circle, step 1024 was the same as step 0.

5 Testing and Results

After learning how to interface with the chosen sensors, initial test implementations were then created for each individual sensor. Many of these implementations were designed for ease of integration into the finalized design, as detailed in the following sections.

5.1 Rangefinder Testing

The URG-04LX scanning laser rangefinder required an external 5V power source connection. It was connected to a lab bench power supply for all of the sensor suite's testing. Once the power supply was turned on the rangefinder's status LED illuminated, signifying that it was ready and waiting to be communicated with.

5.1.1 Testing via the Data Viewing Tool

The URG-04LX has a useful data viewing tool by Hokuyo Automatic Co. that was used to view, record, and replay the device's data. To use this tool, the device was connected to a computer via its USB port and the tool was launched. Figure 31 below shows a screen capture of the application recording data captured by the rangefinder.

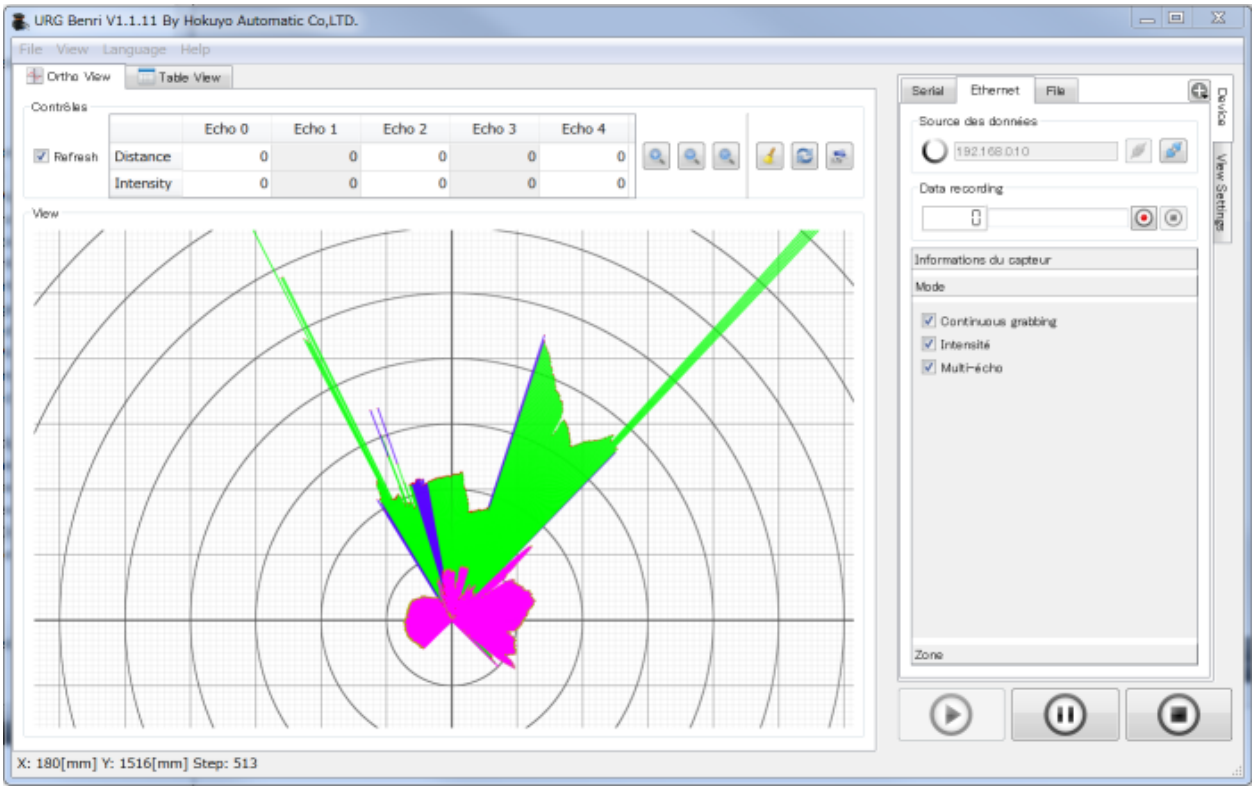


Figure 31: Screen Capture of the URG-04LX Data Viewing Tool [17]

Note that the start point of 0, end point of 768, and dead zone aligned to that shown in Figure 12. For this project the data viewing tool was used to verify the project's 2D map output.

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 32 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.

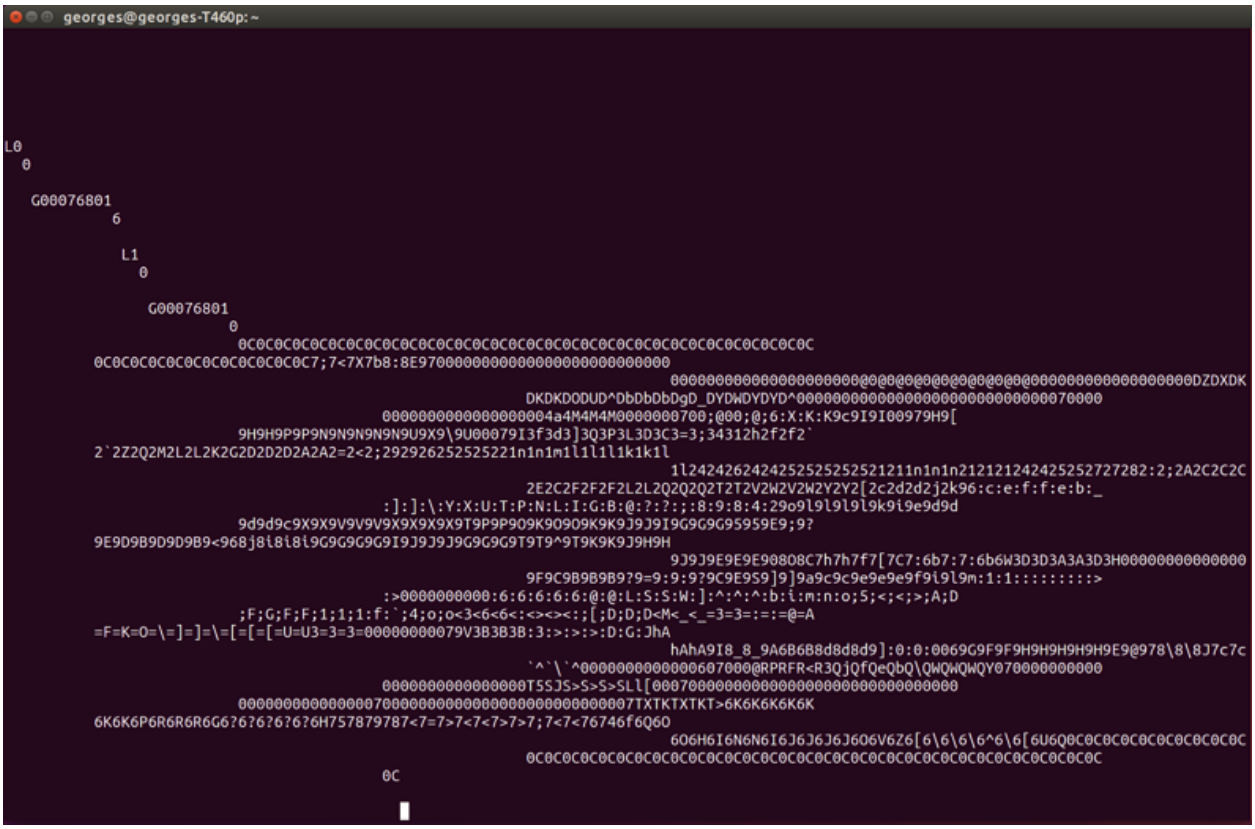


Figure 32: Rangefinder Communication Test via PuTTy

The figure above shows four communication sequences used as a test. The first was the laser illumination command 'L0\n'. 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 was the data acquisition command 'G00076801\n'. The rangefinder responded with '6', indicating an error code. This specific error code was a result of the laser being off when new data was requested [16]. The third command shown was 'L1\n', the laser illumination command again, which turned the laser back on. The rangefinder's response was '0' again, indicating success. The last command shown was the data acquisition command again. The rangefinder's response begins with the echo and then '0', indicating success, followed by the distance data block. The data block consisted of 768 data points, specified by the data acquisition command. By communicating with the rangefinder via PuTTy, the rangefinder's behavior was confirmed to be functional. 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, this mode of communication was continued. 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 [30]. Through USB OTG, devices choose to act as either a peripheral or a host. For the purpose of this project, the ZedBoard was expected to act as the host and initiate communication with the rangefinder. USB OTG was enabled in the Zynq7 Processing System and was controlled by the ARM Processor. The rangefinder's laser illumination command was chosen to be transmitted from the ZedBoard for this communication test. This specific command was chosen because when received, the status LED on the rangefinder blinks until the laser is turned back on. This was 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 a blink of the TX LED. However the rangefinder did not acknowledge the command; its status LED stayed lit signifying the laser stayed on. Due to this failure,[7] using USB OTG was not implemented. Instead the methodology described in Section 4.1.2 was implemented.

[7] 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 [30].

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 the oscillogram in Figure 33. Note that this is a TTL signal.

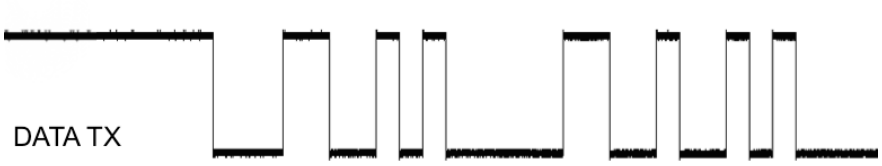


Figure 33: Laser Illumination Command TTL Oscillogram

Since the rangefinder uses RS-232 communication, an RS-232 to TTL converter with an attached breakout board was connected to the ZedBoard. The converter's V_{CC} and GND were connected to the ZedBoard Pmod's respective V_{CC} and GND 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 waveform shown on the oscilloscope. In another attempt, the converter's TX and RX pins were disconnected and swapped, so 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 34 was observed on the oscilloscope. The oscillogram shows a waveform from +6V to -6V, which is a valid RS-232 signal.

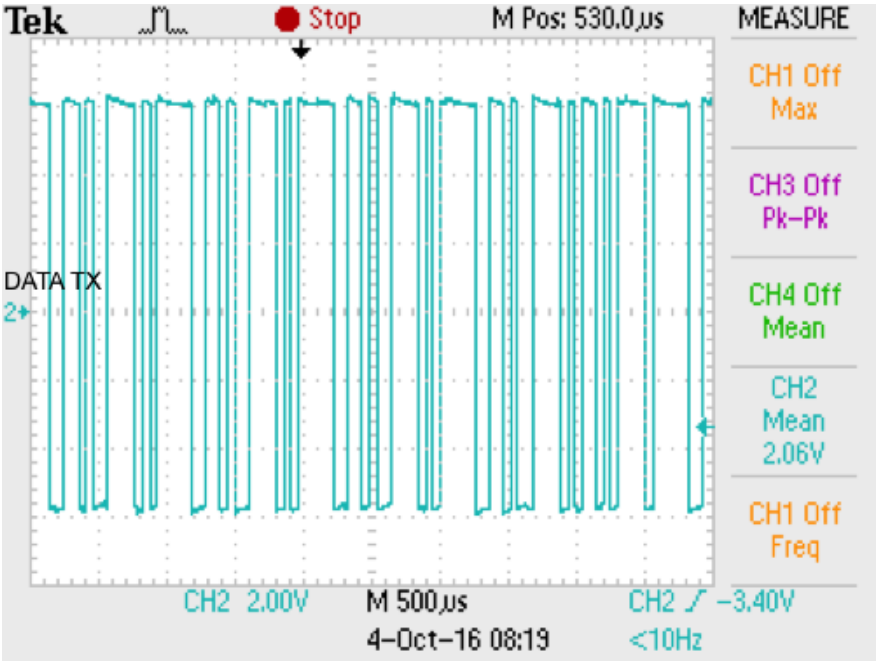


Figure 34: Laser Illumination Command RS-232 Oscillogram

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 that 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 communication routed to USB UART, the TX LED flashes 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*, in place of *slv_reg*0, as seen on line 368 of the custom IP's instantiated file located in Appendix D.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 contained the hardware platform specifications. The base address was the cell with the same name as the custom IP. For this project, the base address was $43C00000_{16}$. Since slv_reg0 was the first of the four designated memory registers, it did not need any address offset from the base address. Reading from the PL was implemented on line 30 of the PS, shown in Appendix D.iv, by using Xilinx's built-in memory access function Xil_In32 to read the data from the memory address that is $baseaddr\_p$.[8] 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 was intended to receive the transmit signal from the PL and then wait 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 instead of the rangefinder. Through the serial console, rangefinder communication was simulated by inputting a test block of rangefinder data. The data was written to the PL one data point at a time by writing to slv_reg1, as shown on line 239 of the custom IP's instantiated file located in Appendix D.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 D.iv. The data written to slv_reg1 was the distance data point combined with 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 test block of rangefinder data sent from the serial console was constant. With data constant across all steps of

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

the rangefinder's field of view, the intended result was 270° of a circle drawn around the rangefinder on the VGA screen. With the VGA module set up and the rangefinder data processing ready to be tested, the ZedBoard was programmed. When BTNR was pushed, an image similar to Figure 35 was observed on the VGA screen with the red dot being the device and the black lines being the rangefinder's distance data.

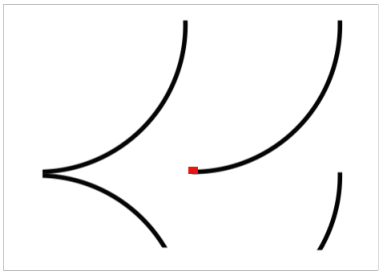


Figure 35: PS to PL Communication Test with Constant Data

Although a circle was not observed, this test confirmed the PS to PL communication was functioning properly. The shape appeared to be four quadrants of a circle, except in the wrong orientation. Since the lines seem semi-circular, the polar-to-rectangular transformation was deemed successful. 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, resulting in Figure 36 being observed on the VGA screen.

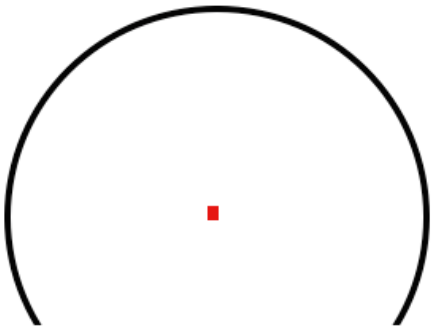


Figure 36: PS to PL Communication with Constant Data and Revised Data Processing

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 37 shows the VGA output. This VGA output shows the rangefinder with a wall directly in front of it.

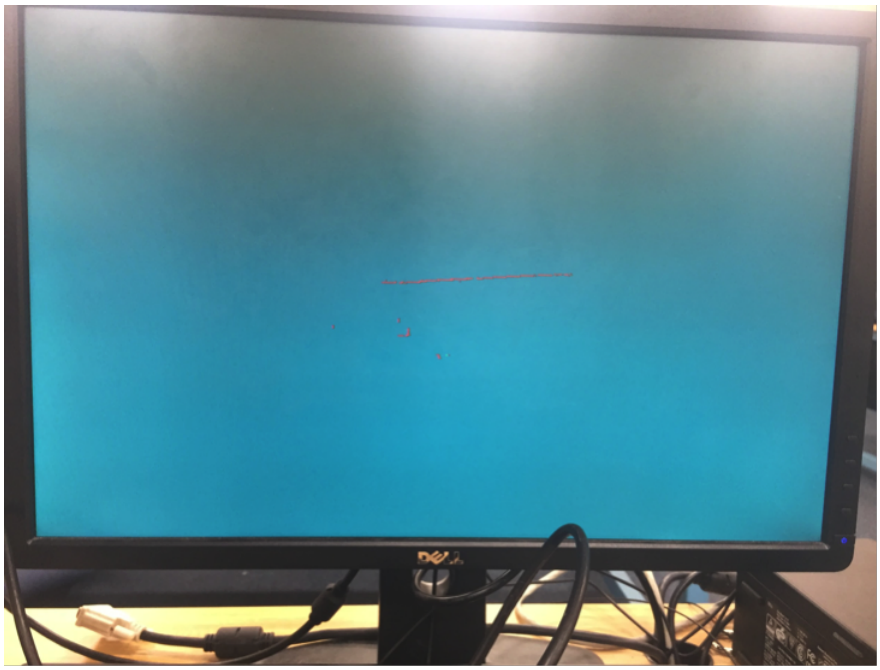


Figure 37: Rangefinder Data Observed on VGA Screen

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 38 were observed on the VGA screen.

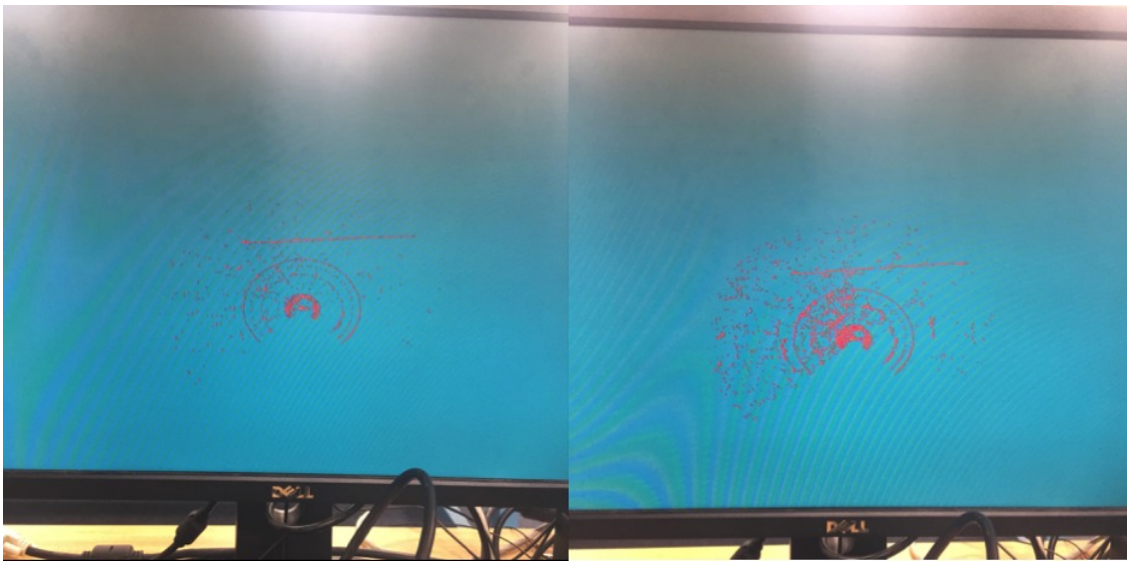
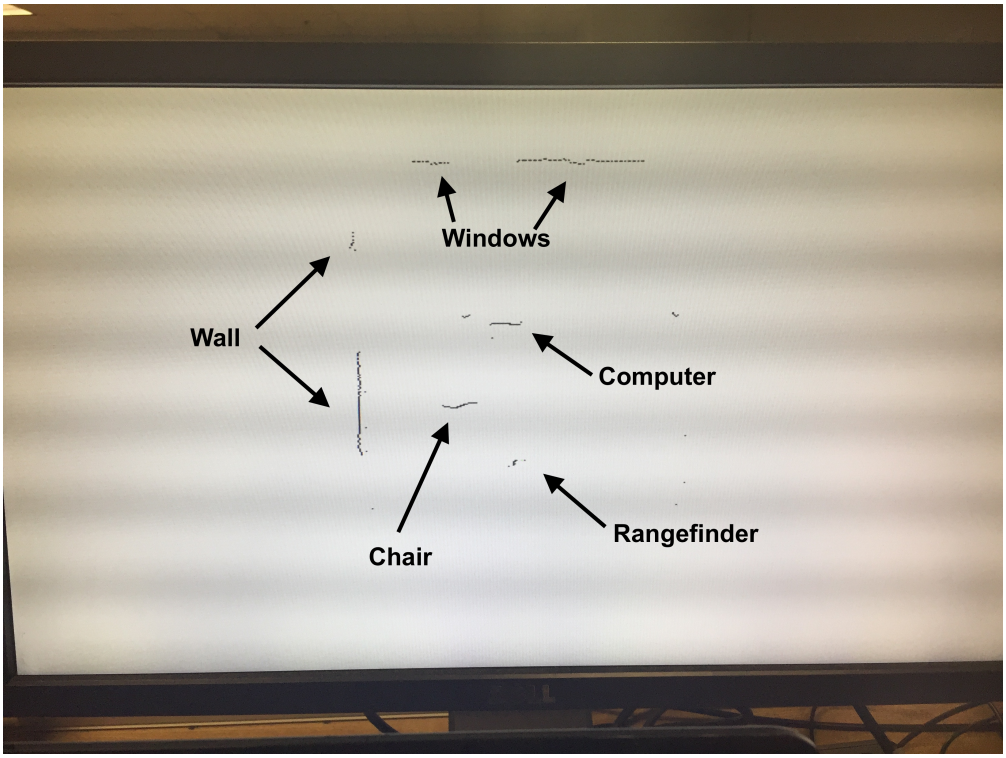


Figure 38: Subsequent Rangefinder Data Observed on VGA Screen

In the SDK the PS was not accounting for enough data points. By design, when the rangefinder receives a command from the ZedBoard it echoes back the command. The PS used this as a test to ensure data accuracy. When 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 rangefinder was tested again. Figure 39 shows the initial data transfer of this test, and the 2D florplan was compared to the objects around it.

(a) Lab at WPI



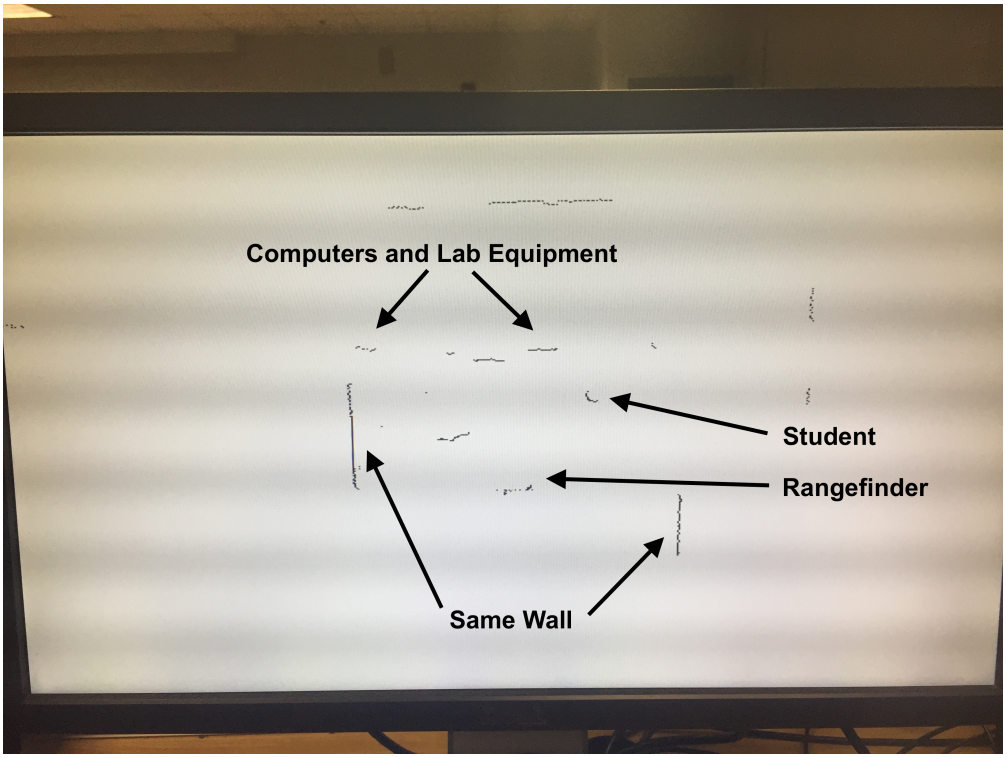
(b) 2D Rangefinder "Floorplan" of Lab at WPI

Figure 39: 2D Floorplan of Lab

In Figure 39b the data points called out in the 2D floorplan include a wall directly to the left of the device, a chair, computers, windows, and the rangefinder. These objects' locations on the floorplan corresponded to their actual locations, so the rangefinder's accuracy was confirmed.

Next, the rangefinder was rotated 180° and the button was pushed again to test the rangefinder's subsequent triggering functionality. Figure 40 shows the rangefinder's view when it was rotated 180°, and the resultant VGA output.

