Simultaneous Personnel Localization and Mapping

A Major Qualifying Project Report Submitted to the Faculty of
Worcester Polytechnic Institute

In partial fulfillment of the requirements for the
Degree of Bachelor of Science in Electrical & Computer Engineering

By:

Georges Gauthier
John DeCusati

August 29, 2016

Advisor:
Professor R. James Duckworth

Contents

List of Figures

Abstract

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

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

1 Introduction

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

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

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

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

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

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

2 Background

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

SLAM is a widely expanding field with much potential for improvement. One application of such a system is a proof of concept of camera-based SLAM systems, presented by Andrew Davison of Oxford University in a research paper entitled "Real-Time SLAM with a Single Camera" [2]. This system is handheld and relies on a computer using a 2.2 GHz Pentium processor connected to a single camera and laser rangefinder. The system requires prior knowledge of the area being analyzed before it can successfully localize and map. It implements edge detection, but is limited to the narrow field of vision of the rangefinder, so it is only able to map an object directly in front of it. This system carries a latency of around 33 milliseconds. An output frame of the device is shown in Figure 1.

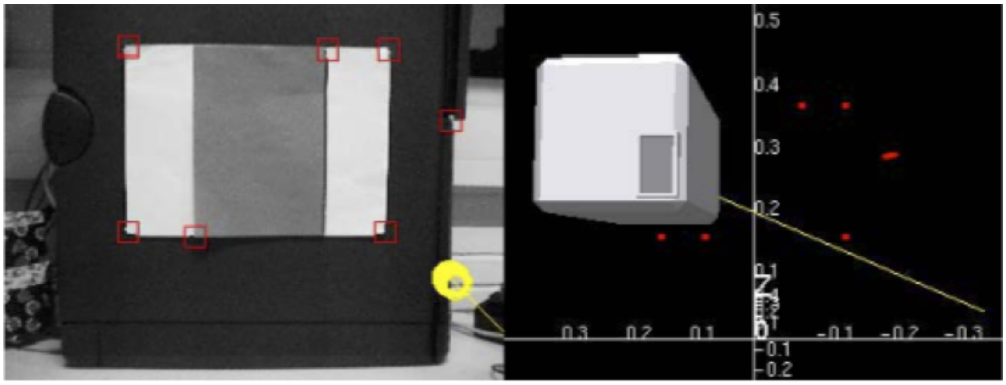


Figure 1: Real-Time SLAM with a Single Camera [2]

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

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

A more commercial device similar to our concept is Serveball?s Squito^{TM} [5]. Squito is a wireless, throwable, 360° panoramic camera that implements target detection to stabilize the video feed from its many cameras. It is shown in Figure 2 below.

Figure 2: Serveball's Squito [5]

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

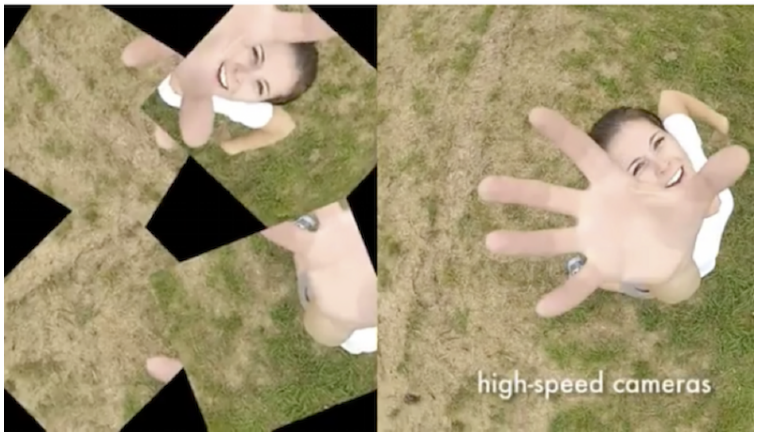


Figure 3: Serveball's Squito Input and Output [5]

By using multiple camera sensors in a sensor suite, it is also possible to determine depth information from corresponding images of an area. This technique is known as stereo imaging, and the process of gathering depth information from a pair of stereo images is known as disparity mapping. University of Bologna researchers Stefano Mattoccia and Matteo Poggi have worked to implement a real-time disparity mapping algorithm on an FPGA, and an example of a stereo image disparity is shown in Figure 4 below [6]. Using their stereo vision algorithm, the researchers are able to generate real-time image data showing the relative locations of objects within an image frame using color gradients. Based on this depth information, it is also possible to detect objects located within the field of view of the stereo imaging system, as shown in Figure 4. An implementation similar to this would be extremely useful in a SLAM-like system, as it would allow for the localization of objects and creation of 2D "floorplans" of an area in real-time using only two camera sensors.

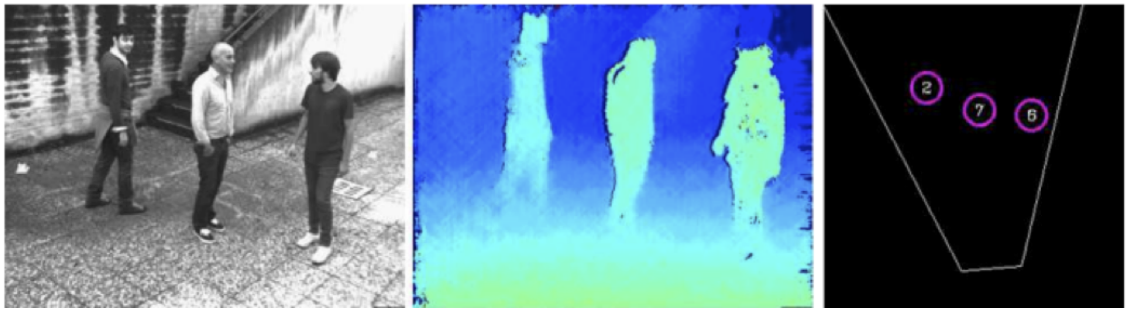


Figure 4: From Left to Right: Original Image, Disparity Map, Object Detection Results [6]

Many security systems implement human detection and human body tracking in order to increase their effectiveness. These devices process real-time images in order to identify human characteristics, and are limited to the field of vision of a stationary or rotating camera. An example of this type of system is explored by the Mitsubishi corporation in a research paper entitled "Human Body Tracking by Adaptive Background Models and Mean-Shift Analysis" [9]. The paper explores a stationary image processing system implemented on a PC platform with a 1.8GHz processor that yields a maximum processing time of 100 milliseconds. An output frame of the system is shown in Figure 5 below.

Figure 5: Human Body Tracking by Adaptive Background Models and Mean-Shift Analysis [9]

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

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

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