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



(b) Two Overlaid Rangefinder "Floorplan" Captures with 180° Offset

Figure 40: Test of Rangefinder's Subsequent Triggering Functionality

This test confirmed that subsequent rangefinder data captures were triggered successfully and without loss of data. The floorplan shown in Figure 40b shows the data from the second trigger overlaid on top of the existing floorplan. The newly detected objects were called out in the figure. Note that although the rangefinder was rotated 180°, the floorplan did not account for that rotation. Because of this, it was possible for the same object to be detected in different places in subsequent data captures, as shown by the wall on each side of the rangefinder. This issue was resolved by incorporating the digital compass's rotational data in order to geographically reference the data. With the compass heading used to offset the direction of the device, 2D floorplan accurately reflected the relative location of objects.

5.2 IMU Testing

The PmodNAV IMU is a small device that was directly connected to the ZedBoard's Pmod connector. As such, it required 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 required an Extended MIO Pmod so that it was able to be controlled by the PS. As such, the SPI pins were routed to the JD Pmod. Since the only slave this project used on the PmodNAV was its magnetometer, the 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. It was observed that as soon as the magnetometer's slave select line was asserted there was unidentified behavior with all of the other pins. Since the PmodNAV was disconnected this issue was thought to be with routing the SPI pins to EMIO incorrectly, so it was decided to test the SPI transfer via

Pmod JE.

The pins were re-routed to the MIO Pmod JE and similar undefined functionality was observed when the magnetometer's slave select line was asserted. The ZedBoard's SPI errata was investigated until AR# 47511 was found, which describes an unresolved issue in the MIO interface where the SPI controller resets itself when slave select 0 signals asserts [32]. Since this issue affected the PS SPI controller itself, this errata was also the cause of the EMIO's undefined behavior.

To avoid the problems with SPI on the ZedBoard, I^2C was attempted next, since the PmodNav supports I^2C communication. I^2C was routed to Pmod JD. Communication with the PmodNAV's magnetometer was tested through I^2C, but it was learned that the magnetometer was inaccessible through the I^2C bus. As such, SPI was the only option and needed to be implemented.

The SPI pins were again routed to Pmod JD. 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 were not used in any manner. Instead, two GPIO pins were used and configured as pull-ups so that these pins idled high and never had to be written to. This setup was configured in Vivado's Synthesis tab, as shown in Figure 41.

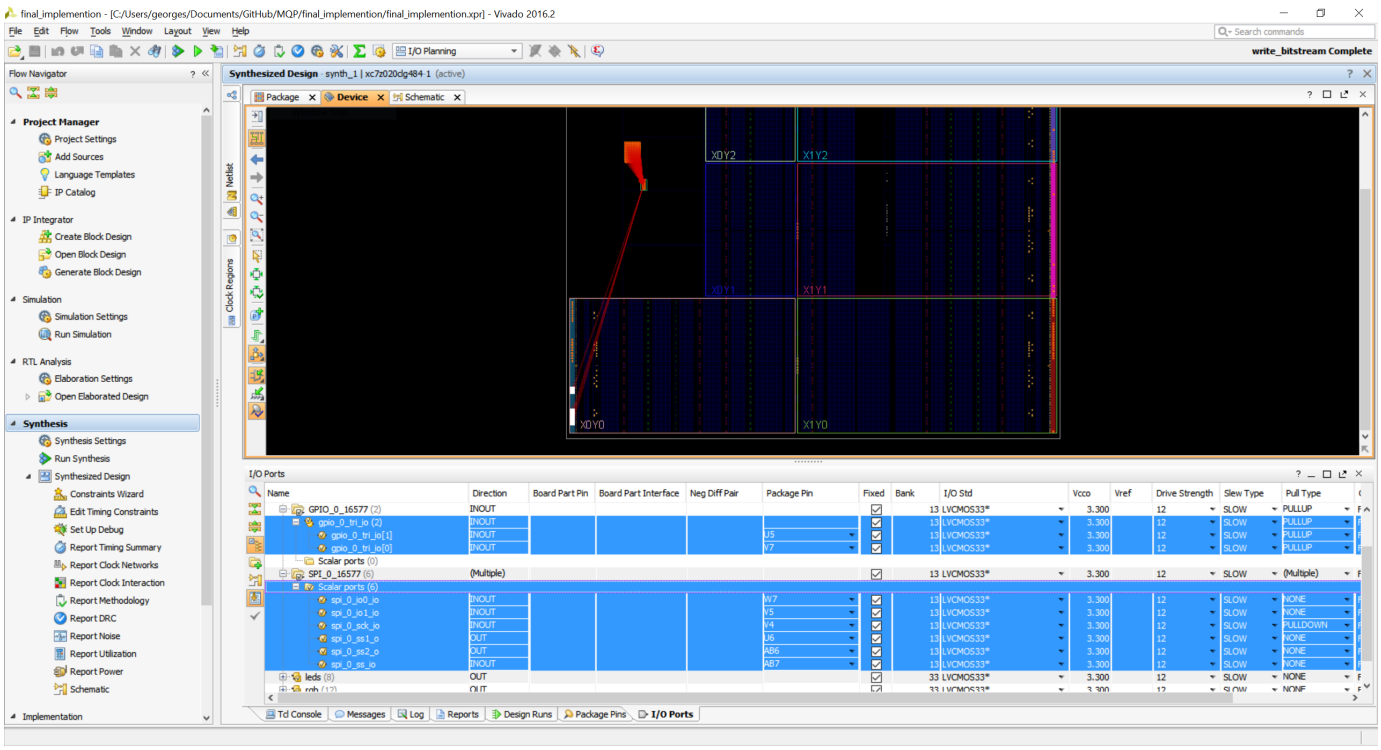
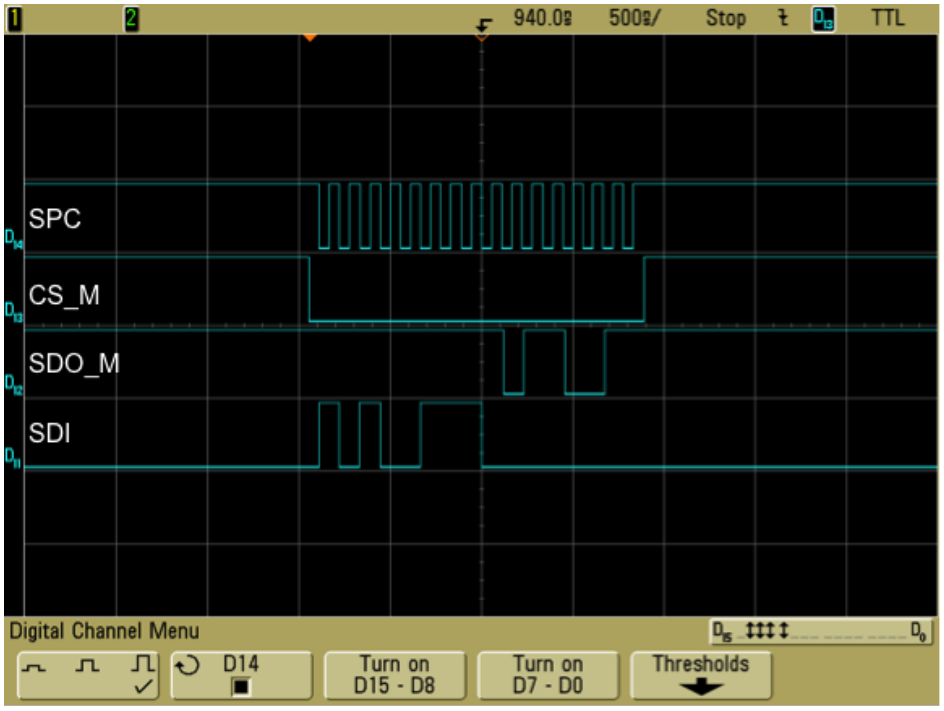
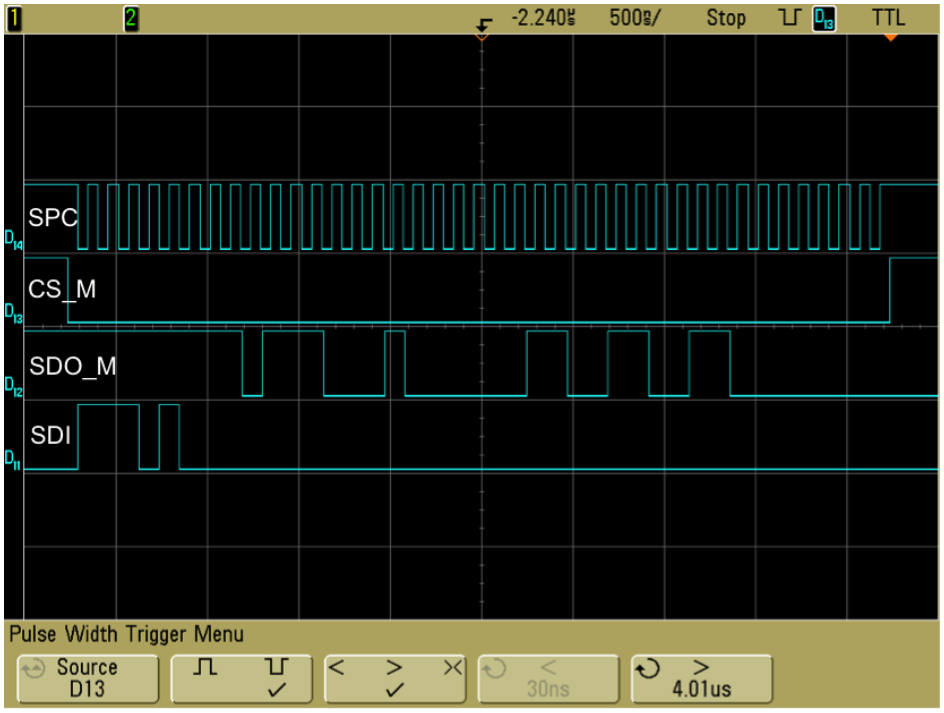


Figure 41: EMIO SPI Configuration for PmodNAV

With this configuration a waveform similar to those shown in Figures 18 and 19 were observed, indicating that the errata was avoided and the the ZedBoard's SPI waveforms were accurate. The PmodNAV 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 after its data processing. By the end of its data processing, the magnetometer's data was transformed into a compass heading. The ZedBoard's UART was routed to USB UART and connected to a serial console. The sensor suite was rotated, and the compass heading was observed. The results were inaccurate and inconsistent until the sensor suite was moved as far from the lab bench as the wires allowed. The PmodNAV was a very sensitive piece of equipment that was susceptible to 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.

5.2.3 Interfacing with the ADIS16375 IMU

Although this project ultimately interfaced with the PmodNAV IMU, this was not the first IMU that communication was attempted with. Originally, the ADIS16375 Six Degrees of Freedom Inertial Sensor, shown in Figure 43, was intended to be used in the sensor suite.

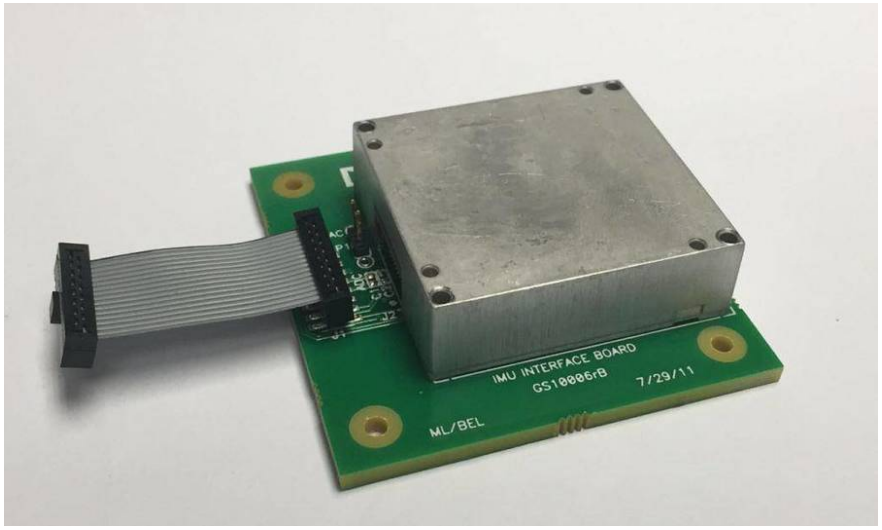


Figure 43: The ADIS16375 Six Degrees of Freedom Inertial Sensor [1]

The ADIS16375 IMU was a highly sensitive, heavy duty device. It had a tri-axis gyroscope, a tri-axis accelerometer, and a temperature sensor, had onboard functionality to

calculate delta-angle and velocity, and used SPI communication. Although the ADIS16375 did not have a magnetometer, its gyroscope's sensitivity combined with its sample rate and onboard delta-angle calculation made it perfect to account for rotation. However, communication was not able to be established with it. There was no way to test the ADIS16375 so its functionality could not be confirmed in any manner. Instead, the PmodNAV was implemented as a quicker and simpler 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 [27]. A simple Verilog module for the Nexys3 Spartan-6 FPGA board was created that supplied the camera module with a 24MHz master clock signal, and a switch was used to toggle the output enable line. With this module implemented, the camera module's default outputs were then observed. In order to interface the camera module with an FPGA, the breakout board shown in Figure 44 was also designed to make the module's pins more easily accessible.

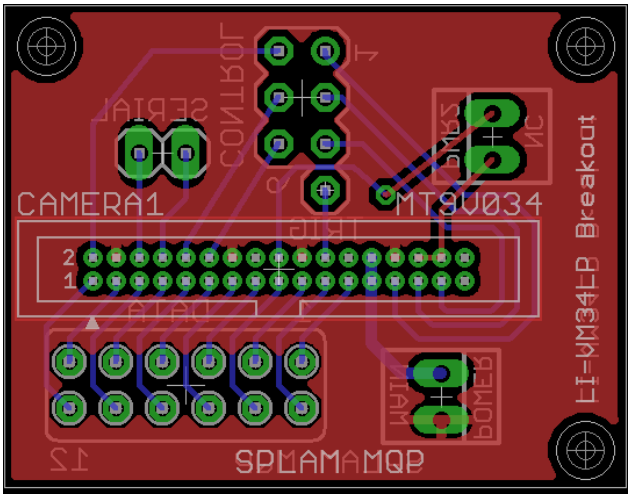


Figure 44: LI-VM34LP Breakout Board

5.3.1 I²C Control

Although the MT9V034 camera's I²C control registers were closed source, the previous model's register settings were found to work with the current model [26]. 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, where the first packet consisted of a request to address 0x00 of device 0x058, and the second packet consisted of the camera's response of 0x1324.

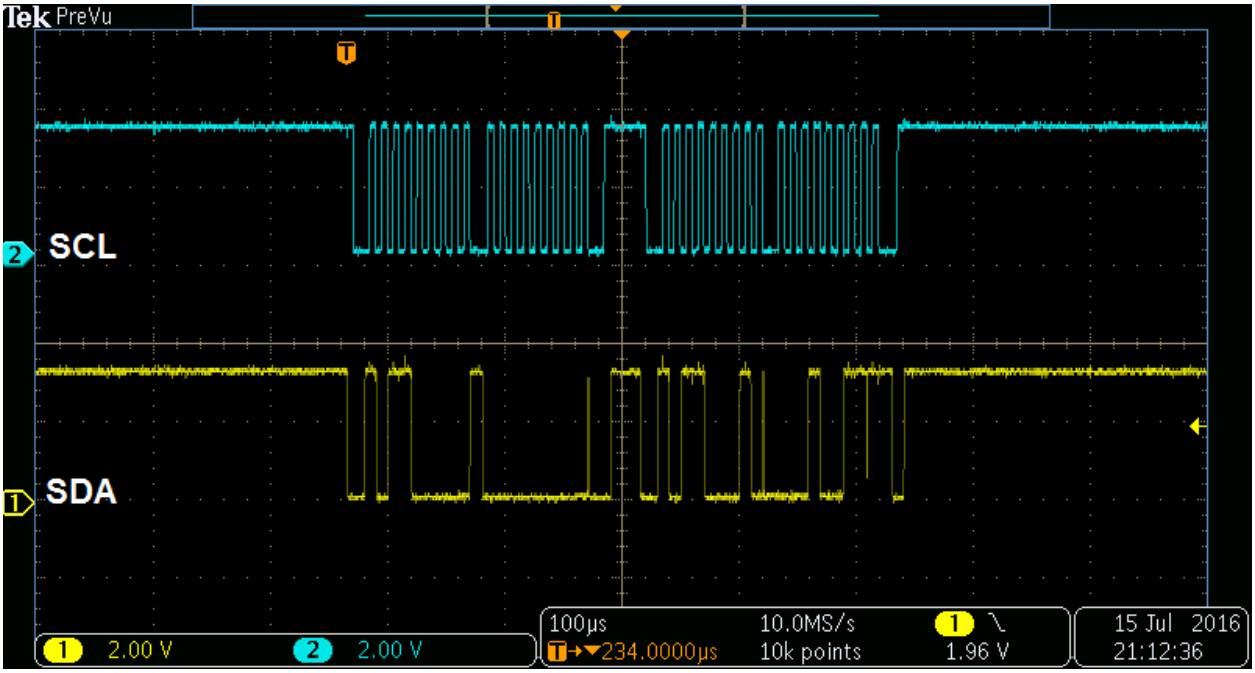


Figure 45: Example I²C Transfer with Camera

After the camera I²C was deemed working, the camera control register was modified to put the camera in "snapshot" mode. In this mode, the camera module no longer continuously took pictures, and only gathered new images when an external trigger was activated. This was the mode in which each camera needed to operate to acquire stereo imagery, since a shared trigger line allowed for both cameras to be controlled simultaneously.

According to the previous camera model's datasheet, the camera module's operational mode was set using control register 0x07. By default, this register was set to a value of 0x0388,

which corresponded to master mode with parallel output and simultaneous readout of pixel data enabled [26]. The camera was placed in trigger mode by writing the control register 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 camera control register bit descriptions can be found in Appendix item B [26].

A button input was then attached to the camera's TRIGGER input line, and the TRIGGER and FRAME_VALID lines were observed on channels one and two of the oscilloscope, as shown in Figure 46. This oscilloscope screenshot demonstrated that the camera was no longer in continuous operation, since FRAME_VALID only asserted itself in response to a TRIGGER input.

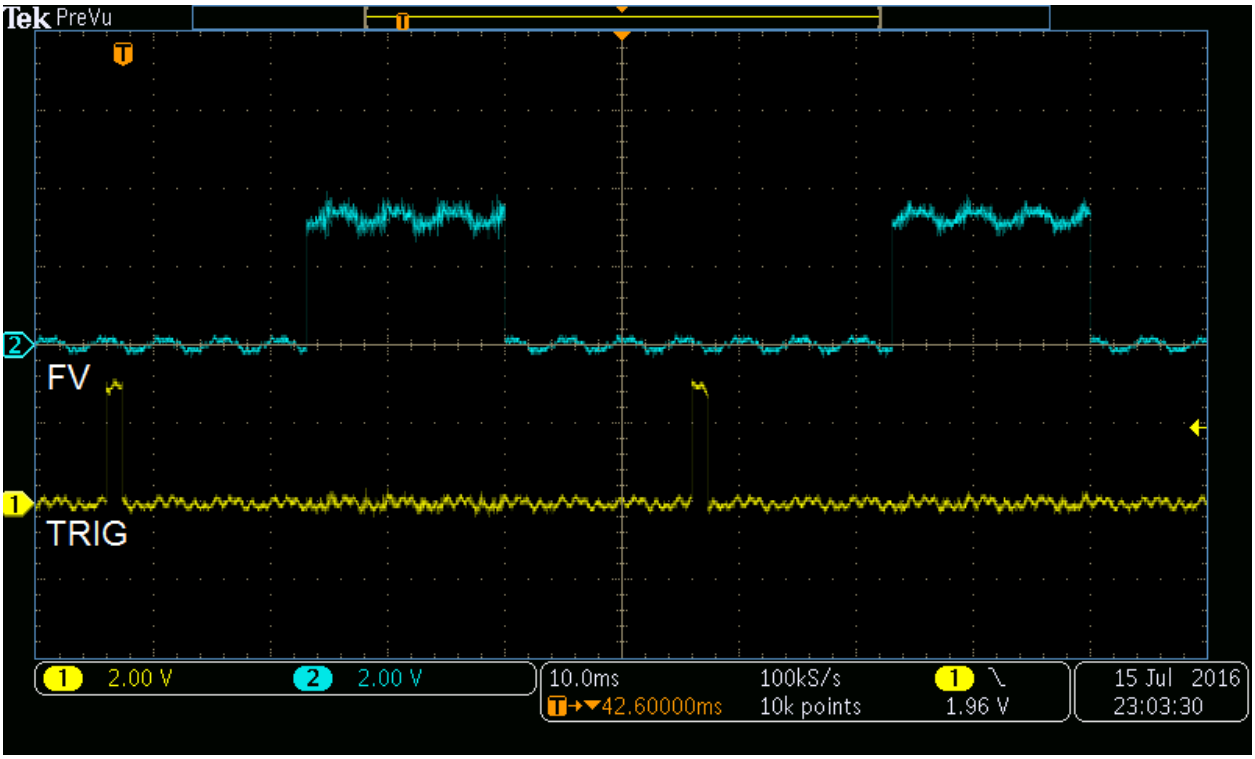


Figure 46: Camera Trigger and FV in Trigger Mode

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, the camera's I^2C write functionality was disabled. This feature was disabled when

0xBEEF was written to the register lock register, or when the power of the device was cycled.

5.3.2 Data Management

After a camera control interface was created that placed the MT9V034 camera module in trigger mode, image data was then acquired from the module for viewing. The included AL422B FIFO IC made it so that all triggered imagery was captured and stored external to the FPGA, and allowed for image data to be read in non-continuous 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. Development time of this system was reduced by including an external microcontroller for controlling the camera module's I^2C control interface. Various buttons and switches on the FPGA controlled the camera output and trigger, and allowed for a user to trigger an image for storage on the AL422B FIFO. Once the image had been stored on the FIFO, the FPGA read the image line-by-line into an internal buffer. An internal System on Chip (SoC) microcontroller controlled 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 then signaled to the microcontroller when this buffer was full, 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 signaled 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, and allowed 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 are located in Appendix item D.i.

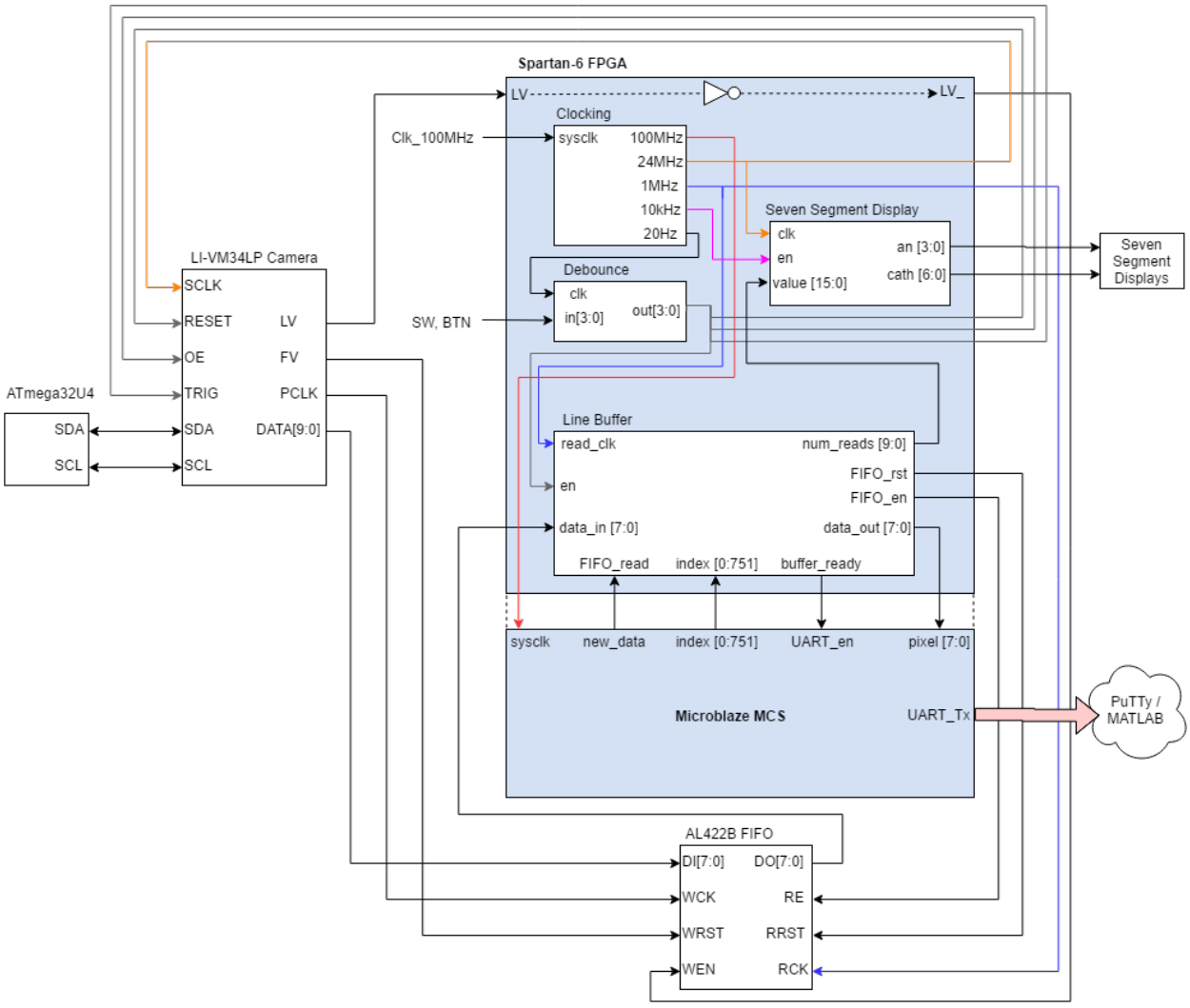


Figure 47: Camera Test System Block Diagram

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. An internal counter and seven-segment display controller were also implemented on the FPGA to simplify debugging, and displayed a running count of the number of pixel lines that were read into the FPGA's internal buffer, ranging from 0x0000-0x01E0 (0-480).

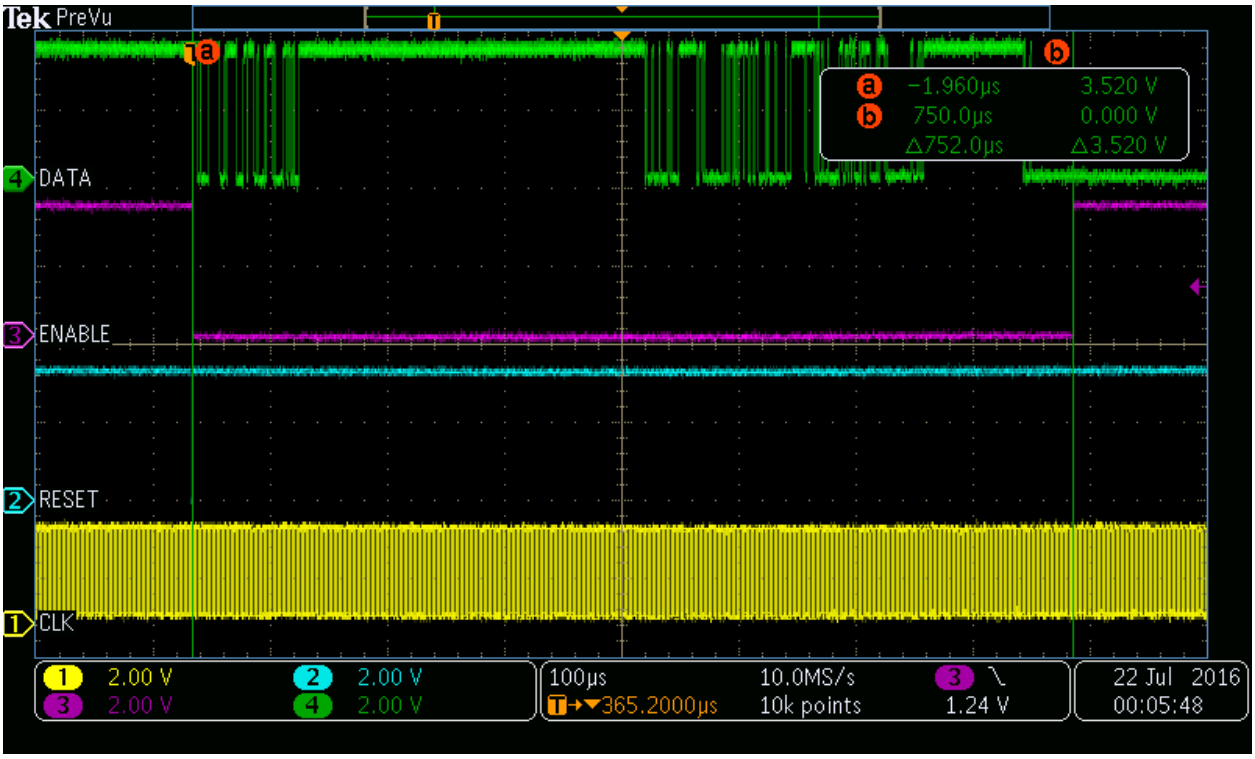


Figure 48: Transferring Line Data from FIFO to FPGA

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 D.i was implemented on a Microblaze SoC that transmitted camera line data from the FPGA's internal line buffer to an attached computer over UART. An example of the microcontroller's UART output is shown in Figure 49. The microcontroller printed the value of each pixel followed by a newline and carriage return, starting with the top left pixel in the acquired image.

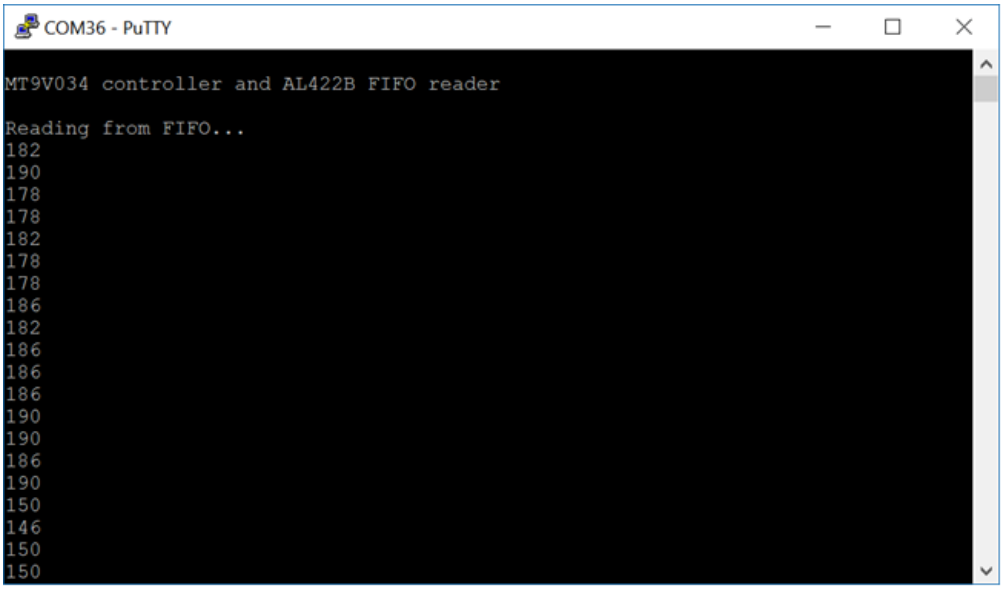


Figure 49: Reading FIFO Data

After the image was received through PuTTy, the MATLAB script found in Appendix item D.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.

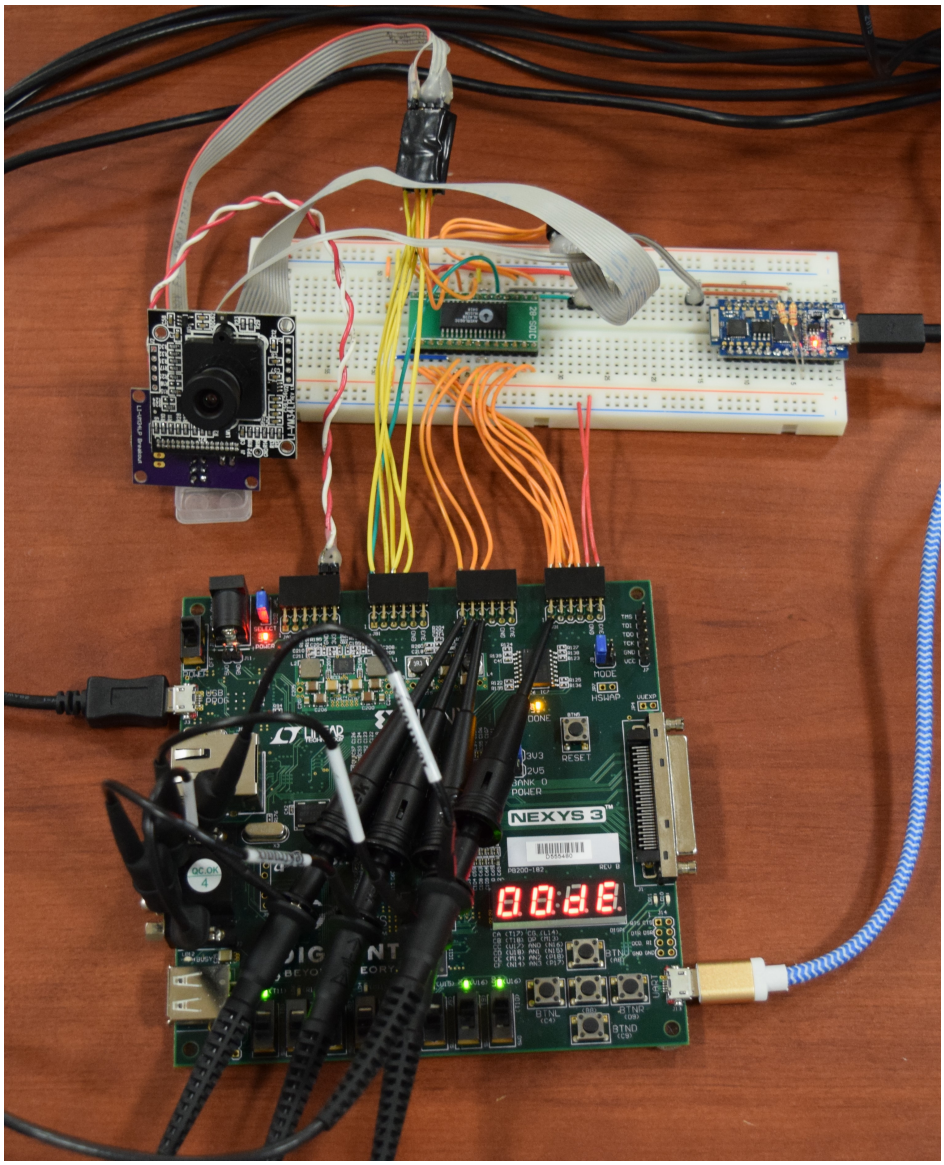


Figure 51: Camera Test Setup

5.4 Final Camera Hardware Implementation

After successfully gathering image data from a single camera module, an interface was 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 IO 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 boards described earlier, 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 have been 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 used for 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, that they significantly reduced FPGA memory and timing requirements. A second major advantage of including these FIFO modules in the camera interface was that their data output lines supported being 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

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

the lines at a time. Since the bulk of each camera module's required pin count lay 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, 27].

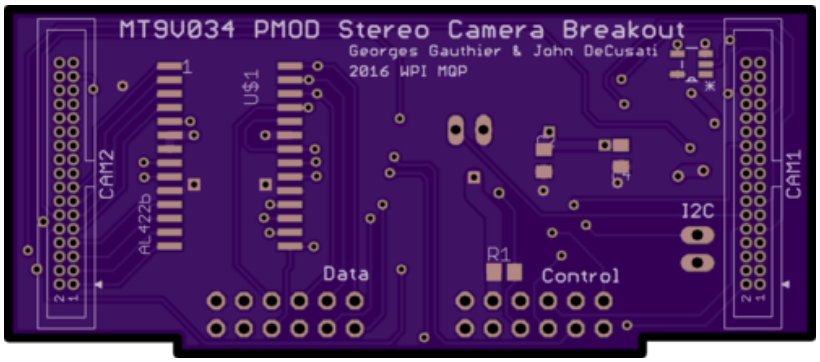
Along with the shared camera data lines between each AL422B FIFO module, it was also possible to connect several other signals in parallel. Since each camera image capture needed to be triggered at approximately the same time for an stereo imaging setup, it was already desirable to connect both camera TRIGGER lines together. The RST, OE, SDL, SCA, and SCLK lines of each camera module were also tied together in pairs of two, and the OE lines were 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.

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

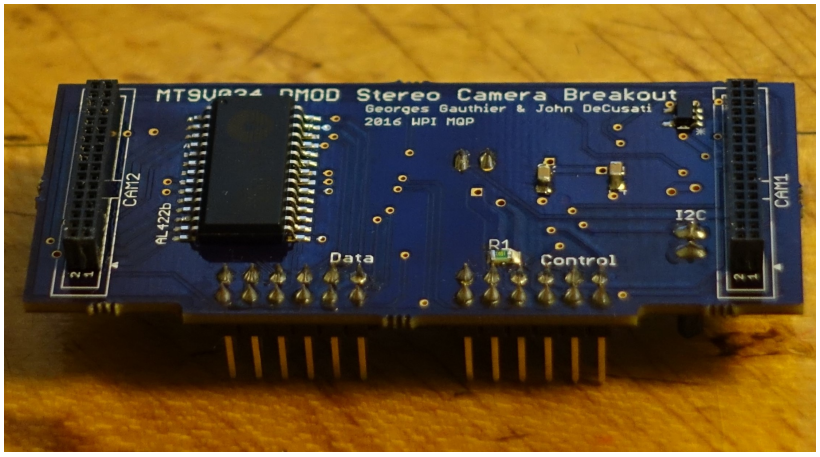
Note that each FIFO needed to be controlled individually, requiring an additional Read Reset (RRST) and Read Enable (RE) pin per camera, as well as a shared Read Clock (RCK) line. This brought the total pin count required by the stereo camera setup to 16 pins plus two I^2C pins, which was conveniently the number of GPIO available in two Pmod headers. This setup was implemented as shown in Appendix item C, 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 D.i that tested the stereo camera breakout board's functionality using the Nexys3 platform. A switch input selected 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 an attached VGA display. The image was then independently written to the display according to VGA pixel timing. This process was repeated at a high rate of speed, and allowed for a real-time 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

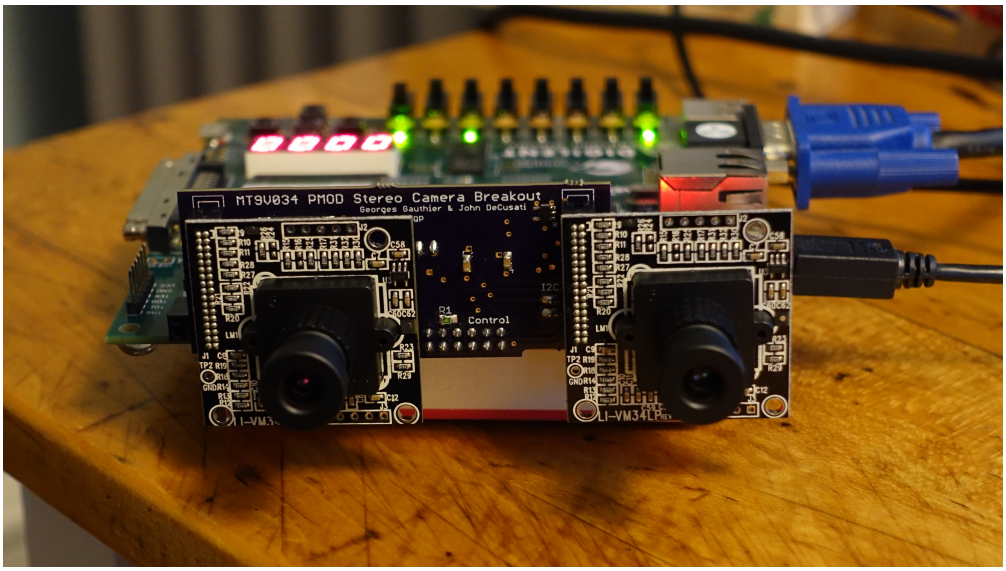


Figure 53: Stereo Camera Breakout Under Test

After manually focusing each camera using the VGA module described above, the UART transmission implementation described in Section 5.3.3 was used to transmit image data from the stereo cameras to a computer for further analysis. As shown in the example image in Figure 54, 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 shown in Figure 50.

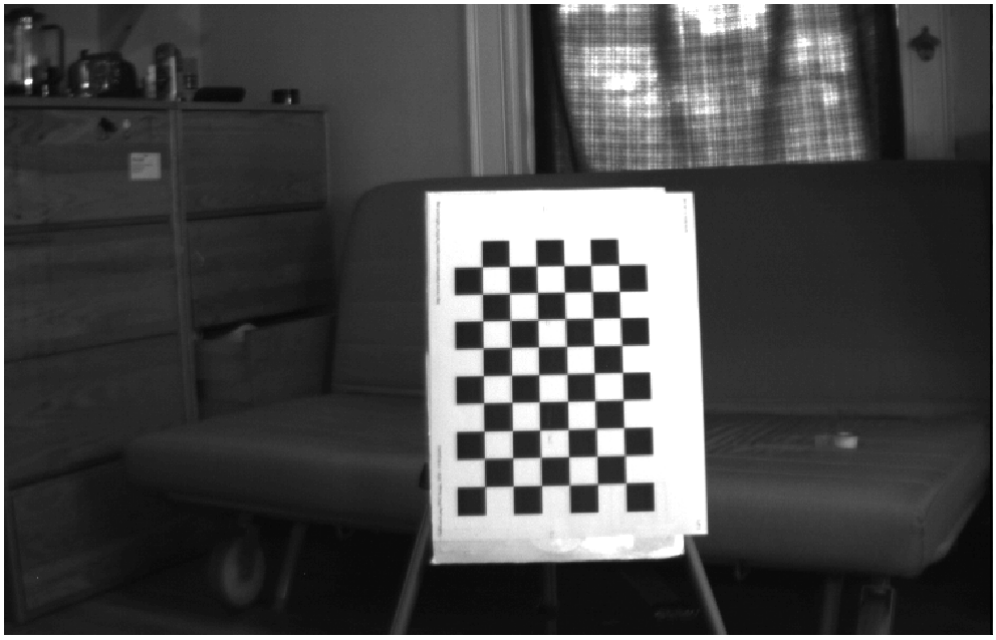


Figure 54: Stereo Camera Breakout Sample Image

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 D.i, with the exception that internal Block RAM was used to buffer an entire image captured from the cameras. Note that a custom AXI interface was also included in the test implementation, which allowed for the option of reading image data into the Zynq Processing System for more advanced testing and export via PS peripherals such as UART.

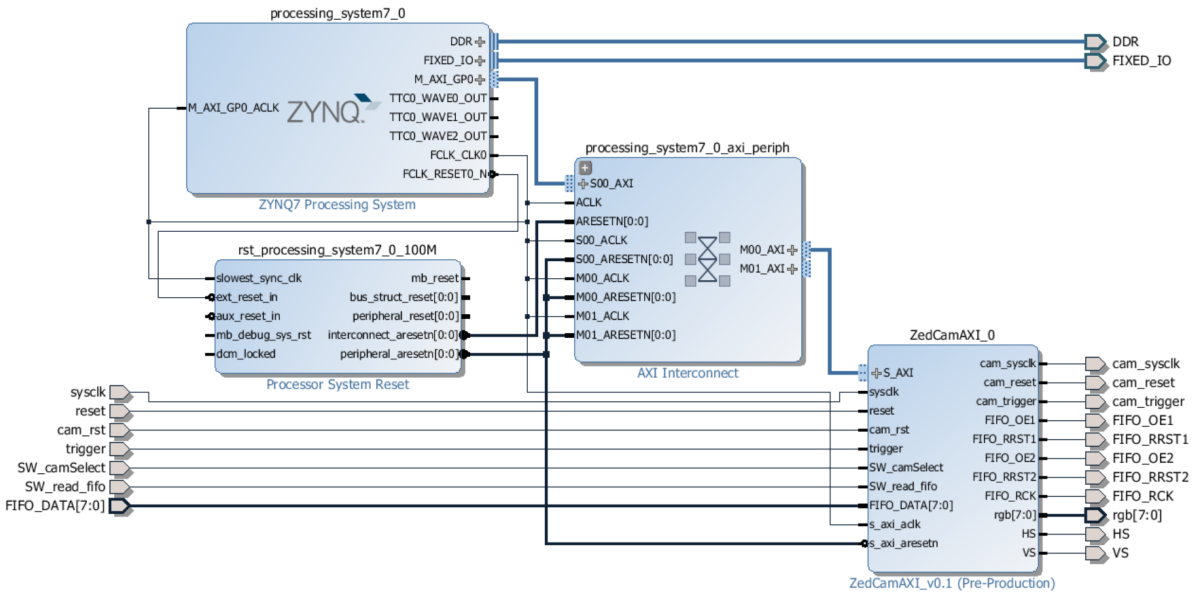


Figure 55: ZedBoard BRAM Camera Test Block Diagram

Since a single camera image contained 8 bits $\times$ (752 $\times$ 480) pixels, a simple dual-port Block RAM module containing $752 \times 480 = 360960$ 8-bit addresses was created for storing the output of the AL422B FIFO reader module. Dual-port Block RAM 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 that the implementation functioned properly. An example of the output from this implementation is shown in Figure 56 below.

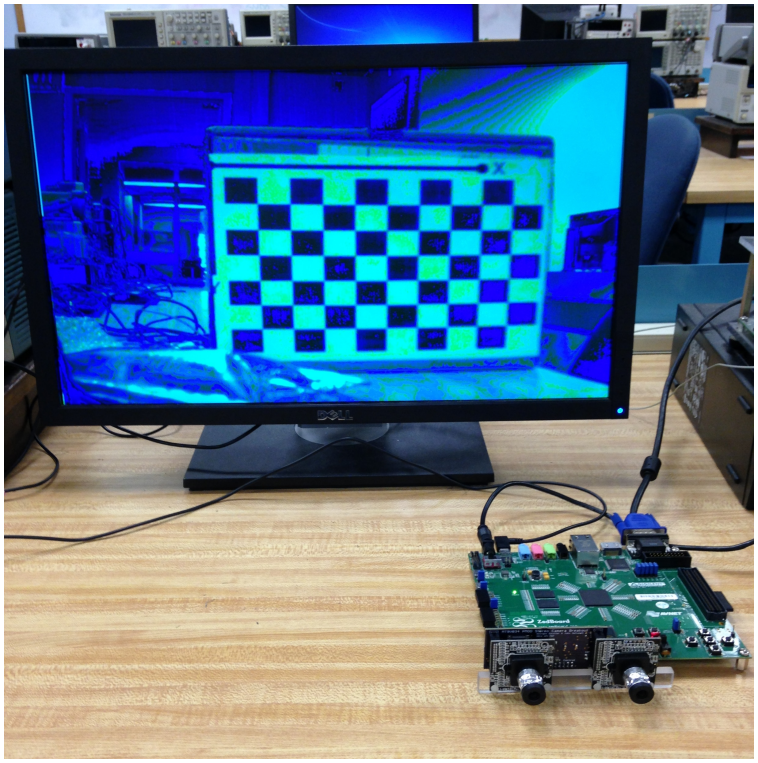


Figure 56: ZedBoard BRAM Camera Test

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 were then addressed.

5.4.3 Resource Management

One major issue encountered while dealing with resource management on the ZedBoard was managing Block RAM resources. The Zynq7020 processor used on the ZedBoard contained 140 individual blocks of 36Kb BRAM, which was equivalent to 630,000 8-bit bytes of memory [35]. Although this was plenty of memory for buffering a single 752x480 camera image, three separate image buffers needed 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 was then 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 per input camera image, 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 36Kb Block RAM modules available on the ZedBoard's Zynq7020 processor, which left additional resources for use in the IMU and rangefinder implementations.

5.5 Disparity Testing

After the camera interface was verified as functional, a large portion of time was spent implementing a disparity algorithm that allowed for the extraction of 3D depth information from stereo image data. This algorithm was first implemented in MATLAB, and was then re-designed to operate within programmable logic.

5.5.1 Image Rectification

In order to perform the most accurate block matching 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 [8].

After extensive testing with the stereo camera breakout board and disparity module detailed in the following sections, it was also found that camera imagery captured from the stereo camera interface contained consistent horizontal epipolar lines between both images.

These lines were accurate enough for the custom disparity algorithm to process without additional calibration, saving a large amount of calibration time in the image processing pipeline. 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 D.v [25]. This implementation operated on the "cones" standard test image set, and produced a resultant disparity image from the given input image pair. In the case of this specific example, the algorithm performed a 7x7 Sum of Absolute Differences block matching process on 50-pixel search ranges across horizontal epipolar lines between the two images. Note that the block size and search range were also implemented so that they could be customized by the user to test the algorithm's functionality.

Overall, the MATLAB disparity test implementation was 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 a resultant disparity image from 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 followed 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. To maintain simplicity for initial testing, the test algorithm originally operated on the 20x7 pixel test images shown in Figure 58. The search range and block size for this module were set to 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 calculated the Sum of Absolute Differences between the template and search block. The SAD value was then placed in a vector that matched the length of the search range, with a vector index that corresponded to the current pixel location being searched. If the vector was not 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.

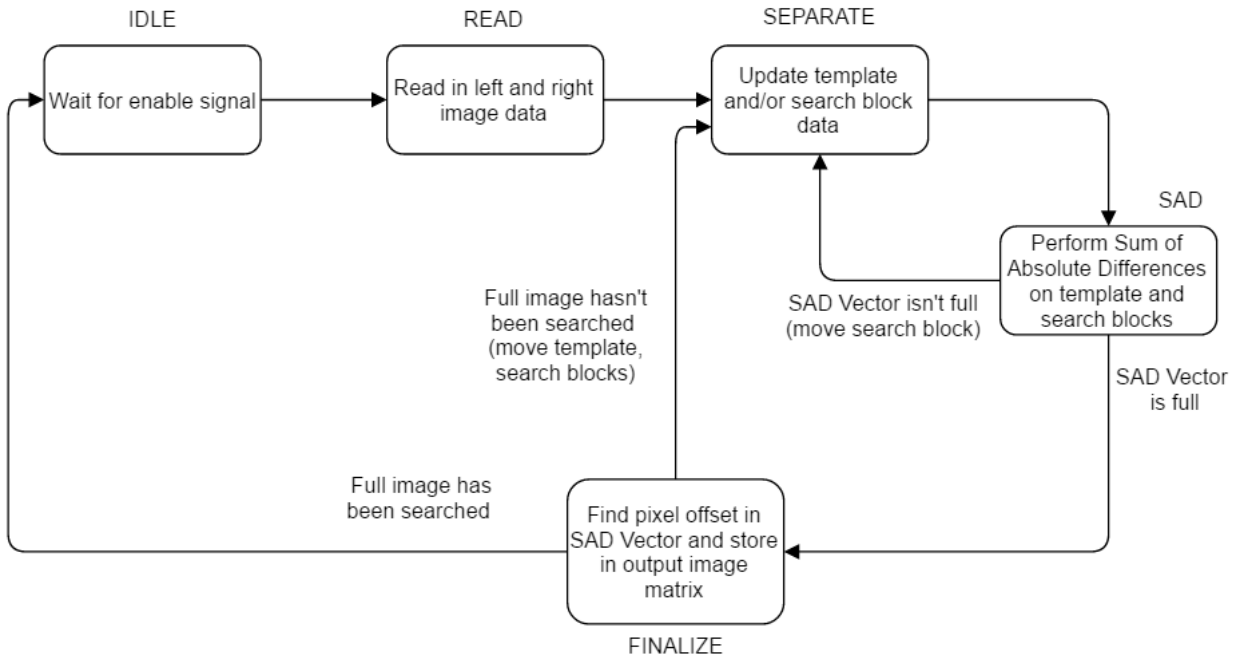


Figure 59: Disparity Test Implementation

The finalization state was used to search through the SAD vector for the lowest value. The index of this value within the SAD vector was equivalent to a disparity value for the given template block location. This value was converted to a distance measurement using Equation 9, and was stored in the output image location. If the output image was not fully populated with distance values, the state machine reverted back to the separate state. Otherwise, the state machine advanced to its idle state, indicating that the resulting disparity image was ready 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 indicated by `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`.

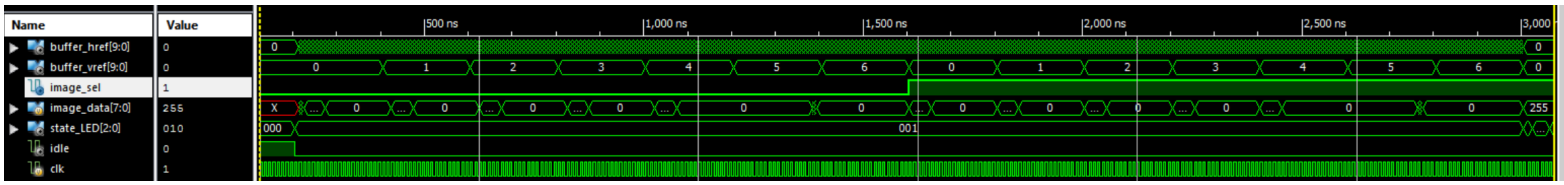


Figure 60: Image Read Sequence

After the state machine finished reading both images into local memory, it then iterated 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 advanced 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`.

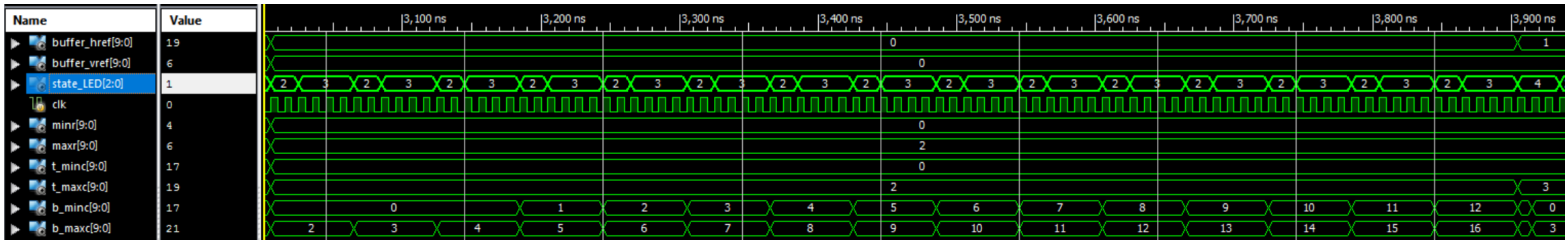


Figure 61: Disparity Search Vector

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 then advanced 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 approached to the right side of the image, since the search range eventually exceeded the distance from the template block to the far 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 from Figure 58 is shown below in Figure 62. In the case of this example, `numBlocks` represented the number of blocks included in the current disparity search. Since the width of the test image was 20px and the search range was set to 15px, `numBlocks` began to decrease after the 4th disparity search was performed, as shown below.

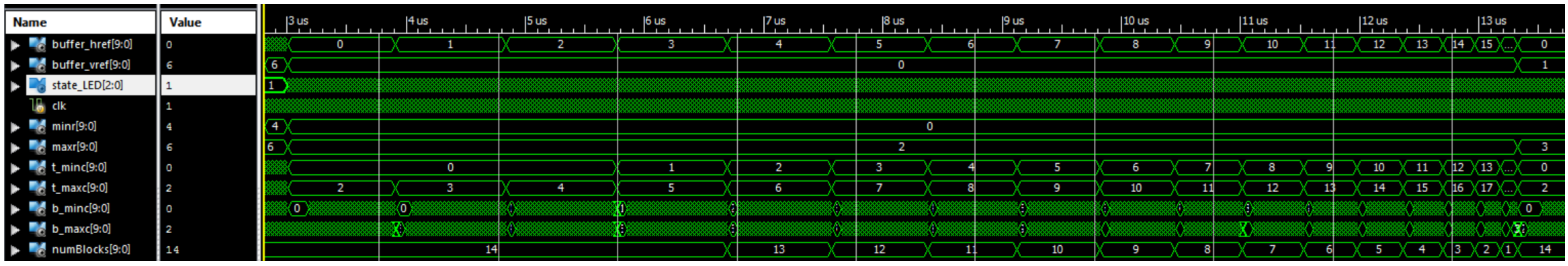


Figure 62: Horizontal Pixel Row Search

After an entire horizontal row of pixels was 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 was analyzed. An example of a disparity search through the entire 20x7 image is shown below in Figure 63.

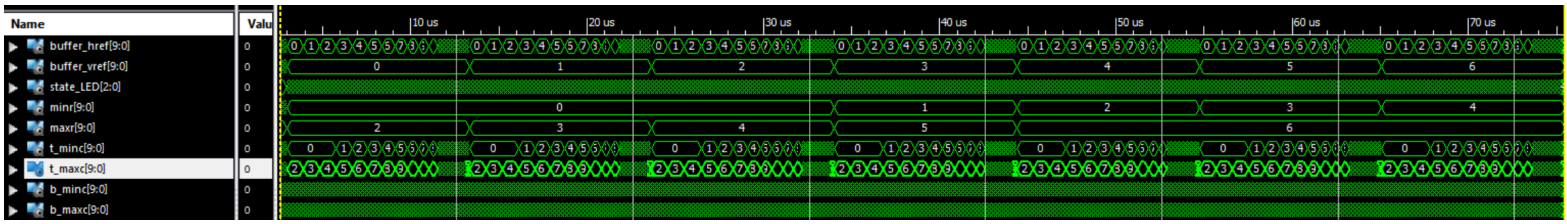


Figure 63: Full Image Search

The output disparity image from the test analyzed above is shown in Figure 64. Note that the artifacts in the resultant image were due to the fact that a 5px∗5px template and search block set was used on a 20px∗7px image. The relatively large block size used in comparison to the size of the image made it impossible for any block comparison to avoid the block contained in the upper left corner of the search image. In addition, 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.

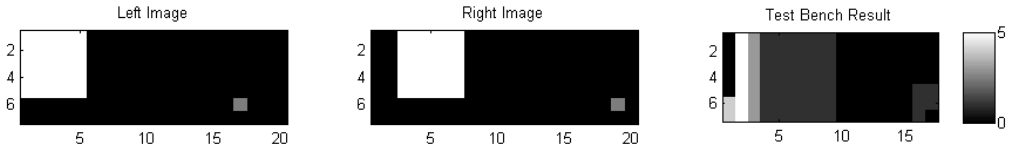
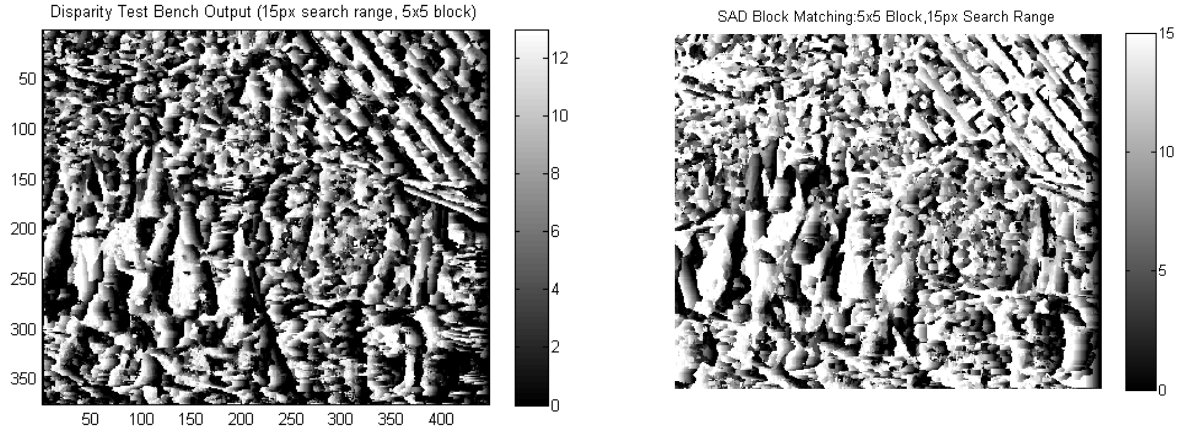


Figure 64: Disparity Test Results

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, the Verilog disparity implementation's results were compared 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 from disparity search on the "cones" test image set. Note that the outputs from the MATLAB and Verilog disparity search algorithms were 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

After the disparity implementation was fully tested in programmable logic using a Verilog Test Bench, a modified version of the implementation was created for final use.

5.5.5 Final Disparity Implementation

Several structural changes were made to the original disparity implementation in order to create a finalized module for use in the overall system. In order to increase the speed of the disparity algorithm's output, several portions of the Verilog module used in the previous section were modified for increased parallelization and decreased overall latency associated with vectorized summation calculations. Most of this parallelization consisted of modifying

2D memory arrays used for storing the template and search blocks, and breaking said arrays into individual vectors. For example, instead of using a 7x7 memory array for storing one "block" of pixels, the block was broken into seven separate 1x7 vectors that were then 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.

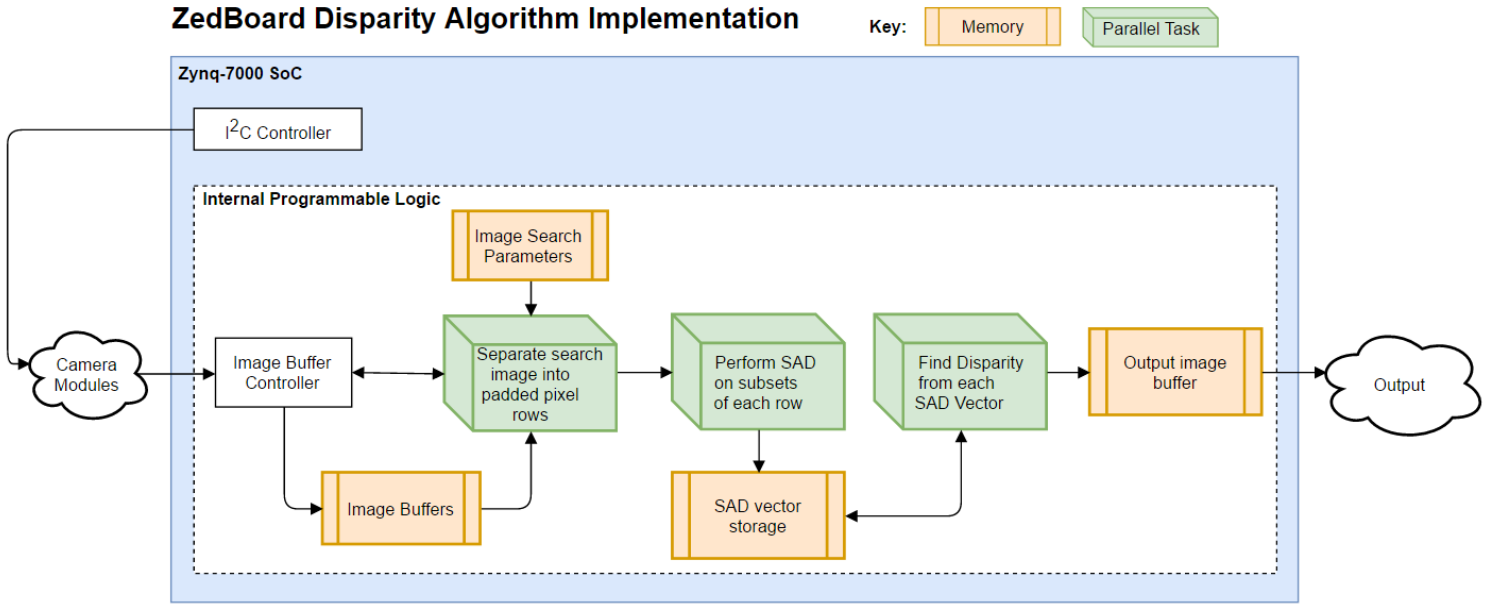
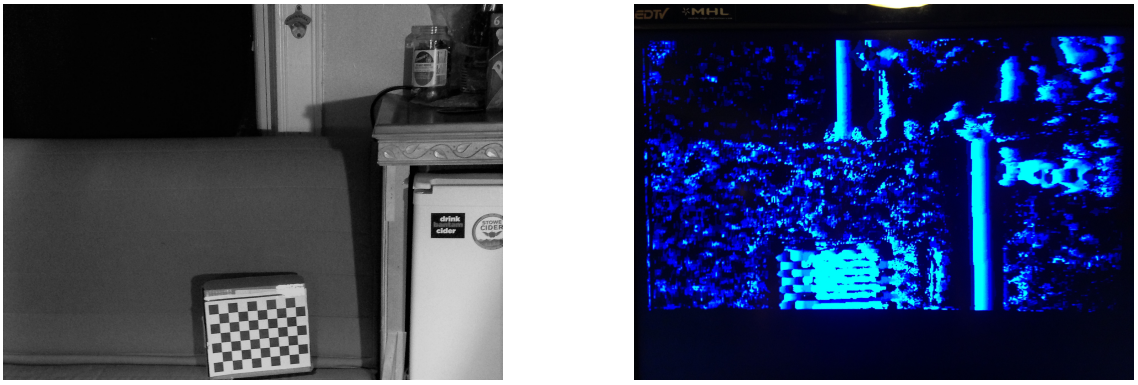


Figure 66: Disparity Final Implementation

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.

The output of the hardware disparity implementation was verified by using a VGA display driver to show the algorithm's output in real time, as demonstrated in Figure 67. With this

output mode successfully implemented, the disparity and camera controller modules were then added into a finalized design that incorporated the IMU and Rangefinder modules.



(a) Device View (b) Resultant Disparity

Figure 67: Disparity Algorithm Output

5.6 Full System Operation

The result of this finalized implementation was a proof of concept SLAM system capable of generating compass-referenced 2D floorplans of its surroundings, as well as 3D depth maps of its entire field of view. The operating modes of this device were user-selectable, and were output to an external VGA display. The final hardware implementation of the project is shown in Figure 68. Note that the stereo cameras and digital compass were mounted rigidly to the ZedBoard, while the scanning laser rangefinder and RS232-TTL converter were attached externally.

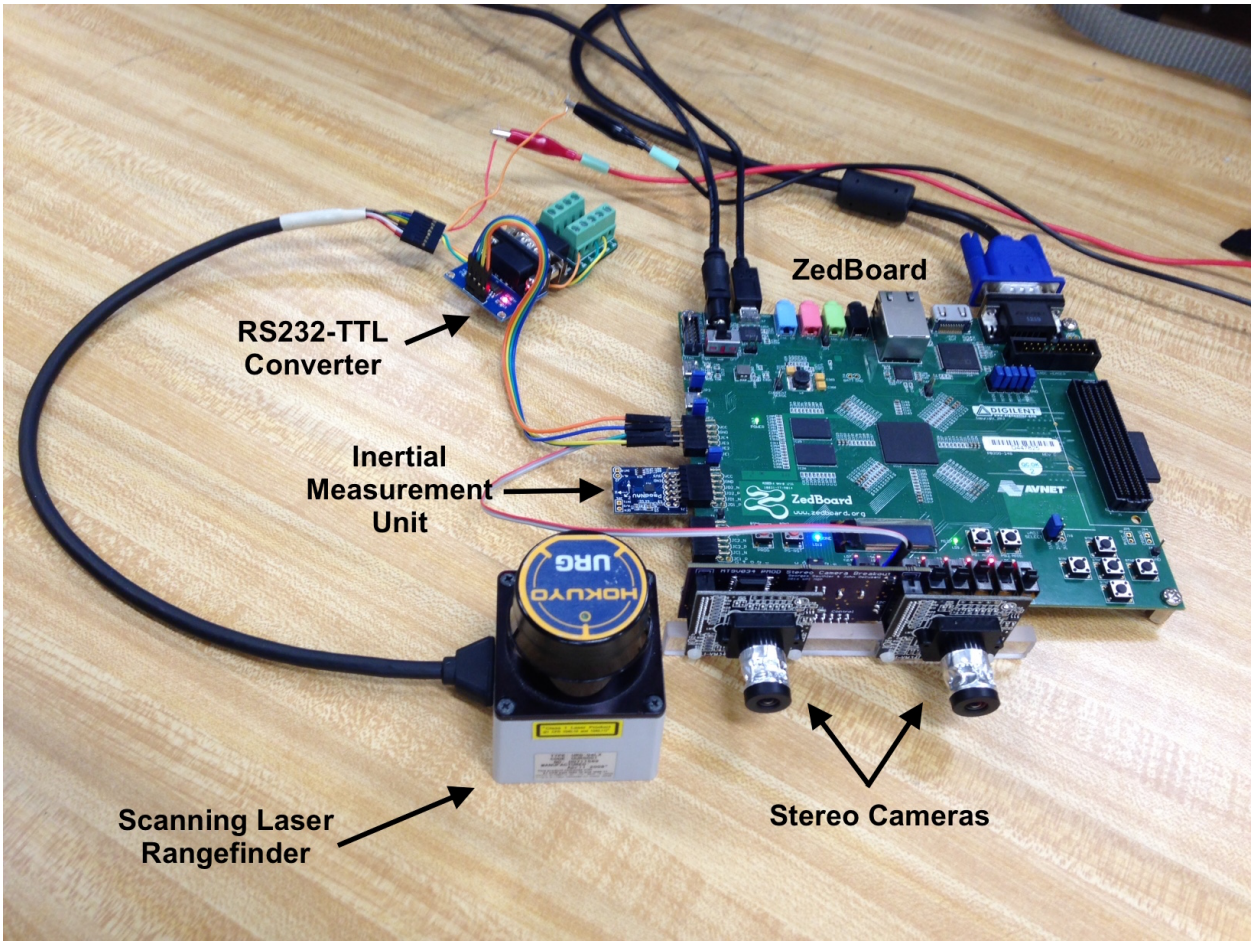


Figure 68: System Hardware

An overall system block diagram is shown in Figure 69. The system implementation was broken down into several major components. At the top level, the Zynq-7020 ARM Cortex A9 processing system was used for controlling low-level sensor peripherals. These included the stereo camera I^2C lines, as well as UART and SPI interfaces for the rangefinder and digital compass, respectively. The ARM processor was also used for continuously pre-parsing rangefinder and digital compass data before it was written to the programmable logic.

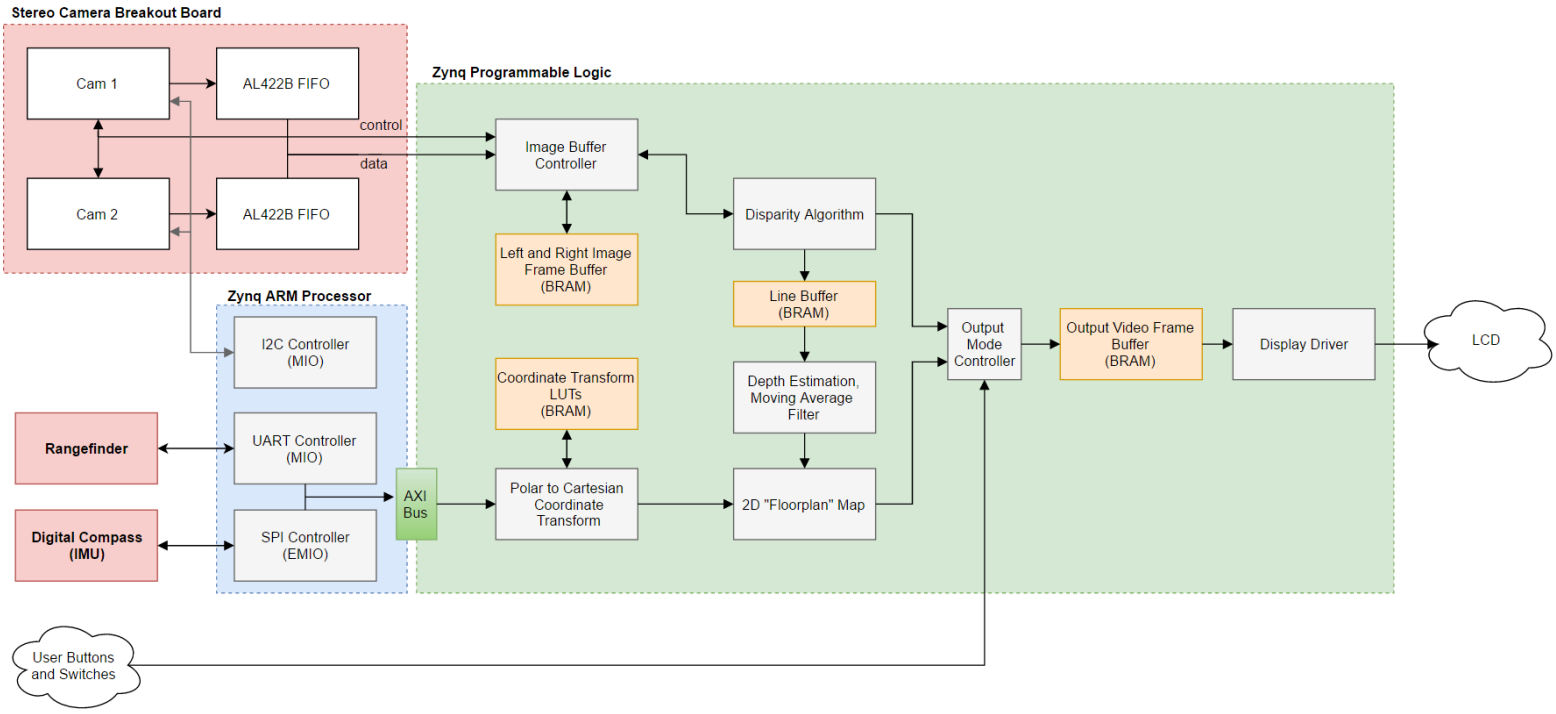


Figure 69: Final System Block Diagram

The programmable logic for this implementation consisted of two main processing pipelines. One main functional process of the PL was to parse image data into depth measurements using disparity. The other was to read rangefinder and digital compass data from the programmable software and convert said data into a compass-referenced 2D "floorplan". Both the stereo disparity and 2D floorplan data paths were then combined at the output stage, allowing for resultant data to be viewed on an attached VGA display based on the current operating mode.

In order to create a fully integrated hardware-software interface that allowed for communication between the Zynq FPGA fabric and ARM processor, a custom AXI IP core was created. This IP core served as a top module for all programmable logic located within the green portion of Figure 69. This included all user-defined logic for the rangefinder, digital compass, camera interfaces, and VGA controller.

This implementation supported several output modes based on the positions of the user switches. These output modes included a rangefinder mode, disparity mode, and combined

2D "floorplan" mode. As a demonstration of the device output modes, the sensor suite was used to observe the scene shown in Figure 70, and each output mode was then documented.

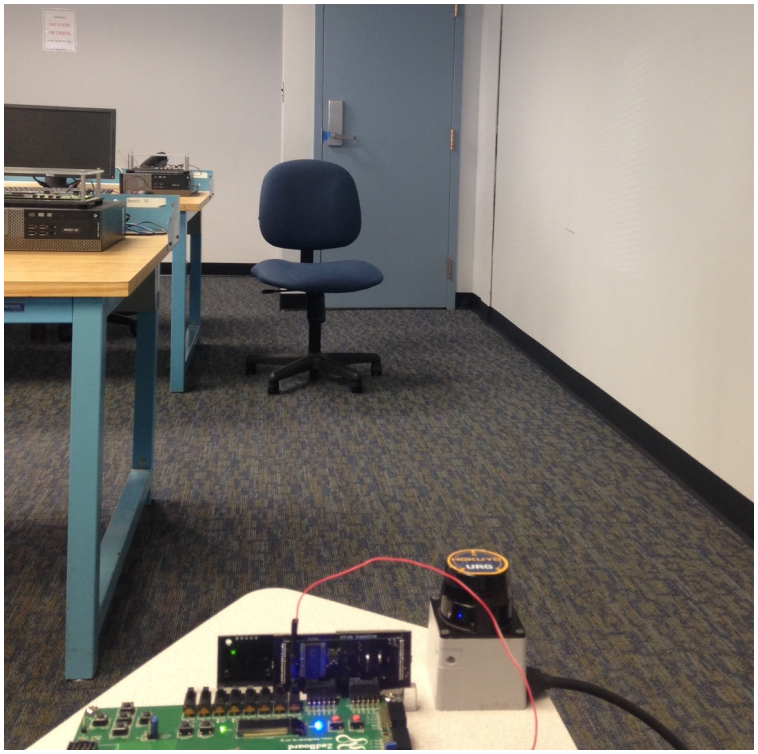
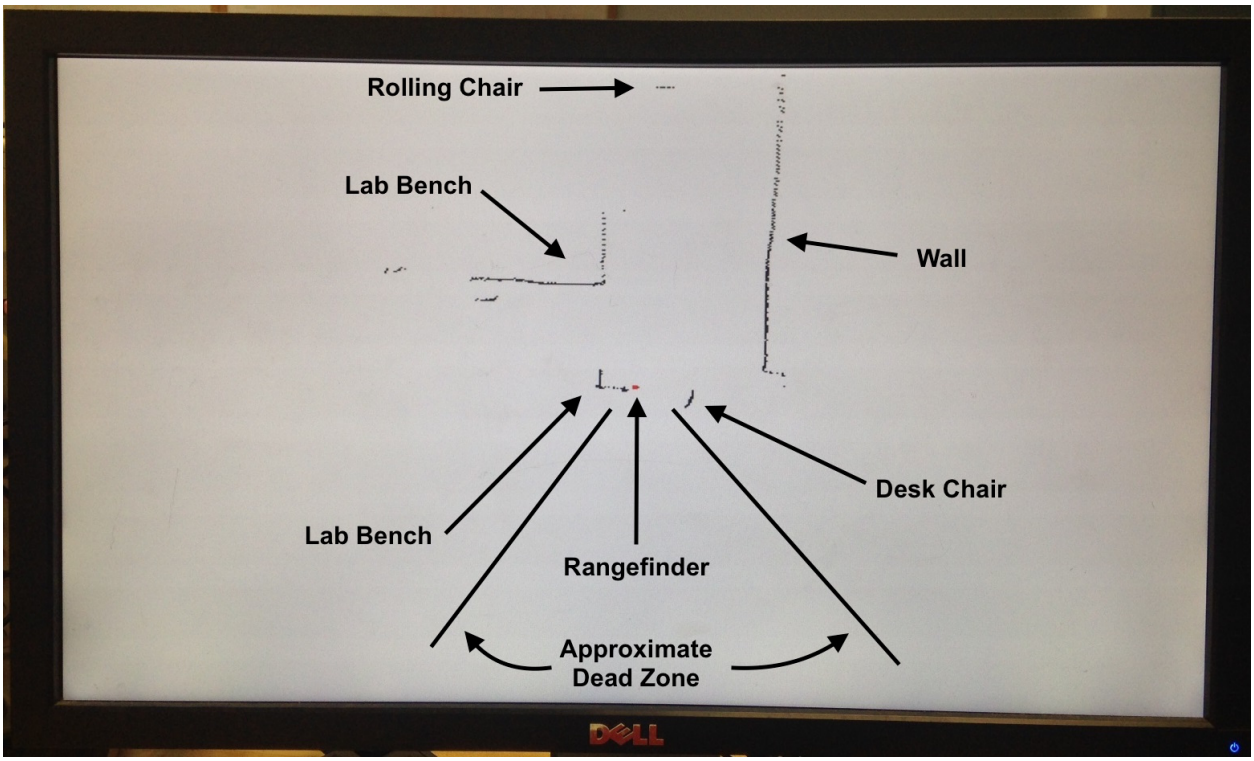
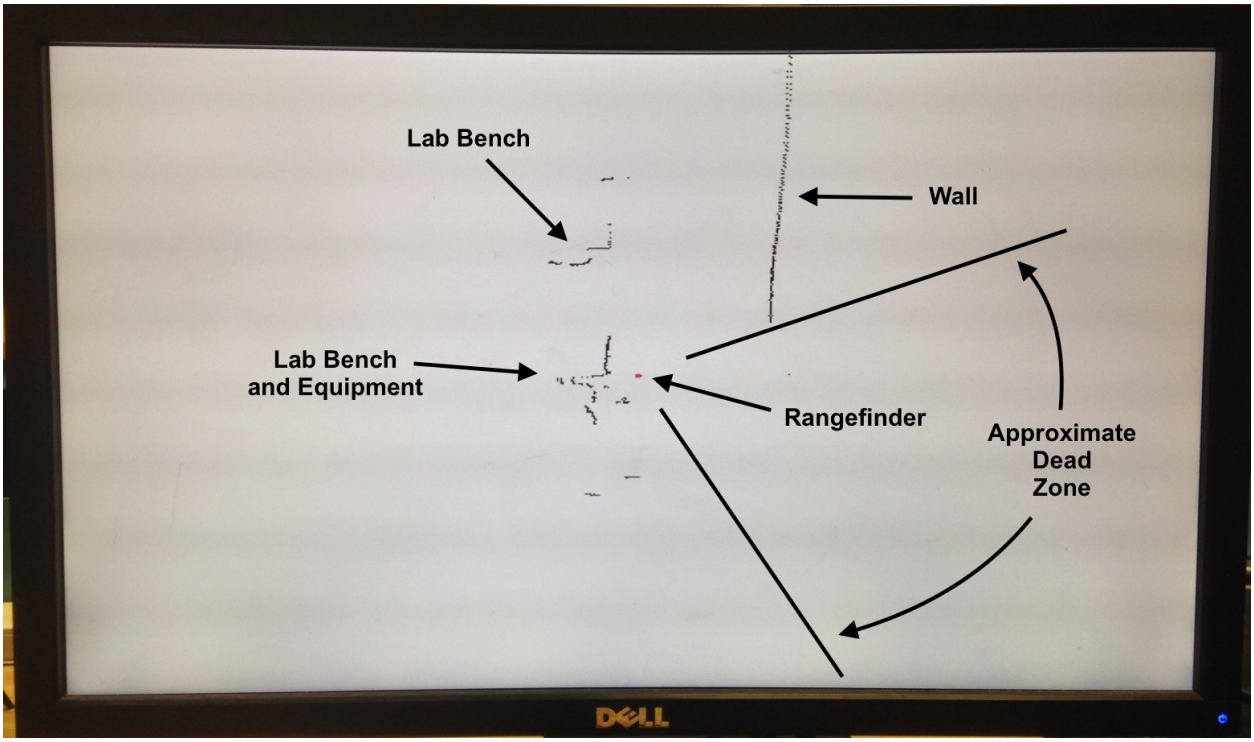


Figure 70: Demonstration Scene

By default, the system was placed in rangefinder output mode, as shown in Figure 71a. In this mode, objects found by the scanning laser rangefinder were displayed in black. The entire scan was also localized to the device's central location, shown in red. All rangefinder data was pre-processed by the programmable software to include an offset from the digital compass before being written to programmable logic. In Figure 71, picture (a) shows the rangefinder output with the device facing due north, while picture (b) shows the rangefinder output with the device rotated west by approximately 90°.



(a) Rangefinder Output



(b) Rotated Rangefinder Output

Figure 71: 2D "Floorplan" Output Modes

A second device output mode was included for displaying continuous stereo disparity. While in disparity mode, a 384x288 pixel depth map was continuously updated to reflect the camera's current field of view. An example of the system output for this mode may be found in Figure 72.

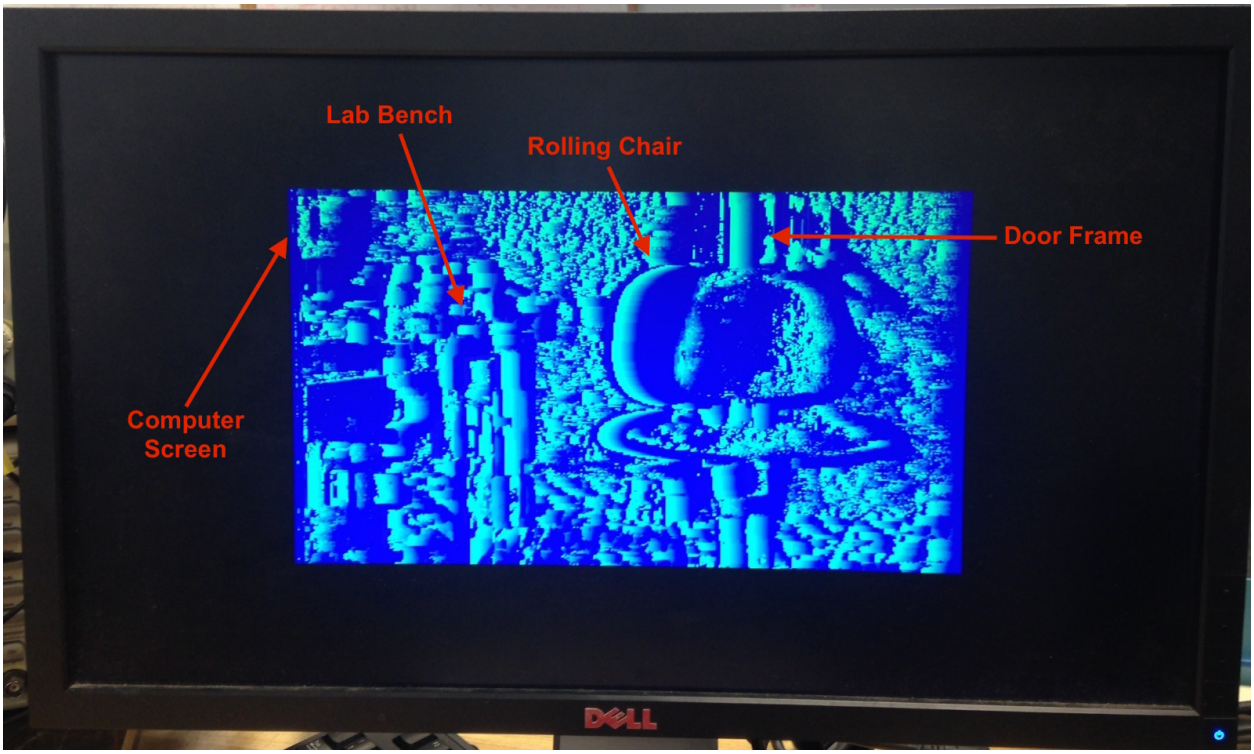


Figure 72: Disparity Output Mode

A final output mode was also included to incorporate filtered disparity data overlaid onto the 2D "floorplan" 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 limitations in viewing distance. This output mode is shown in Figure 73.

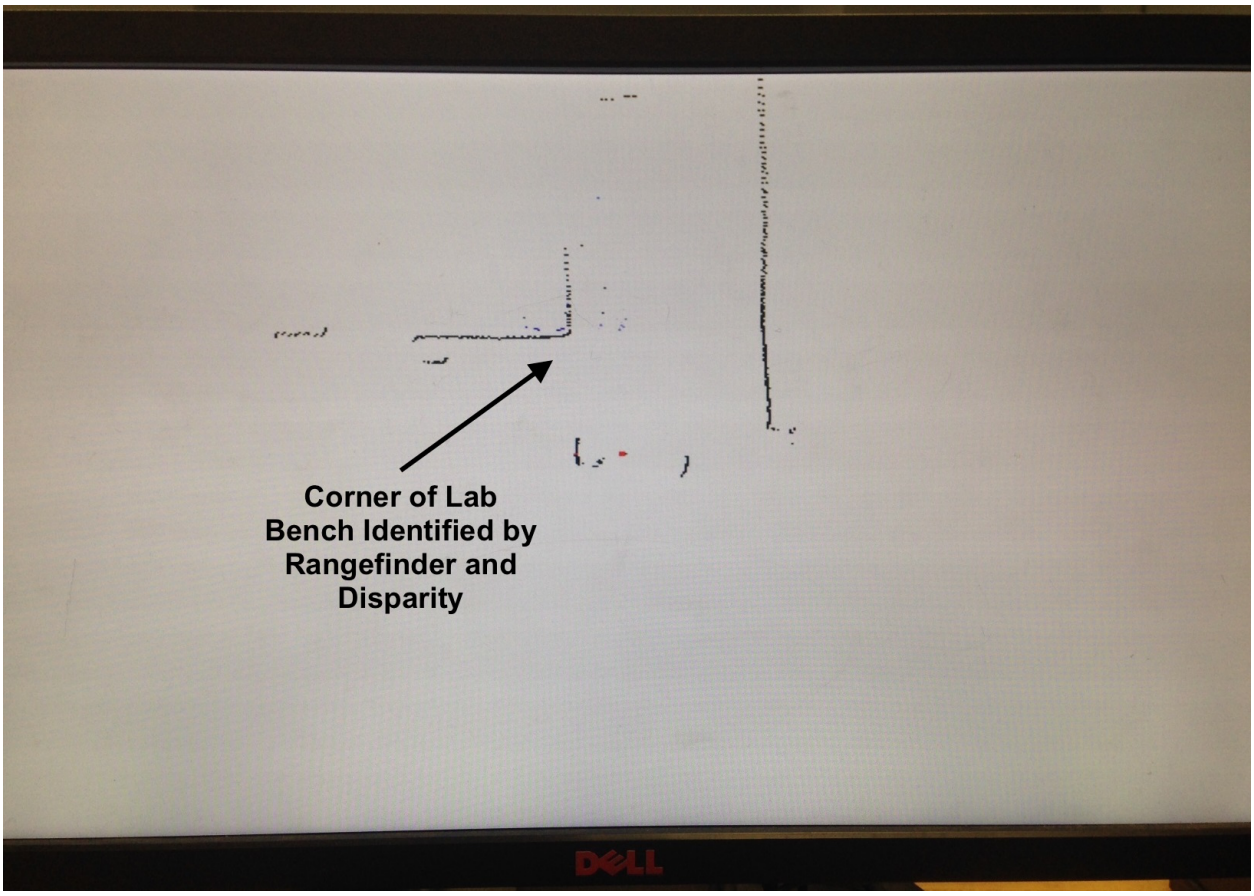


Figure 73: Combined Output Mode

Several steps were taken during the creation of each output mode to aid in hardware debugging. As a basic confirmation of working hardware, the current state of the rangefinder or disparity state machines were passed to the ZedBoard's LEDs. Another important debugging step included an additional output mode that showed the current camera images being used by the disparity algorithm. An example of this mode's output is shown in Figure 74. Note that the image coloration was a result of mapping a monochrome image to arbitrary VGA colors to account for a lack of grayscale color space.

Figure 74: Raw Camera Data Mode

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

6 Conclusions

This project demonstrated a proof of concept SLAM sensor suite as an effective replacement for the simple camera image sensors available on existing remote situational awareness products. Through the use of a stereo camera pair, scanning laser rangefinder, digital compass, and Zynq-7020 All Programmable SoC on a ZedBoard development platform, an all-in-one sensor suite was developed for capturing continuous localization and depth data on its surroundings. This sensor suite supported three main output modes that each displayed different localization information to an attached VGA display.

A compass-referenced real-time 2D floorplan mode was created by using a scanning laser rangefinder to map the objects closest to the sensor suite. The rangefinder and digital compass modules 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 these sensors was combined and written to the FPGA fabric of the Zynq SoC, where a coordinate-axis transformation was used to localize distance data and prepare it for output via VGA display.

As an accompaniment to the 2D floorplan mode, a disparity depth mapping mode was also created. This operating mode relied on a stereo camera pair. In order to minimize costs, a custom printed circuit board was designed and built that connected two camera modules and video frame buffer ICs to the FPGA fabric of the Zynq SoC by plugging it into two of the Pmod ports. Using this interface, programmable logic was able to read image data from the circuit board's frame buffers and trigger new image captures. After acquiring stereo camera image data, a Sum of Absolute Differences block matching algorithm was used to calculate depth measurements on a pixel by pixel basis throughout the scene. This depth information was then used to create a 3D disparity image.

A final output mode was included to incorporate slices of disparity data into the 2D floorplan. Since the rangefinder and stereo cameras shared a horizontal viewing plane, there were some distinguishable overlaps in sensor data. These overlaps were correlated by filtering

several horizontal lines of depth information from the disparity algorithm and combining them with 2D rangefinder data at the output stage. The resultant floorplan matched stereo camera depth data with rangefinder distance data, and allowed for compensation between measurement methods.

Through its different output modes, this sensor suite demonstrated that FPGA-based data processing was a viable replacement for the simple imaging sensors of existing remote situational awareness products, and serves as an excellent platform for future development.

6.1 Future Work

The utility of this sensor suite could be increased by adding additional image processing algorithms for human recognition and object detection. In a first responder situation, human recognition could be used to provide potentially life-saving information about where persons of interest are located. This feature could be added through the use of Xilinx's High Level Synthesis environment, which would allow for the creation of programmable hardware from a powerful C/C++ image processing library such as Open Computer Vision (OpenCV).

A simpler improvement could be to incorporate all available inertial displacement data from the PmodNAV IMU into the device's processing pipeline. With inertial displacement data completely integrated, this sensor suite would have the capability to produce a more sophisticated 2D map showing the complete path traversed by the device. This functionality would allow for a more complete understanding of the environment around the device.

One other important step in the future of this design would be to combine all hardware used onto a single platform such as a printed circuit board. This board would contain a Zynq chip, an onboard IMU, and mounted stereo camera and rangefinder hardware. With the creation of a customized sensor board, the device could then be added to existing robotic platforms for field testing.

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] Bounce Imaging, *Tactical Throwable Cameras*. Available from: http://www.bounceimaging.com.

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

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

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

[11] 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.

[12] Endeavor Robotics, *110 FirstLook®*. Available from: http://endeavorrobotics.com/products#110-firstlook.

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

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

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

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

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

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

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

[20] 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.

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

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

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

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

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

[26] 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`.

[27] 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`.

[28] 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.

[29] 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.

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

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

[32] 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`.

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

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

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

[36] Xilinx. *ZedBoard Zynq-7000 ARM/FPGA SoC Development Board*. 2017. Available from: `https://www.xilinx.com/products/boards-and-kits/1-elhabt.html`.

Appendix

A 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

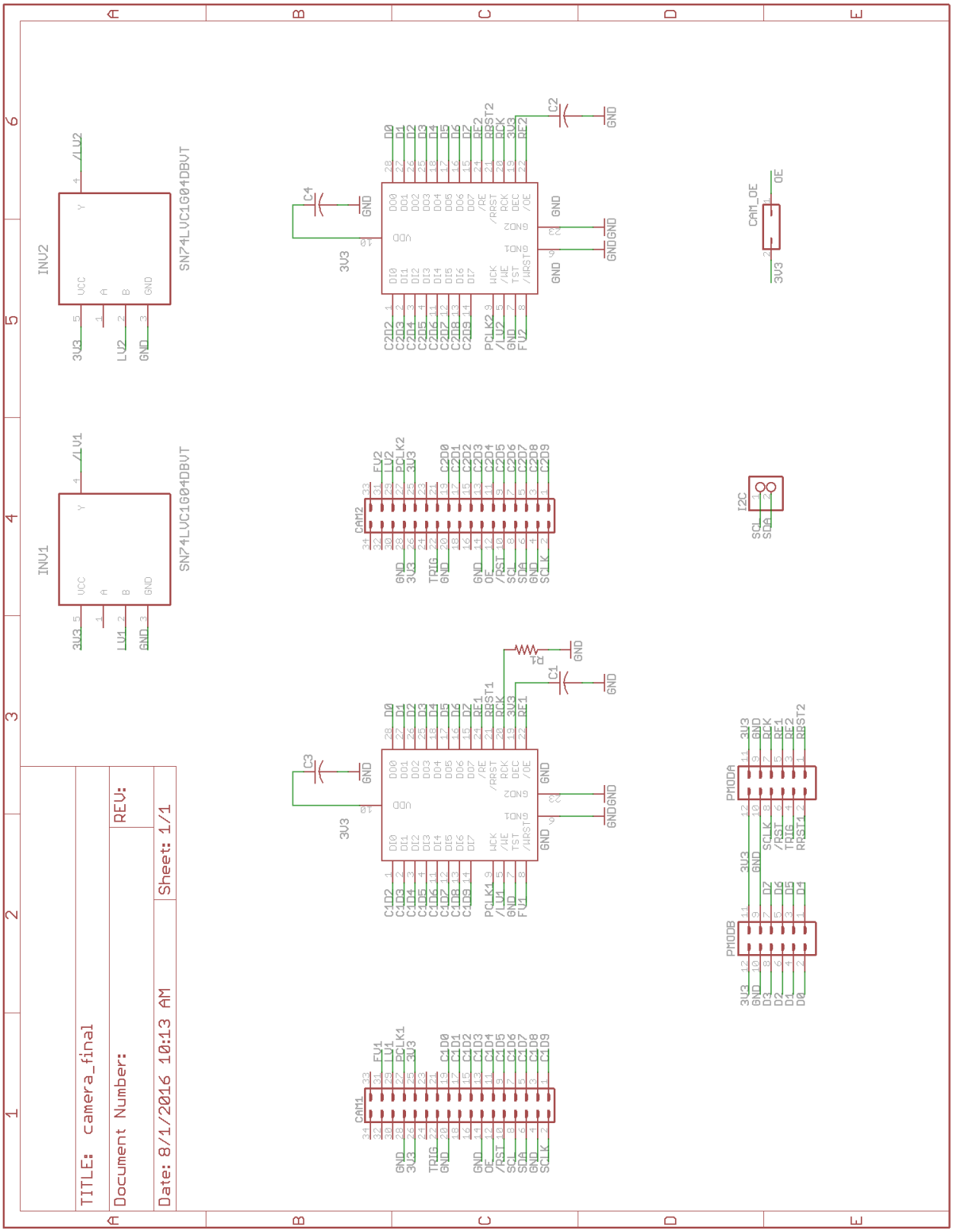
[†] Note that we originally planned to purchase a Flir Lepton thermal camera module and accompanying breakout board to support two stereo OV7670 camera modules. After experimenting with the OV7670 camera module on our FPGA board, we realized that these camera modules are highly limited due to their low frame rate and poor documentation, and decided 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 believed that we were much better off spending our money on better camera modules that were more usable for our task. For more information see Section 4.8.1.

B 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 [26]

C Stereo Camera Schematic



D Code

D.i MT9V034 and Al422b Test Code

Top Module:

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

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

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

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

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

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

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

endmodule
```

Local Data Buffer:

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

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

```
                        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;
```

```
27
28      XIOModule gpi;
29      XIOModule gpo;
30
31      // GPI1 = pixel_value(7:0)
32      XIOModule_Initialize(&gpi, XPAR_IOMODULE_0_DEVICE_ID);
33      XIOModule_Start(&gpi);
34
35      // GPO1 = (GETDATA)|(RRST)|(LED)
36      XIOModule_Initialize(&gpo, XPAR_IOMODULE_0_DEVICE_ID);
37      XIOModule_Start(&gpo);
38
39      print("\n\rMT9V034 controller and AL422B FIFO reader\n\r");
40      while(1)
41      {
42          // get switch position
43          GPI2 = XIOModule_DiscreteRead(&gpi,2);
44
45          if((GPI2&SW_READ)!=0){
46              swState = 1;
47              if (row >= 480) GPO1 &=~(LED);
48              else GPO1 |= LED;
49              GPO1 &=~(RRST|GETDATA);
50              XIOModule_DiscreteWrite(&gpo,1,GPO1);
51          }else{
52              GPO1 &= ~(RRST|LED|GETDATA);
53              XIOModule_DiscreteWrite(&gpo,1,GPO1);
54              row = 0;
55              swState = 0;
56          }
57
58          // code below runs only once based on SW state change
59          if (prevState != swState){
60              if(swState){
61                  print("\n\rReading from FIFO...\n\r");
62
63                  GPO1 |= (RRST); // reset FIFO position to 0th index
64                  GPO1 &=~ (GETDATA); // make sure we're not trying to read data
65                  XIOModule_DiscreteWrite(&gpo,1,GPO1);
66                  pixel_position = 0;
67                  GPO1 &=~(RRST);
68                  XIOModule_DiscreteWrite(&gpo,1,GPO1);
69                  GPO1 |= (GETDATA); // make sure we're not trying to read data
70                  XIOModule_DiscreteWrite(&gpo,1,GPO1);
71
72                  u32 pixelsRead = 0;
73
74                  while(row<480){
75                      // make sure read_sw hasn't been turned off
76                      GPI2 = XIOModule_DiscreteRead(&gpi,2);
77                      if ((GPI2&SW_READ)==0) break;
78
79                      u8 i=0;
80                      // check to see if BUF_READY is good to go
81                      GPI2 = XIOModule_DiscreteRead(&gpi,2);
82                      // wait until it is
83                      while((GPI2&BUF_READY)==0){
84                          if(i==0){
85                              i++;
86                              //print("\n\t buffer not ready \n\r");
87                          }
88                          GPI2 = XIOModule_DiscreteRead(&gpi,2);
89                      }
90                      GPO1 &=~ (GETDATA); // make sure we're not trying to read data
91                      XIOModule_DiscreteWrite(&gpo,1,GPO1);
92
93                      while (pixel_position < 752){
94                          // update pixel position for FPGA buffer
```

```
                        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
```

```
26        ERRNUM = ERRNUM + 1;
27        fprintf('Error #%d: Line %d contains no data\n\r',ERRNUM,LINENUM)
28    end
29    if XPOS<752 % update pixel x position
30        XPOS = XPOS + 1;
31    else % update pixel y position
32        XPOS = 1;
33        YPOS = YPOS + 1;
34    end
35    if mod(LINENUM,36096)==0 % update the loading bar every so often
36      waitbar(LINENUM /360000);
37    end
38    LINENUM = LINENUM + 1; % current line in file (for debug)
39  end
40
41  % display the image...
42  figure, imshow(image);
43  % ... also save the image to a file, overwrite if already saved
44  [path,name,ext] = fileparts(FILENAME);
45  imgname = strcat(name,'.png');
46  if (exist(imgname, 'file') == 2)
47      fprintf('File for image already exists... overwriting it\n\r')
48      delete(imgname);
49  end
50  saveas(gcf,imgname);
51  fprintf('Saved figure to image %s\n\r',imgname);
52
53  % close the file and waitbar before exit
54  fclose(fid);
55  close(h)
```

D.ii Custom IP

Top Module:

```
1
2  `timescale 1 ns / 1 ps
3
4          module custom_logic_v1_0 #
5          (
6                  // Users to add parameters here
7
8                  // User parameters ends
9                  // Do not modify the parameters beyond this line
10
11
12                 // Parameters of Axi Slave Bus Interface S00_AXI
13                 parameter integer C_S00_AXI_DATA_WIDTH  = 32,
14                 parameter integer C_S00_AXI_ADDR_WIDTH  = 4
15         )
16         (
17                 // Users to add ports here
18
19         //physical pins
20         input wire fpga_clk,
21         input wire reset,
22         input wire button,
23         input wire [7:0] sw,
24         output wire [7:0] leds,
25         output wire hsync,
26         output wire vsync,
27         output wire [11:0] rgb,
28
29         // cameras
30         input wire cam_rst, // button for camera RESET_BAR
31         output wire cam_sysclk, // sysclk out to camera
32         output wire cam_reset, // reset_bar out to camera
33         output wire cam_trigger, // trigger out to camera
34         input wire [7:0] FIFO_DATA, // DO[7:0] from AL422b fifo
35         output wire FIFO_OE1, // read enable to fifo (active low)
36         output wire FIFO_RRST1, // read reset to fifo (active low)
37         output wire FIFO_OE2, // read enable to fifo (active low)
38         output wire FIFO_RRST2, // read reset to fifo (active low)
39         output wire FIFO_RCK, // rck to fifo (1MHz)
40
41         //rangefinder BRAM
42         output wire [7:0] addra1,
43         input wire [12:0] coord1_data,
44         output wire clk_100M,
45         output wire [7:0] addra2,
46         input wire [12:0] coord2_data,
47         //output wire clk_100M2,
48
49         //vga map BRAM
50         output wire [18:0] vga_waddr,
51         //output wire clk_100M3,
52         output wire [7:0] dina,
53         output wire ena,
54         output wire wea,
55
56         output wire [18:0] vga_raddr,
57         output wire clk_25M,
58         input wire [7:0] x_vga,
59         output wire enb,
60
61                 // User ports ends
62                 // Do not modify the ports beyond this line
```

```
63
64                  // Ports of Axi Slave Bus Interface S00_AXI
65                  input wire  s00_axi_aclk,
66                  input wire  s00_axi_aresetn,
67                  input wire [C_S00_AXI_ADDR_WIDTH-1 : 0] s00_axi_awaddr,
68                  input wire [2 : 0] s00_axi_awprot,
69                  input wire  s00_axi_awvalid,
70                  output wire  s00_axi_awready,
71                  input wire [C_S00_AXI_DATA_WIDTH-1 : 0] s00_axi_wdata,
72                  input wire [(C_S00_AXI_DATA_WIDTH/8)-1 : 0] s00_axi_wstrb,
73                  input wire  s00_axi_wvalid,
74                  output wire  s00_axi_wready,
75                  output wire [1 : 0] s00_axi_bresp,
76                  output wire  s00_axi_bvalid,
77                  input wire  s00_axi_bready,
78                  input wire [C_S00_AXI_ADDR_WIDTH-1 : 0] s00_axi_araddr,
79                  input wire [2 : 0] s00_axi_arprot,
80                  input wire  s00_axi_arvalid,
81                  output wire  s00_axi_arready,
82                  output wire [C_S00_AXI_DATA_WIDTH-1 : 0] s00_axi_rdata,
83                  output wire [1 : 0] s00_axi_rresp,
84                  output wire  s00_axi_rvalid,
85                  input wire  s00_axi_rready
86          );
87
88      //processing system
89      wire [27:0] data_enable_step;
90      wire transmit;
91
92  // Instantiation of Axi Bus Interface S00_AXI
93          custom_logic_v1_0_S00_AXI # (
94                  .C_S_AXI_DATA_WIDTH(C_S00_AXI_DATA_WIDTH),
95                  .C_S_AXI_ADDR_WIDTH(C_S00_AXI_ADDR_WIDTH)
96          ) custom_logic_v1_0_S00_AXI_inst (
97              .data_enable_step(data_enable_step),
98              .transmit(transmit),
99                  .S_AXI_ACLK(s00_axi_aclk),
100                 .S_AXI_ARESETN(s00_axi_aresetn),
101                 .S_AXI_AWADDR(s00_axi_awaddr),
102                 .S_AXI_AWPROT(s00_axi_awprot),
103                 .S_AXI_AWVALID(s00_axi_awvalid),
104                 .S_AXI_AWREADY(s00_axi_awready),
105                 .S_AXI_WDATA(s00_axi_wdata),
106                 .S_AXI_WSTRB(s00_axi_wstrb),
107                 .S_AXI_WVALID(s00_axi_wvalid),
108                 .S_AXI_WREADY(s00_axi_wready),
109                 .S_AXI_BRESP(s00_axi_bresp),
110                 .S_AXI_BVALID(s00_axi_bvalid),
111                 .S_AXI_BREADY(s00_axi_bready),
112                 .S_AXI_ARADDR(s00_axi_araddr),
113                 .S_AXI_ARPROT(s00_axi_arprot),
114                 .S_AXI_ARVALID(s00_axi_arvalid),
115                 .S_AXI_ARREADY(s00_axi_arready),
116                 .S_AXI_RDATA(s00_axi_rdata),
117                 .S_AXI_RRESP(s00_axi_rresp),
118                 .S_AXI_RVALID(s00_axi_rvalid),
119                 .S_AXI_RREADY(s00_axi_rready)
120         );
121
122         // Add user logic here
123
124     mqp_top mqp_top
125     (
126         //physical pins
127         .fpga_clk(fpga_clk),
128         .reset(reset),
129         .button(button),
130         .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,
```

```
20                  // User ports ends
21                  // Do not modify the ports beyond this line
22
23                  // Global Clock Signal
24                  input wire  S_AXI_ACLK,
25                  // Global Reset Signal. This Signal is Active LOW
26                  input wire  S_AXI_ARESETN,
27                  // Write address (issued by master, acceped by Slave)
28                  input wire [C_S_AXI_ADDR_WIDTH-1 : 0] S_AXI_AWADDR,
29                  // Write channel Protection type. This signal indicates the
30                  // privilege and security level of the transaction, and whether
31                  // the transaction is a data access or an instruction access.
32                  input wire [2 : 0] S_AXI_AWPROT,
33                  // Write address valid. This signal indicates that the master signaling
34                  // valid write address and control information.
35                  input wire  S_AXI_AWVALID,
36                  // Write address ready. This signal indicates that the slave is ready
37                  // to accept an address and associated control signals.
38                  output wire  S_AXI_AWREADY,
39                  // Write data (issued by master, acceped by Slave)
40                  input wire [C_S_AXI_DATA_WIDTH-1 : 0] S_AXI_WDATA,
41                  // Write strobes. This signal indicates which byte lanes hold
42                  // valid data. There is one write strobe bit for each eight
43                  // bits of the write data bus.
44                  input wire [(C_S_AXI_DATA_WIDTH/8)-1 : 0] S_AXI_WSTRB,
45                  // Write valid. This signal indicates that valid write
46                  // data and strobes are available.
47                  input wire  S_AXI_WVALID,
48                  // Write ready. This signal indicates that the slave
49                  // can accept the write data.
50                  output wire  S_AXI_WREADY,
51                  // Write response. This signal indicates the status
52                  // of the write transaction.
53                  output wire [1 : 0] S_AXI_BRESP,
54                  // Write response valid. This signal indicates that the channel
55                  // is signaling a valid write response.
56                  output wire  S_AXI_BVALID,
57                  // Response ready. This signal indicates that the master
58                  // can accept a write response.
59                  input wire  S_AXI_BREADY,
60                  // Read address (issued by master, acceped by Slave)
61                  input wire [C_S_AXI_ADDR_WIDTH-1 : 0] S_AXI_ARADDR,
62                  // Protection type. This signal indicates the privilege
63                  // and security level of the transaction, and whether the
64                  // transaction is a data access or an instruction access.
65                  input wire [2 : 0] S_AXI_ARPROT,
66                  // Read address valid. This signal indicates that the channel
67                  // is signaling valid read address and control information.
68                  input wire  S_AXI_ARVALID,
69                  // Read address ready. This signal indicates that the slave is
70                  // ready to accept an address and associated control signals.
71                  output wire  S_AXI_ARREADY,
72                  // Read data (issued by slave)
73                  output wire [C_S_AXI_DATA_WIDTH-1 : 0] S_AXI_RDATA,
74                  // Read response. This signal indicates the status of the
75                  // read transfer.
76                  output wire [1 : 0] S_AXI_RRESP,
77                  // Read valid. This signal indicates that the channel is
78                  // signaling the required read data.
79                  output wire  S_AXI_RVALID,
80                  // Read ready. This signal indicates that the master can
81                  // accept the read data and response information.
82                  input wire  S_AXI_RREADY
83              );
84
85      reg [31:0] extra_data_enable_step;
86
87              // AXI4LITE signals
```

```
88          reg [C_S_AXI_ADDR_WIDTH-1 : 0]  axi_awaddr;
89          reg     axi_awready;
90          reg     axi_wready;
91          reg [1 : 0]     axi_bresp;
92          reg     axi_bvalid;
93          reg [C_S_AXI_ADDR_WIDTH-1 : 0]  axi_araddr;
94          reg     axi_arready;
95          reg [C_S_AXI_DATA_WIDTH-1 : 0]  axi_rdata;
96          reg [1 : 0]     axi_rresp;
97          reg     axi_rvalid;
98
99          // Example-specific design signals
100         // local parameter for addressing 32 bit / 64 bit C_S_AXI_DATA_WIDTH
101         // ADDR_LSB is used for addressing 32/64 bit registers/memories
102         // ADDR_LSB = 2 for 32 bits (n downto 2)
103         // ADDR_LSB = 3 for 64 bits (n downto 3)
104         localparam integer ADDR_LSB = (C_S_AXI_DATA_WIDTH/32) + 1;
105         localparam integer OPT_MEM_ADDR_BITS = 1;
106         //------------------------------------------------
107         //-- Signals for user logic register space example
108         //------------------------------------------------
109         //-- Number of Slave Registers 4
110         reg [C_S_AXI_DATA_WIDTH-1:0]    slv_reg0;
111         reg [C_S_AXI_DATA_WIDTH-1:0]    slv_reg1;
112         reg [C_S_AXI_DATA_WIDTH-1:0]    slv_reg2;
113         reg [C_S_AXI_DATA_WIDTH-1:0]    slv_reg3;
114         wire     slv_reg_rden;
115         wire     slv_reg_wren;
116         reg [C_S_AXI_DATA_WIDTH-1:0]     reg_data_out;
117         integer  byte_index;
118
119         // I/O Connections assignments
120
121         assign S_AXI_AWREADY    = axi_awready;
122         assign S_AXI_WREADY     = axi_wready;
123         assign S_AXI_BRESP      = axi_bresp;
124         assign S_AXI_BVALID     = axi_bvalid;
125         assign S_AXI_ARREADY    = axi_arready;
126         assign S_AXI_RDATA      = axi_rdata;
127         assign S_AXI_RRESP      = axi_rresp;
128         assign S_AXI_RVALID     = axi_rvalid;
129         // Implement axi_awready generation
130         // axi_awready is asserted for one S_AXI_ACLK clock cycle when both
131         // S_AXI_AWVALID and S_AXI_WVALID are asserted. axi_awready is
132         // de-asserted when reset is low.
133
134         always @( posedge S_AXI_ACLK )
135         begin
136           if ( S_AXI_ARESETN == 1'b0 )
137             begin
138               axi_awready <= 1'b0;
139             end
140           else
141             begin
142               if (~axi_awready && S_AXI_AWVALID && S_AXI_WVALID)
143                 begin
144                   // slave is ready to accept write address when
145                   // there is a valid write address and write data
146                   // on the write address and data bus. This design
147                   // expects no outstanding transactions.
148                   axi_awready <= 1'b1;
149                 end
150               else
151                 begin
152                   axi_awready <= 1'b0;
153                 end
154             end
155         end
```

```
156
157         // Implement axi_awaddr latching
158         // This process is used to latch the address when both
159         // S_AXI_AWVALID and S_AXI_WVALID are valid.
160
161         always @( posedge S_AXI_ACLK )
162         begin
163           if ( S_AXI_ARESETN == 1'b0 )
164             begin
165               axi_awaddr <= 0;
166             end
167           else
168             begin
169               if (~axi_awready && S_AXI_AWVALID && S_AXI_WVALID)
170                 begin
171                   // Write Address latching
172                   axi_awaddr <= S_AXI_AWADDR;
173                 end
174             end
175         end
176
177         // Implement axi_wready generation
178         // axi_wready is asserted for one S_AXI_ACLK clock cycle when both
179         // S_AXI_AWVALID and S_AXI_WVALID are asserted. axi_wready is
180         // de-asserted when reset is low.
181
182         always @( posedge S_AXI_ACLK )
183         begin
184           if ( S_AXI_ARESETN == 1'b0 )
185             begin
186               axi_wready <= 1'b0;
187             end
188           else
189             begin
190               if (~axi_wready && S_AXI_WVALID && S_AXI_AWVALID)
191                 begin
192                   // slave is ready to accept write data when
193                   // there is a valid write address and write data
194                   // on the write address and data bus. This design
195                   // expects no outstanding transactions.
196                   axi_wready <= 1'b1;
197                 end
198               else
199                 begin
200                   axi_wready <= 1'b0;
201                 end
202             end
203         end
204
205         // Implement memory mapped register select and write logic generation
206         // The write data is accepted and written to memory mapped registers when
207         // axi_awready, S_AXI_WVALID, axi_wready and S_AXI_WVALID are asserted. Write
                strobes are used to
208         // select byte enables of slave registers while writing.
209         // These registers are cleared when reset (active low) is applied.
210         // Slave register write enable is asserted when valid address and data are available
211         // and the slave is ready to accept the write address and write data.
212         assign slv_reg_wren = axi_wready && S_AXI_WVALID && axi_awready && S_AXI_AWVALID;
213
214         always @( posedge S_AXI_ACLK )
215         begin
216           if ( S_AXI_ARESETN == 1'b0 )
217             begin
218               slv_reg0 <= 0;
219               slv_reg1 <= 0;
220               slv_reg2 <= 0;
221               slv_reg3 <= 0;
222             end
```

```
223             else begin
224               if (slv_reg_wren)
225                 begin
226                   case ( axi_awaddr[ADDR_LSB+OPT_MEM_ADDR_BITS:ADDR_LSB] )
227                     2'h0:
228                       for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1; byte_index
                                = byte_index+1 )
229                         if ( S_AXI_WSTRB[byte_index] == 1 ) begin
230                           // Respective byte enables are asserted as per write strobes
231                           // Slave register 0
232                           slv_reg0[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];
233                         end
234                     2'h1:
235                       for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1; byte_index
                                = byte_index+1 )
236                         if ( S_AXI_WSTRB[byte_index] == 1 ) begin
237                           // Respective byte enables are asserted as per write strobes
238                           // Slave register 1
239                           extra_data_enable_step[(byte_index*8) +: 8] <= S_AXI_WDATA[(
                                  byte_index*8) +: 8];
240                         end
241                     2'h2:
242                       for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1; byte_index
                                = byte_index+1 )
243                         if ( S_AXI_WSTRB[byte_index] == 1 ) begin
244                           // Respective byte enables are asserted as per write strobes
245                           // Slave register 2
246                           slv_reg2[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];
247                         end
248                     2'h3:
249                       for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1; byte_index
                                = byte_index+1 )
250                         if ( S_AXI_WSTRB[byte_index] == 1 ) begin
251                           // Respective byte enables are asserted as per write strobes
252                           // Slave register 3
253                           slv_reg3[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];
254                         end
255                     default : begin
256                                 slv_reg0 <= slv_reg0;
257                                 slv_reg1 <= slv_reg1;
258                                 slv_reg2 <= slv_reg2;
259                                 slv_reg3 <= slv_reg3;
260                               end
261                   endcase
262                 end
263             end
264           end
265
266           // Implement write response logic generation
267           // The write response and response valid signals are asserted by the slave
268           // when axi_wready, S_AXI_WVALID, axi_wready and S_AXI_WVALID are asserted.
269           // This marks the acceptance of address and indicates the status of
270           // write transaction.
271
272           always @( posedge S_AXI_ACLK )
273           begin
274             if ( S_AXI_ARESETN == 1'b0 )
275               begin
276                 axi_bvalid  <= 0;
277                 axi_bresp   <= 2'b0;
278               end
279             else
280               begin
281                 if (axi_awready && S_AXI_AWVALID && ~axi_bvalid && axi_wready && S_AXI_WVALID)
282                   begin
283                     // indicates a valid write response is available
284                     axi_bvalid <= 1'b1;
285                     axi_bresp  <= 2'b0; // 'OKAY' response
```

```
286                 end                   // work error responses in future
287               else
288                 begin
289                   if (S_AXI_BREADY && axi_bvalid)
290                     //check if bready is asserted while bvalid is high)
291                     //(there is a possibility that bready is always asserted high)
292                     begin
293                       axi_bvalid <= 1'b0;
294                     end
295                 end
296             end
297         end
298 
299         // Implement axi_arready generation
300         // axi_arready is asserted for one S_AXI_ACLK clock cycle when
301         // S_AXI_ARVALID is asserted. axi_awready is
302         // de-asserted when reset (active low) is asserted.
303         // The read address is also latched when S_AXI_ARVALID is
304         // asserted. axi_araddr is reset to zero on reset assertion.
305 
306         always @( posedge S_AXI_ACLK )
307         begin
308           if ( S_AXI_ARESETN == 1'b0 )
309             begin
310               axi_arready <= 1'b0;
311               axi_araddr  <= 32'b0;
312             end
313           else
314             begin
315               if (~axi_arready && S_AXI_ARVALID)
316                 begin
317                   // indicates that the slave has acceped the valid read address
318                   axi_arready <= 1'b1;
319                   // Read address latching
320                   axi_araddr  <= S_AXI_ARADDR;
321                 end
322               else
323                 begin
324                   axi_arready <= 1'b0;
325                 end
326             end
327         end
328 
329         // Implement axi_arvalid generation
330         // axi_rvalid is asserted for one S_AXI_ACLK clock cycle when both
331         // S_AXI_ARVALID and axi_arready are asserted. The slave registers
332         // data are available on the axi_rdata bus at this instance. The
333         // assertion of axi_rvalid marks the validity of read data on the
334         // bus and axi_rresp indicates the status of read transaction.axi_rvalid
335         // is deasserted on reset (active low). axi_rresp and axi_rdata are
336         // cleared to zero on reset (active low).
337         always @( posedge S_AXI_ACLK )
338         begin
339           if ( S_AXI_ARESETN == 1'b0 )
340             begin
341               axi_rvalid <= 0;
342               axi_rresp  <= 0;
343             end
344           else
345             begin
346               if (axi_arready && S_AXI_ARVALID && ~axi_rvalid)
347                 begin
348                   // Valid read data is available at the read data bus
349                   axi_rvalid <= 1'b1;
350                   axi_rresp  <= 2'b0; // 'OKAY' response
351                 end
352               else if (axi_rvalid && S_AXI_RREADY)
353                 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)
```

```
14      input enable, // PL data processing enable (from PS/PL AXI)
15      input [10:0] step, // current step offset (0-768) from PS/PL AXI
16      output reg [18:0] vga_waddr, // write address to VGA logic
17      output reg transmit, // enable new data capture (to PS via AXI)
18      output reg [7:0] dina, // VGA output buffer pixel data
19      output reg wea, // VGA output buffer write enable
20
21      output reg [7:0] addra1, // 0-45 degree ROM lookup table address
22      input [12:0] coord1_data, // 0-45 degree ROM lookup table data
23
24      output reg [7:0] addra2, // 45-90 degree ROM lookup table address
25      input [12:0] coord2_data // 45-90 degree ROM lookup table address
26      );
27
28      reg [15:0] watchdog; // timeout to begin new data capture sequence
29      reg [15:0] decoded; // stores 2 bytes of rangefinder data as (MSB -> LSB)-0x30
30      reg [24:0] xmult, ymult; // rangefinder data point x,y location relative to (0,0)
31      wire xneg, yneg; // flag for indicating if x,y data is positive or negative
32      reg [9:0] xlocation; // rangefinder data point x location relative to device_x
33      reg [8:0] ylocation; // rangefinder data point y location relative to device_y
34
35      reg [2:0] offset_counter; // counter to wait for multiply operation in OFFSET state
36      reg [1:0] address_counter; // counter to wait for multiply operation in ADDRESS state
37
38      wire [18:0] vga_product; // current pixel row for output address
39
40      wire [24:0] product1, product2; // x/y offset output from trig lookup / multiplier
            conversion
41
42      // FSM states, next state logic
43      parameter [2:0] IDLE = 0, CLEAR = 1, DECODE = 2, OFFSET = 3, SCALE = 4, ADDRESS = 5,
            WRITE = 6;
44      reg [2:0] current_state, next_state;
45
46      reg [10:0] row_count; // 0-480
47      reg [10:0] col_count; // 0-640
48      reg [8:0] step_count = 9'd0;
49  //----------------------------------------------------------
50  //data processing
51      //sets flag to indicate negative horizontal value on circle
52      assign xneg = (step >= 0 && step <= 128)  ? 1'b0 :
53                    (step > 128 && step <= 384) ? 1'b0 :
54                    (step > 384 && step <= 640) ? 1'b1 : //1'b1;
55                    (step > 640 && step <= 768) ? 1'b1 :
56                    (step > 768 && step <= 896) ? 1'b1 : 1'b0;
57
58      //sets flag to indicate negative vertical value on circle
59      assign yneg = (step >= 0 && step <= 128)  ? 1'b1 :
60                    (step > 128 && step <= 384) ? 1'b0 :
61                    (step > 384 && step <= 640) ? 1'b0 : //1'b1;
62                    (step > 640 && step <= 768) ? 1'b1 :
63                    (step > 768 && step <= 896) ? 1'b1 : 1'b1;
64
65      // State machine overhead control
66      always @ (posedge clk)
67          if (reset)
68              current_state <= IDLE;
69          else
70              current_state <= next_state;
71
72      //next state logic
73      always @ (current_state, offset_counter, address_counter, enable, row_count, col_count,
            button, step)
74      begin
75          case (current_state)
76              // stays in IDLE until a new enable pulse
77              IDLE:
78                  if(enable)
```

```
79                      next_state = DECODE;
80                  else if (button) // was (button || step == 768)
81                      next_state = CLEAR;
82                  else
83                      next_state = IDLE;
84
85              CLEAR: // clear pixel data from previous run
86                  if(row_count == 479 && col_count == 639)
87                      next_state = IDLE;
88                  else
89                      next_state = CLEAR;
90              // stays in DECODE for one clock cycle
91              DECODE:
92                  next_state = OFFSET;
93
94              // stays in OFFSET for a multiply operation
95              OFFSET:
96                  if(offset_counter == 4)
97                      next_state = SCALE;
98                  else
99                      next_state = OFFSET;
100
101             // stays in SCALE for one clock cycle
102             SCALE:
103                 next_state = ADDRESS;
104
105             // stays in ADDRESS for a multiply operation
106             ADDRESS:
107                 if(address_counter == 3)
108                     next_state = WRITE;
109                 else
110                     next_state = ADDRESS;
111
112             // stays in ADDRESS for one clock cycle
113             WRITE:
114                 next_state = IDLE;
115
116             default:
117                 next_state = IDLE;
118         endcase
119     end
120
121     //state machine
122     always @ (posedge clk)
123     begin
124         //resets registers, waits for data to process
125         if(current_state == IDLE)
126         begin
127             wea <= 1'b0;
128             dina <= 8'h00;
129             row_count <= 11'd0;
130             col_count <= 11'd0;
131             if(step_count == 9'd767)
132                 step_count <= 9'd0;
133         end
134
135         // clear previous set of data from BRAM
136         else if(current_state == CLEAR)
137         begin
138             vga_waddr <= (640*row_count) + col_count;
139             wea <= 1'b1;
140             dina <= 8'h00;
141             if(col_count < 639)
142                 col_count <= col_count + 1'b1;
143             else if(col_count == 639 && row_count < 479)
144                 begin
145                 col_count <= 11'd0;
146                 row_count <= row_count + 1'b1;
```

```
147                 end
148
149         end
150         //calculates adresses for trig LUTs
151         //decodes data by substracting 0x30 from both halves of data point
152         else if(current_state == DECODE)
153         begin
154             step_count <= step_count + 1'b1;
155
156             wea <= 1'b0;
157
158             addra1 <= (step >= 0 && step <= 128)  ? 128-step :  //index 128 to 0  - Q1 - hz
159                       (step > 128 && step <= 256) ? step-128 :  //index 1 to 128  - Q2 - hz
160                       (step > 256 && step <= 384) ? 384-step :  //index 127 to 0  - Q2 - vr
161                       (step > 384 && step <= 512) ? step-384 :  //index 1 to 128  - Q3 - vr
162                       (step > 512 && step <= 640) ? 640-step :  //index 127 to 0  - Q3 - hz
163                       (step > 640 && step <= 768) ? step-640 :  //index 1 to 128  - Q4 - hz
164                       (step > 768 && step <= 896) ? 896-step :  //index 127 to 0  - Q4 - vr
165                       //(step > 896 && step <= 1023) ? step - 896 :
166                                                     step-896;   //index 0 to 128  - Q1 - vr
167
168             addra2 <= (step >= 0 && step <= 128)  ? step :      //index 128 to 256 - Q1 - vr
169                       (step > 128 && step <= 256) ? 256-step :  //index 255 to 128 - Q2 - vr
170                       (step > 256 && step <= 384) ? step-256 :  //index 129 to 256 - Q2 - hz
171                       (step > 384 && step <= 512) ? 512-step :  //index 255 to 128 - Q3 - hz
172                       (step > 512 && step <= 640) ? step-512 :  //index 129 to 256 - Q3 - vr
173                       (step > 640 && step <= 768) ? 768-step :
174                       (step > 768 && step <= 896) ? step-768 :
175                       //(step > 896 && step <= 1023) ? 1023 - step :
176                                                     1023-step;   //index 255 to 128 - Q4 -
                                                          vr
177
178
179             decoded <= {data[7:0]-8'h30, data[15:8]-8'h30};
180         end
181
182         //drops upper bit of each data point
183         //calculates horizontal and vertical distance for each data point
184         else if(current_state == OFFSET)
185         begin
186             if(offset_counter == 4)
187             begin
188                 if((step > 256 && step <= 512) || (step > 768 && step <= 1023))
189                 begin
190                     xmult <= product2;
191                     ymult <= product1;
192                 end
193
194                 else
195                 begin
196                     xmult <= product1;
197                     ymult <= product2;
198                 end
199             end
200
201             offset_counter <= offset_counter + 1'b1;
202         end
203
204         //scales data and localizes to device
205         else if(current_state == SCALE)
206         begin
207             offset_counter <= 3'b0;
208
209             if(xneg)
210                 xlocation <= device_x - xmult[23:16];
211             else
212                 xlocation <= device_x + xmult[23:16];
213
```

```
214             if(yneg)
215                 ylocation <= device_y + ymult[23:16];
216             else
217                 ylocation <= device_y - ymult[23:16];
218         end
219 
220         //calculates address for VGA BRAM
221         else if(current_state == ADDRESS)
222         begin
223             if(address_counter == 3)
224                 vga_waddr <= vga_product + xlocation;
225 
226             address_counter <= address_counter + 1'b1;
227         end
228 
229         //writes to BRAM
230         else if(current_state == WRITE)
231         begin
232             address_counter <= 2'b00;
233             dina <= 8'hFF;
234             wea <= 1'b1;
235         end
236     end
237 
238     // convert data point, trig lookup to x or y offset
239     mult_gen_0 trig_mult_1
240     (
241         .CLK(clk),
242         .A({decoded[13:8], decoded[5:0]}),
243         .B(coord1_data),
244         .P(product1)
245     );
246     // convert data point, trig lookup to x or y offset
247     mult_gen_0 trig_mult_2
248     (
249         .CLK(clk),
250         .A({decoded[13:8], decoded[5:0]}),
251         .B(coord2_data),
252         .P(product2)
253     );
254 
255     // calculates 640*row_count portion of VGA write address
256     mult_gen_2 vga_multiplier
257     (
258         .CLK(clk),
259         .A(ylocation),
260         .P(vga_product)
261     );
262 
263     //watchdog counter watches for missed data transfers
264     always @(posedge clk, posedge reset)
265     begin
266         if(reset || transmit)
267             watchdog <= 0;
268         else if (current_state == IDLE)
269             watchdog <= watchdog + 1'b1;
270         else
271             watchdog <= watchdog;
272     end
273 
274     // trigger a new rangefinder data aquisition sequence when the entire screen has been
        cleared
275     always @ (posedge clk)
276     begin
277         if((row_count == 479 && col_count == 639) || step_count == 9'd767)
278             transmit <= 1'b1;
279         else
280             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
```

```
--                    - 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
```

```
126
127     -- output horizontal and vertical counters
128     hcount <= hcounter;
129     vcount <= vcounter;
130
131     -- blank is active when outside screen visible area
132     -- color output should be blacked (put on 0) when blank in active
133     -- blank is delayed one pixel clock period from the video_enable
134     -- signal to account for the pixel pipeline delay.
135     blank <= not video_enable when rising_edge(pixel_clk);
136
137     -- increment horizontal counter at pixel_clk rate
138     -- until HMAX is reached, then reset and keep counting
139     h_count: process(pixel_clk)
140     begin
141        if(rising_edge(pixel_clk)) then
142           if(rst = '1') then
143              hcounter <= (others => '0');
144           elsif(hcounter = HMAX) then
145              hcounter <= (others => '0');
146           else
147              hcounter <= hcounter + 1;
148           end if;
149        end if;
150     end process h_count;
151
152     -- increment vertical counter when one line is finished
153     -- (horizontal counter reached HMAX)
154     -- until VMAX is reached, then reset and keep counting
155     v_count: process(pixel_clk)
156     begin
157        if(rising_edge(pixel_clk)) then
158           if(rst = '1') then
159              vcounter <= (others => '0');
160           elsif(hcounter = HMAX) then
161              if(vcounter = VMAX) then
162                 vcounter <= (others => '0');
163              else
164                 vcounter <= vcounter + 1;
165              end if;
166           end if;
167        end if;
168     end process v_count;
169
170     -- generate horizontal synch pulse
171     -- when horizontal counter is between where the
172     -- front porch ends and the synch pulse ends.
173     -- The HS is active (with polarity SPP) for a total of 96 pixels.
174     do_hs: process(pixel_clk)
175     begin
176        if(rising_edge(pixel_clk)) then
177           if(hcounter >= HFP and hcounter < HSP) then
178              HS <= SPP;
179           else
180              HS <= not SPP;
181           end if;
182        end if;
183     end process do_hs;
184
185     -- generate vertical synch pulse
186     -- when vertical counter is between where the
187     -- front porch ends and the synch pulse ends.
188     -- The VS is active (with polarity SPP) for a total of 2 video lines
189     -- = 2*HMAX = 1600 pixels.
190     do_vs: process(pixel_clk)
191     begin
192        if(rising_edge(pixel_clk)) then
193           if(vcounter >= VFP and vcounter < VSP) then
```

```
194             VS <= SPP;
195         else
196             VS <= not SPP;
197         end if;
198       end if;
199   end process do_vs;
200
201   -- enable video output when pixel is in visible area
202   video_enable <= '1' when (hcounter < HLINES and vcounter < VLINES) else '0';
203
204 end Behavioral;
```

D.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
```

D.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
```

```
63  }UART_STATE;
64
65  // function prototype for camera initialization fxn
66  void init_cams();
67
68  void delay(int cycles);
69  void IMU_init();
70  int getIMUdata();
71  double getCompassHeading();
72  double getStepOffset(double compassHeading);
73
74  int main()
75  {
76      init_platform();
77
78      char echo[11];
79      char status[2];
80      char databuf[65][24];
81      char databufbuf[3];
82      char linefeed[4];
83
84      echo[10] = '\0';
85      databufbuf[2] = '\0';
86
87      Xuint32 *baseaddr_p = (Xuint32 *)XPAR_CUSTOM_LOGIC_S00_AXI_BASEADDR;
88
89      u32 data_enable_step;
90      u32 write;
91      u32 stepbuf;
92
93      char *echotestnl = "G00076801\n";
94      char *echotestlf = "G00076801\r";
95
96      UART_STATE STATE = WAIT;
97
98      // setup LED output
99      XGpioPs_Config * ConfigPtr = XGpioPs_LookupConfig(XPAR_PS7_GPIO_0_DEVICE_ID);
100     XGpioPs_CfgInitialize(&Gpio, ConfigPtr, ConfigPtr->BaseAddr);
101     XGpioPs_SetDirectionPin(&Gpio, 7, 1);
102     // initialize cameras using I2C
103     init_cams();
104
105     int Status;
106     // initialize SPI
107     XSpiPs_Config *SpiConfig;
108     SpiConfig = XSpiPs_LookupConfig(SPI_DEVICE_ID);
109     // initialize the spi hardware with the device config
110     Status = XSpiPs_CfgInitialize(&SpiInstance, SpiConfig, SpiConfig->BaseAddress);
111     // Set the Spi device as a master
112     XSpiPs_SetOptions(&SpiInstance, XSPIPS_MASTER_OPTION | XSPIPS_CLK_PHASE_1_OPTION |
            XSPIPS_FORCE_SSELECT_OPTION);
113     // Set the SPI clock prescaler
114     XSpiPs_SetClkPrescaler(&SpiInstance, XSPIPS_CLK_PRESCALE_256);
115     // initialize the imu
116     IMU_init();
117
118     int deviceStepOffset = 0;
119
120
121     // Continuously run rangefinder state machine after initialization is complete
122     while(1)
123     {
124         switch(STATE)
125         {
126             // Wait until the PL indicates that it wants new rangefinder data
127             case WAIT:
128             {
129                 //blocks until the flag is set (from PL) to initiate data transfer
```

```
130                 while(Xil_In32(baseaddr_p) == 0);
131                 while(getIMUdata()==0);
132                 deviceStepOffset = (int) round(getStepOffset(getCompassHeading()));
133 
134                 STATE = TX_COMMAND;
135                 break;
136             }
137             // Call to the rangedfinder for a new round of data acquisition
138             case TX_COMMAND:
139             {
140                 printf("G00076801\n"); // data acquisition command
141                 STATE = RX_DATA;
142                 break;
143             }
144             // Process incoming data from the rangefinder byte by byte
145             case RX_DATA:
146             {
147                 int rx_index = 0;   // indexes rows
148                 int rx_line = 0;    // indexes columns - counts which data line is being
                        received
149 
150                 // receives echo
151                 for(rx_index = 0; rx_index < 10; rx_index++)
152                 {
153                     echo[rx_index] = inbyte();  // blocking - inbyte is polled
154                 }
155 
156                 // makes sure the echo command is successful
157                 // if not, it repeats the tx_command state
158                 if(strcmp(echo, echotestnl) != 0 && strcmp(echo, echotestlf) != 0)
159                 {
160                     STATE = WAIT;
161                     break;
162                 }
163 
164                 // receives status
165                 for(rx_index = 0; rx_index < 2; rx_index++)
166                 {
167                     status[rx_index] = inbyte();    // blocking - inbyte is polled
168                 }
169 
170                 int iteration = 0;
171 
172                 // receives 24 data blocks
173                 stepbuf = deviceStepOffset; //deviceStepOffset;
174                 for(rx_line = 0; rx_line < 24; rx_line++)
175                 {
176                     iteration = 0;
177                     // receives 65 bytes per block
178                     for(rx_index = 0; rx_index < 65; rx_index++)
179                     {
180                         iteration++;
181                         databuf[rx_line][rx_index] = inbyte();  // blocking - inbyte is
                                polled
182                         write = 0;
183                         databufbuf[1-(iteration%2)] = databuf[rx_line][rx_index];
184 
185                         // process data two chars at a time
186                         if(iteration%2 == 0)
187                         {
188                             //'enables' data in memory when the data is not an error code
189                             if(!(strcmp(databufbuf[0], '0') == 0 && strcmp(databufbuf[1], 'C
                                    ') <= 0))
190                                 write = 1;
191                             stepbuf++;
192                         }
193 
194                         //sends information to the programmable logic for each data point (2
```

```
                            chars)
195                         //in the order of {data, enable, step}
196                         //data_enable_step = (400 << 20) | (400 << 12) | (write << 11) |
                                stepbuf;
197                         data_enable_step = (databufbuf[1] << 20) | (databufbuf[0] << 12) | (
                                write << 11) | stepbuf;
198                         *(baseaddr_p+1) = data_enable_step;
199
200                     }
201                 }
202
203                 iteration = 0;
204                 // account for 4 chars of data after step data has been sent
205                 for(rx_index = 0; rx_index < 4; rx_index++)
206                 {
207                     linefeed[rx_index] = inbyte();  // blocking - inbyte is polled
208                 }
209
210                 echo[0] = '\0';
211
212                 STATE = WAIT;
213                 break;
214             }
215         }
216     }
217     cleanup_platform();
218     return 0;
219 }
220
221 void init_cams()
222 {
223     // number of bytes to be written from write buffer
224     u8 ByteCount = 4;
225     // Create and fill write buffer with ByteCount bytes of data
226     u8 WriteBuffer[sizeof(AddressType) + PAGE_SIZE];
227     WriteBuffer[0] = (u8) (CAM1_ADDR); // camera 1 I2C address
228     WriteBuffer[1] = (u8) (CAMERA_CTL); // camera control register
229     WriteBuffer[2] = (u8) (MANUAL_TRIG >> 8); // 16 bit write value (upper byte)
230     WriteBuffer[3] = (u8) (MANUAL_TRIG); // 16 bit write value (lower byte)
231
232     XIicPs_Config *ConfigPtr;
233     int Status;
234
235     // Look up Zynq-specific IIC device configuration
236     ConfigPtr = XIicPs_LookupConfig(IIC_DEVICE_ID);
237     // Initialize said device-specific iic configuration so the driver is ready for use
238     Status = XIicPs_CfgInitialize(&IicInstance, ConfigPtr, ConfigPtr->BaseAddress);
239     // Run a self test to make sure the driver works (will return XST_SUCCESS if working)
240     Status = XIicPs_SelfTest(&IicInstance);
241     // Set iic SCLK rate (bus now ready for use)
242     XIicPs_SetSClk(&IicInstance, IIC_SCLK_RATE);
243
244     // Initialize Cam1
245     XIicPs_MasterSendPolled(&IicInstance, WriteBuffer, ByteCount,CAM1_ADDR);
246     while(XIicPs_BusIsBusy(&IicInstance));
247
248     // Initialize Cam2
249     WriteBuffer[0] = (u8) (CAM2_ADDR); // change write address to cam2
250     XIicPs_MasterSendPolled(&IicInstance, WriteBuffer, ByteCount,CAM2_ADDR);
251     while(XIicPs_BusIsBusy(&IicInstance));
252
253     // Write LED to indicate finished initialization sequence
254     XGpioPs_WritePin(&Gpio, 7, 0x01);
255 }
256
257 void delay(int cycles)
258 {
259     int i = 0;
```

```
260     while(i<cycles)
261         i++;
262     return;
263 }
264 
265 void IMU_init()
266 {
267     u8 DataBuffer[2]; // addr + 8 bits write val
268     // high power mode
269     DataBuffer[0] = (u8) CTL1 & WRITE;
270     DataBuffer[1] = (u8) 0x7C;
271     // ONLY SET SLAVE SELECT ON THE 1ST TRANSFER
272     XSpiPs_SetSlaveSelect(&SpiInstance,0x01);
273     XSpiPs_PolledTransfer(&SpiInstance, DataBuffer, NULL, 2);
274     delay(500);
275 
276     DataBuffer[0] = (u8) CTL2 & WRITE;
277     DataBuffer[1] = (u8) 0x00;
278     XSpiPs_PolledTransfer(&SpiInstance, DataBuffer, NULL, 2);
279     delay(500);
280 
281     // turn on the device
282     DataBuffer[0] = (u8) CTL3 & WRITE;
283     DataBuffer[1] = (u8) 0x80;
284     XSpiPs_PolledTransfer(&SpiInstance, DataBuffer, NULL, 2);
285     delay(250);
286 
287     return;
288 }
289 
290 int getIMUdata()
291 {
292 
293     int newDataReady = 0;
294 
295     // check status address
296     u8 DataBuffer[2]; // addr + 8 bits read val
297     DataBuffer[0] = (u8) STATUS_ADDRESS | READ;
298     DataBuffer[1] = (u8) 0x00;
299     XSpiPs_PolledTransfer(&SpiInstance, DataBuffer, &DataBuffer[0], 2);
300 
301     // if XY data's available, indicate so and grab it
302     if ((DataBuffer[1] & 0x03) == 0x03) {
303 
304         // grabs x and y magnetometer environmental offset
305         u8 XYoffset[5];
306         XYoffset[0] = (u8) OFFSET_ADDRESS_XLOW | CASCADE | READ;
307         XYoffset[1] = (u8) 0x00;
308         XYoffset[2] = (u8) 0x00;
309         XYoffset[3] = (u8) 0x00;
310         XYoffset[4] = (u8) 0x00;
311         XSpiPs_PolledTransfer(&SpiInstance, XYoffset, &XYoffset[0], 5);
312         xOffset = (XYoffset[2]<<8) | XYoffset[1];
313         yOffset = (XYoffset[4]<<8) | XYoffset[3];
314 
315         // grabs x and y magnetometer data
316         u8 XYdata[5];
317         XYdata[0] = (u8) DATA_ADDRESS | CASCADE | READ;
318         XYdata[1] = (u8) 0x00;
319         XYdata[2] = (u8) 0x00;
320         XYdata[3] = (u8) 0x00;
321         XYdata[4] = (u8) 0x00;
322         XSpiPs_PolledTransfer(&SpiInstance, XYdata, &XYdata[0], 5);
323         xData = (XYdata[2]<<8) | XYdata[1];
324         yData = (XYdata[4]<<8) | XYdata[3];
325 
326 
327         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;
}
```

D.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.
```

```
63          % e.g., for the first column, minc = 1 and maxc = 4
64          minc = max(1, n - halfBlockSize);
65          maxc = min(imgWidth, n + halfBlockSize);
66
67          % Define the search boundaries as offsets from the template location.
68          % Limit the search so that we don't go outside of the image.
69          % 'mind' is the the maximum number of pixels we can search to the left.
70          % 'maxd' is the maximum number of pixels we can search to the right.
71          %
72          % In the "Cones" dataset, we only need to search to the right, so mind
73          % is 0.
74          %
75          % For other images which require searching in both directions, set mind
76          % as follows:
77          %   mind = max(-disparityRange, 1 - minc);
78          mind = 0;
79          maxd = min(disparityRange, imgWidth - maxc);
80
81          % Select the block from the right image to use as the template.
82          template = rightI(minr:maxr, minc:maxc);
83
84          % Get the number of blocks in this search.
85          numBlocks = maxd - mind + 1;
86
87          % Create a vector to hold the block differences.
88          blockDiffs = zeros(numBlocks, 1);
89
90          % Calculate the difference between the template and each of the blocks.
91          for (i = mind : maxd)
92
93              % Select the block from the left image at the distance 'i'.
94              block = leftI(minr:maxr, (minc + i):(maxc + i));
95
96              % Compute the 1-based index of this block into the 'blockDiffs' vector.
97              blockIndex = i - mind + 1;
98
99              % Take the sum of absolute differences (SAD) between the template
100             % and the block and store the resulting value.
101             blockDiffs(blockIndex, 1) = sum(sum(abs(template - block)));
102         end
103
104         % Sort the SAD values to find the closest match (smallest difference).
105         % Discard the sorted vector (the "~" notation), we just want the list
106         % of indices.
107         [temp, sortedIndeces] = sort(blockDiffs);
108
109         % Get the 1-based index of the closest-matching block.
110         bestMatchIndex = sortedIndeces(1, 1);
111
112         % Convert the 1-based index of this block back into an offset.
113         % This is the final disparity value produced by basic block matching.
114         d = bestMatchIndex + mind - 1;
115
116         % Store the calculated disparity value in the resultant img matrix
117         DbasicSubpixel(m, n) = d;
118     end
119
120     % Update progress bar every 5th row.
121     if (mod(m, 5) == 0)
122         str = sprintf('  Image Row %d / %d (%.0f%%)\n', m, imgHeight, (m / imgHeight) * 100)
                ;
123         waitbar(m/imgHeight,h,str)
124     end
125
126 end
127
128 % close the progress bar
129 close(h);
```

```
130
131 % Display the disparity map.
132 % Passing an empty matrix as the second argument tells imshow to take the
133 % minimum and maximum values of the data and map the data range to the
134 % display colors.
135 figure, imshow(DbasicSubpixel, []);
136 axis image;
137 colorbar;
138
139 % Specify the minimum and maximum values in the disparity map so that the
140 % values can be properly mapped into the full range of colors.
141 % If you have negative disparity values, this will clip them to 0.
142 caxis([0 disparityRange]);
143
144 % Set the title to display.
145 title(strcat('SAD Block Matching:  ',num2str(blockSize),'x',...
146     num2str(blockSize),' Block, ',num2str(disparityRange),'px Search Range'));
147
148 % plot both images in a final output graph
149 figure,
150 if (EXAMPLE_DATA == 0)
151     subplot(1,3,1), imshow(leftI)
152     title('Left Input Image')
153     subplot(1,3,3), imshow(rightI)
154 else
155     subplot(1,3,1), imshow(left)
156     title('Left Input Image')
157     subplot(1,3,3), imshow(right)
158 end
159 title('Right Input Image')
160 subplot(1,3,2), imshow(DbasicSubpixel,[])
161 title(strcat('SAD Block Matching: ',num2str(blockSize),'x',...
162     num2str(blockSize),' Block, ',num2str(disparityRange),'px Search Range'));
```

Verilog Algorithm:

```
1  `timescale 1ns / 1ps
2  //////////////////////////////////////////////////////////////////////////////////
3  // Disparity Algorithm implementation
4  //////////////////////////////////////////////////////////////////////////////////
5  module parallel_disparity(
6           input clk, // Read clk signal
7           input enable, // Enable new disparity calculation
8           input sw, // Input from left/right image buffer controller
9           input reset, // Reset disparity FSM
10          input [7:0] ldata, // Left bram pixel data in
11          input [7:0] rdata, // Right bram pixel data in
12          output reg [16:0] laddr, // Left bram read address
13          output reg [16:0] raddr, // Right bram read address
14          output reg [18:0] result_addr, // Result bram write address
15          output reg [7:0] result_data, // Result bram pixel data
16          output result_wea, // Result bram write enable
17          output [2:0] state_LED, // Current state indicator
18          output [7:0] lineout, // Single line pixel data out
19          input [6:0] lineaddr // Single line pixel data address
20      );
21
22 // user-defined constants (image search parameters)
23 parameter WIDTH = 384 - 1; // output image width (0-indexed)
24 parameter HEIGHT = 288 - 1; // output image height (0-indexed)
25 parameter SEARCH_RANGE = 20-1; // disparity block comparison search range (0-indexed)
26 parameter HALF_BLOCK = 3; // half block size
27 parameter FOCAL_LENGTH = 6; // 6mm
28 parameter BASELINE = 63; //63mm
29
30 // calculated constants
```

```
31 parameter BLOCK_SIZE = (2*HALF_BLOCK) + 1; // block size
32 parameter BLOCK_SIZE0 = (2*HALF_BLOCK); // block size with zero based index
33 parameter FB = FOCAL_LENGTH*BASELINE; // focal length * baseline (done here to save
       computation space)
34
35 // search variables (incremented automatically)
36 reg [8:0] col_count = 9'b0;  // number of cols iterated through (m in matlab code)
37 reg [8:0] row_count = 9'b0; // number of rows iterated through (n in matlab code)
38 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
39 reg [8:0] rcnt = 9'b0, dcnt=9'b0; // temporary counters based on above wires for search
       blocks
40 reg [2:0] cdcnt = 3'b0, rdcnt = 3'b0, ccnt = 3'b0;//temporary counters based on above wires
       for search blocks
41 reg [5:0] mind = 6'b0, maxd = 6'b0; // min/max disparity search bounds (limit SEARCH_RANGE
       to 63 blocks!)
42 reg [5:0] scnt = 6'b0; // number of disparity search comparisons performed
43 reg [5:0] numBlocks = 6'b0; // number of blocks within current search bounds
44 reg [5:0] blockIndex = 6'b0; // current block being searched in numBlocks
45 reg [5:0] index; // index of the min number in the disparity vector (disparity value)
46 reg [7:0] min; // value of min number in disparity vector
47 reg [1:0] pipe = 2'b00; // pipeline control for FSM
48 reg done; // will be 1 if disparity is 100% done, 0 otherwise (used for next_state == IDLE)
49
50 // temporary memory for template block, search block, and computational storage
51 // note that these blocks are broken down by row to allow for faster summations
52 // 7x7 template block
53 reg [7:0] template0 [0:BLOCK_SIZE0]; // template block row 0
54 reg [7:0] template1 [0:BLOCK_SIZE0]; // template block row 1
55 reg [7:0] template2 [0:BLOCK_SIZE0]; // template block row 2
56 reg [7:0] template3 [0:BLOCK_SIZE0]; // template block row 3
57 reg [7:0] template4 [0:BLOCK_SIZE0]; // template block row 4
58 reg [7:0] template5 [0:BLOCK_SIZE0]; // template block row 5
59 reg [7:0] template6 [0:BLOCK_SIZE0]; // template block row 6
60 // 7x7 search block
61 reg [7:0] block0 [0:BLOCK_SIZE0]; // search block row 0
62 reg [7:0] block1 [0:BLOCK_SIZE0]; // search block row 1
63 reg [7:0] block2 [0:BLOCK_SIZE0]; // search block row 2
64 reg [7:0] block3 [0:BLOCK_SIZE0]; // search block row 3
65 reg [7:0] block4 [0:BLOCK_SIZE0]; // search block row 4
66 reg [7:0] block5 [0:BLOCK_SIZE0]; // search block row 5
67 reg [7:0] block6 [0:BLOCK_SIZE0]; // search block row 6
68 // 7x7 storage for abs(template-block)
69 reg [7:0] SAD_diffs0 [0:BLOCK_SIZE0]; // block for holding abs(template-block) row 0
70 reg [7:0] SAD_diffs1 [0:BLOCK_SIZE0]; // block for holding abs(template-block) row 1
71 reg [7:0] SAD_diffs2 [0:BLOCK_SIZE0]; // block for holding abs(template-block) row 2
72 reg [7:0] SAD_diffs3 [0:BLOCK_SIZE0]; // block for holding abs(template-block) row 3
73 reg [7:0] SAD_diffs4 [0:BLOCK_SIZE0]; // block for holding abs(template-block) row 4
74 reg [7:0] SAD_diffs5 [0:BLOCK_SIZE0]; // block for holding abs(template-block) row 5
75 reg [7:0] SAD_diffs6 [0:BLOCK_SIZE0]; // block for holding abs(template-block) row 6
76 // 7x1 storage for sum(abs(template-block))
77 reg [10:0] temp0; // block for holding sum(abs(template-block)) row 0
78 reg [10:0] temp1; // block for holding sum(abs(template-block)) row 1
79 reg [10:0] temp2; // block for holding sum(abs(template-block)) row 2
80 reg [10:0] temp3; // block for holding sum(abs(template-block)) row 3
81 reg [10:0] temp4; // block for holding sum(abs(template-block)) row 4
82 reg [10:0] temp5; // block for holding sum(abs(template-block)) row 5
83 reg [10:0] temp6; // block for holding sum(abs(template-block)) row 6
84 // 20x1 for storing SAD value for ALL search blocks corresponding to a single template block
85 reg [14:0] SAD_vector [0:SEARCH_RANGE]; // block for holding sum(sum(abs(template-block))) -
       up to 9x9 block size
86 reg [10:0] line; // reg for holding [(focal length)*(baseline)]/index
87
88 // ~~~~~~~~~~~~~~~ Disparity FSM ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
89 parameter [2:0] IDLE = 3'b000, // wait for next read sequence
90                 READ = 3'b001, // read data from FIFO
91                                SEPARATE = 3'b010, // separate search image into rows
92                                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)
```

```
159                 // search block: (b_minc:b_maxc,minr:maxr)
160                 // read in template image block
161                 if(ccnt <= (t_maxc-t_minc) && rcnt <= (maxr-minr)) // fully within template
                        search bounds
162                     case(rcnt)
163                  0:  template0[ccnt] <= ldata;
164                  1:  template1[ccnt] <= ldata;
165                  2:  template2[ccnt] <= ldata;
166                  3:  template3[ccnt] <= ldata;
167                  4:  template4[ccnt] <= ldata;
168                  5:  template5[ccnt] <= ldata;
169                  6:  template6[ccnt] <= ldata;
170                      endcase
171                 else // outside tempate bounds
172                     case(rcnt)
173                  0:  template0[ccnt] <= 8'h00;
174                  1:  template1[ccnt] <= 8'h00;
175                  2:  template2[ccnt] <= 8'h00;
176                  3:  template3[ccnt] <= 8'h00;
177                  4:  template4[ccnt] <= 8'h00;
178                  5:  template5[ccnt] <= 8'h00;
179                  6:  template6[ccnt] <= 8'h00;
180              endcase
181
182                 // read in search image block
183                 if(ccnt <= (b_maxc-b_minc) && rcnt <= (maxr-minr)) // fully within template
                        search bounds
184             case(rcnt)
185                 0: block0[ccnt] <= rdata;
186                 1: block1[ccnt] <= rdata;
187                 2: block2[ccnt] <= rdata;
188                 3: block3[ccnt] <= rdata;
189                 4: block4[ccnt] <= rdata;
190                 5: block5[ccnt] <= rdata;
191                 6: block6[ccnt] <= rdata;
192                     endcase
193                 else // outside tempate bounds
194                     case(rcnt)
195                 0: block0[ccnt] <= 8'h00;
196                 1: block1[ccnt] <= 8'h00;
197                 2: block2[ccnt] <= 8'h00;
198                 3: block3[ccnt] <= 8'h00;
199                 4: block4[ccnt] <= 8'h00;
200                 5: block5[ccnt] <= 8'h00;
201                 6: block6[ccnt] <= 8'h00;
202             endcase
203
204                 // increment ccnt and rcnt to iterate through all pixels within blocks
205                 if(pipe == 2'b11) begin
206                     pipe <= 2'b00;
207             if(ccnt<(BLOCK_SIZE0))
208                 ccnt<=ccnt+1'b1;
209             else if(rcnt<(BLOCK_SIZE0) && ccnt==(BLOCK_SIZE0)) begin
210                 rcnt <= rcnt+1'b1;
211                 ccnt <= 3'b0;
212             end
213                 end
214                 else
215                   pipe <= pipe + 1'b1;
216
217                 // make sure pipe is clear for SAD
218                 if(next_state == SAD)begin
219                         pipe <= 2'b00;
220                         ccnt <= 3'b0;
221                         rcnt <= 9'b0;
222                         cdcnt <= 3'b0;
223                         rdcnt <= 3'b0;
224                 end
```

```
225         end
226
227         SAD:
228         begin
229         // ~~~~~~~~~~~~~~~~~ abs(template-block) ~~~~~~~~~~~~~~~~
230                 if (pipe == 2'b00) begin
231                     // abs0
232                         if(template0[ccnt]>block0[ccnt])
233                                 SAD_diffs0[ccnt] <= template0[ccnt] - block0[ccnt];
234                         else
235                                 SAD_diffs0[ccnt] <= block0[ccnt] - template0[ccnt];
236                     // abs1
237                     if(template1[ccnt]>block1[ccnt])
238                 SAD_diffs1[ccnt] <= template1[ccnt] - block1[ccnt];
239             else
240                 SAD_diffs1[ccnt] <= block1[ccnt] - template1[ccnt];
241             // abs2
242             if(template2[ccnt]>block2[ccnt])
243                 SAD_diffs2[ccnt] <= template2[ccnt] - block2[ccnt];
244             else
245                 SAD_diffs2[ccnt] <= block2[ccnt] - template2[ccnt];
246             // abs3
247             if(template3[ccnt]>block3[ccnt])
248                 SAD_diffs3[ccnt] <= template3[ccnt] - block3[ccnt];
249             else
250                 SAD_diffs3[ccnt] <= block3[ccnt] - template3[ccnt];
251             // abs4
252             if(template4[ccnt]>block4[ccnt])
253                 SAD_diffs4[ccnt] <= template4[ccnt] - block4[ccnt];
254             else
255                 SAD_diffs4[ccnt] <= block4[ccnt] - template4[ccnt];
256             // abs5
257             if(template5[ccnt]>block5[ccnt])
258                 SAD_diffs5[ccnt] <= template5[ccnt] - block5[ccnt];
259             else
260                 SAD_diffs5[ccnt] <= block5[ccnt] - template5[ccnt];
261             // abs6
262             if(template6[ccnt]>block6[ccnt])
263                 SAD_diffs6[ccnt] <= template6[ccnt] - block6[ccnt];
264             else
265                 SAD_diffs6[ccnt] <= block6[ccnt] - template6[ccnt];
266
267             // increment through each index in all template and search rows
268                         if(ccnt<(BLOCK_SIZE0))
269                                 ccnt<=ccnt+1'b1;
270                         else
271                                 pipe <= 2'b01; // proceed to next stage of SAD
272                 end
273
274                 // ~~~~~~~~~~~~~~~~ sum(abs(template-block)) ~~~~~~~~~~~~~~~~
275                 if(pipe == 2'b01) begin
276                         if(cdcnt < BLOCK_SIZE) begin // 0 .. block_size-1
277                 case(rdcnt) // sum(abs0 + abs1 + abs2 + abs3 + abs4 + abs5 + abs6) for all
                        columns within each row
278                     0: temp0 <= SAD_diffs0[cdcnt] + SAD_diffs1[cdcnt] + SAD_diffs2[cdcnt] +
                            SAD_diffs3[cdcnt] + SAD_diffs4[cdcnt] + SAD_diffs5[cdcnt] +
                            SAD_diffs6[cdcnt];
279                     1: temp1 <= SAD_diffs0[cdcnt] + SAD_diffs1[cdcnt] + SAD_diffs2[cdcnt] +
                            SAD_diffs3[cdcnt] + SAD_diffs4[cdcnt] + SAD_diffs5[cdcnt] +
                            SAD_diffs6[cdcnt];
280                     2: temp2 <= SAD_diffs0[cdcnt] + SAD_diffs1[cdcnt] + SAD_diffs2[cdcnt] +
                            SAD_diffs3[cdcnt] + SAD_diffs4[cdcnt] + SAD_diffs5[cdcnt] +
                            SAD_diffs6[cdcnt];
281                     3: temp3 <= SAD_diffs0[cdcnt] + SAD_diffs1[cdcnt] + SAD_diffs2[cdcnt] +
                            SAD_diffs3[cdcnt] + SAD_diffs4[cdcnt] + SAD_diffs5[cdcnt] +
                            SAD_diffs6[cdcnt];
282                     4: temp4 <= SAD_diffs0[cdcnt] + SAD_diffs1[cdcnt] + SAD_diffs2[cdcnt] +
                            SAD_diffs3[cdcnt] + SAD_diffs4[cdcnt] + SAD_diffs5[cdcnt] +
```

```
                    SAD_diffs6[cdcnt];
283                 5: temp5 <= SAD_diffs0[cdcnt] + SAD_diffs1[cdcnt] + SAD_diffs2[cdcnt] +
                        SAD_diffs3[cdcnt] + SAD_diffs4[cdcnt] + SAD_diffs5[cdcnt] +
                        SAD_diffs6[cdcnt];
284                 6: temp6 <= SAD_diffs0[cdcnt] + SAD_diffs1[cdcnt] + SAD_diffs2[cdcnt] +
                        SAD_diffs3[cdcnt] + SAD_diffs4[cdcnt] + SAD_diffs5[cdcnt] +
                        SAD_diffs6[cdcnt];
285                     endcase
286                 end else begin // avg accross block width when one sum is done temp[
                        idx]=[sum[idx](0..7)/7]
287                     case(rdcnt)
288                 0: temp0 <= (temp0/BLOCK_SIZE);
289                 1: temp1 <= (temp1/BLOCK_SIZE);
290                 2: temp2 <= (temp2/BLOCK_SIZE);
291                 3: temp3 <= (temp3/BLOCK_SIZE);
292                 4: temp4 <= (temp4/BLOCK_SIZE);
293                 5: temp5 <= (temp5/BLOCK_SIZE);
294                 6: temp6 <= (temp6/BLOCK_SIZE);
295                         endcase
296                 end
297
298                     // iterate through each colum within each row
299                     if(cdcnt<BLOCK_SIZE)
300                         cdcnt<=cdcnt+1'b1;
301                 // after finishing all sums, reduce to an average value
302                 else if(cdcnt == BLOCK_SIZE && rdcnt < BLOCK_SIZE0) begin
303                     rdcnt <= rdcnt + 1'b1;
304                     cdcnt <= 0;
305                 // when finished, reset for next stage of SAD
306                     end else begin
307                         pipe <= 2'b10;
308                         ccnt <= 3'b0;
309                         rcnt <= 9'b0;
310                         cdcnt <= 3'b0;
311                         rdcnt <= 3'b0;
312                     end
313             end
314
315             // ~~~~~~~~~~~~~~~~ sum(sum(abs(template-block))) ~~~~~~~~~~~~~~~~
316             if (pipe == 2'b10) begin // pipe = 2'b10
317                     if(ccnt<3'b001) begin
318                         SAD_vector[blockIndex] <= temp0+temp1+temp2+temp3+temp4+
                            temp5+temp6;
319                         ccnt <= ccnt + 1'b1;
320                     end
321                     else begin
322                         SAD_vector[blockIndex] <= SAD_vector[blockIndex]/(BLOCK_SIZE
                            );
323                         ccnt <= 3'b0;
324                         pipe <= 2'b11;
325                     end
326             end
327
328         // update SAD vector index (when full, proceed to finalization)
329         // this index represents the position of the current search block
330         // in relation to the template block (0-19 for a 20px disparity search)
331             if(dcnt < maxd && pipe == 2'b11) begin
332                     dcnt <= dcnt + 1'b1; // number of searches performed
333                     blockIndex <= dcnt - mind; // index in SAD_vector
334             end
335
336             // update cols & rows processed by the algorithm after comparing
337             // SEARCH_RANGE search blocks to the current template
338             if (next_state == FINALIZE) begin
339                     scnt <= 6'b0;
340                     pipe <= 2'b00;
341
342                     // increment accross each row of pixels
```

```
343                     if(col_count < (WIDTH-(HALF_BLOCK+1'b1)))
344                 col_count <= col_count + 1'b1;
345             // increment through all rows if NOT performing a line search
346                     else if (~sw && col_count == (WIDTH-(HALF_BLOCK+1'b1)) && row_count
                            < HEIGHT) begin
347                 row_count <= row_count + 1'b1;
348                 col_count <= 9'b0;
349                     end
350                     // increment through two rows if performing a line search
351                 else if (sw && col_count == (WIDTH-(HALF_BLOCK+1'b1)) && row_count < 9'd145)
                         begin
352                     if(row_count < 9'd144)
353                        row_count <= 9'd144;
354                     else
355                     row_count <= row_count + 1'b1;
356                 col_count <= 9'b0;
357                     end
358
359                     // full disparity search complete when all rows and cols have been
                            processed
360                     if(~sw && col_count == (WIDTH-(HALF_BLOCK+1'b1)) && row_count ==
                            HEIGHT)
361                        done <= 1'b1;
362                     // line disparity search complete when two most central rows and
                            cols have been processed
363                     else if (sw && col_count == (WIDTH-(HALF_BLOCK+1'b1)) && row_count
                            >= 9'd145)
364                        done <= 1'b1;
365                     // if neither search is complete, allow for the FSM to keep cycling
366                     else
367                        done <= 1'b0;
368             end
369
370             // reset for a new SAD sequence if there are more search blocks to compare
                    to the template
371             if (next_state == SEPARATE) begin
372                     ccnt <= 3'b0;
373                     rcnt <= 9'b0;
374                     cdcnt <= 3'b0;
375                     rdcnt <= 3'b0;
376                     pipe <= 2'b00;
377             end
378         end
379
380         FINALIZE:
381         begin
382                 dcnt <= 9'b0;
383                 // search for index of min value in SAD_vector
384                 // this index corresponds to the pixel offset between the template
385                 // block and the closest matching search block
386                 if(scnt<numBlocks) begin
387                     // value at idx=0 will always be the lowest value to start...
388                         if(scnt == 6'b0) begin
389                                 min <= SAD_vector[0];
390                                 index <= 8'h00;
391                                 end
392                     // update this value and its index while incrementing through...
393                         else if(SAD_vector[scnt]<min) begin
394                                 min <= SAD_vector[scnt];
395                                 index <= scnt;
396                                 end
397                         scnt <= scnt + 1'b1;
398                 end
399
400                 // place disparity value in output image array after finding the index
401                 else begin
402             if(sw == 1'b0) // use disparity values when in full search mode
403                 result_data <= index;
```

```
404             else // use depth values when in line mode
405                 // in this case I sum 8 pixel values accross two lines for every output
                        pixel
406                 // non-zero pixel value on any col other than 1st
407                 if(index > 0 && pipe > 2'b00)
408                   line <= line + (FB/index);
409                 // non-zero pixel value on first column of pixels
410                 else if (index > 0)
411                   line <= (FB/index);
412                 // zero pixel value on first row of pixels
413                 else if (row_count == 9'd144 && pipe <= 2'b00)
414                   line <= 8'h00;
415                 // zero pixel value in any other location
416                 else
417                   line = line;
418
419             // count through 4 pixels for each iteration of line mode
420             pipe <= pipe + 1'b1; //pipe <= 2'b11; // was 2'b11, changed to add a little
                    extra time
421                 end
422         end
423 endcase
424
425 // single line pixel buffer
426 line_bram linebuf (
427   .clka(clk),    // input wire clka
428   .wea(pipe == 2'b11 && sw && row_count == 9'd144),// input wire [0 : 0] wea
429   .addra(col_count[8:2]),  // addra will increment every 4 pixels
430   .dina(line>>3),    // dina is the sum of 8 depth values / 8
431   .clkb(clk),    // input wire clkb
432   .addrb(lineaddr),  // line address from top-level display logic
433   .doutb(lineout)  // depth value to top-level display logic
434 );
435
436 // disparity value output buffer write enable
437 assign result_wea = (current_state == FINALIZE && scnt == (numBlocks)) ? 1'b1 : 1'b0;
438
439 // result address
440 always @(posedge clk)
441     result_addr = ((WIDTH+1'b1)*row_count)+col_count;
442
443 // left image buffer read address (for template block)
444 always @(posedge clk)
445     laddr = ((WIDTH+1'b1)*(minr+rcnt))+(t_minc+ccnt);
446
447 // right image buffer read address (for search block)
448 always @(posedge clk)
449     raddr = ((WIDTH+1'b1)*(minr+rcnt))+(b_minc+ccnt);
450
451 // assign disparity block search bounds
452 always @(row_count,col_count,t_maxc,maxd,mind,dcnt)
453 begin
454                 minr = (0 > $signed(row_count - HALF_BLOCK)) ? 9'b0 : (row_count -
                        HALF_BLOCK);
455                 maxr = ((HEIGHT) < (row_count + HALF_BLOCK)) ? HEIGHT : (row_count +
                        HALF_BLOCK);
456                 t_minc = (0 > $signed(col_count - HALF_BLOCK)) ? 9'b0 : (col_count -
                        HALF_BLOCK);
457                 t_maxc = ((WIDTH) < (col_count + HALF_BLOCK)) ? WIDTH : (col_count +
                        HALF_BLOCK);
458                 b_minc =(0 > $signed(dcnt - HALF_BLOCK+col_count)) ? 9'b0 : (dcnt -
                        HALF_BLOCK + col_count);
459                 b_maxc = ((WIDTH) < (dcnt + HALF_BLOCK)) ? WIDTH : (dcnt + col_count +
                        HALF_BLOCK);
460 end
461
462 // assign disparity search bounds
463 always @(t_maxc,maxd,mind,current_state)
```

```
464         begin
465                 mind = 6'b0; // or = max(-SEARCH_RANGE, 1-t_minc)
466                 if (current_state == READ)
467                         maxd = SEARCH_RANGE;
468                 else if(current_state == FINALIZE)
469                         maxd = (SEARCH_RANGE < ((WIDTH) - t_maxc)) ? SEARCH_RANGE : ((WIDTH)
                                - t_maxc);
470                 numBlocks = maxd - mind;
471 end
472
473 // current state indicator LED
474 assign state_LED = current_state;
475
476 endmodule
```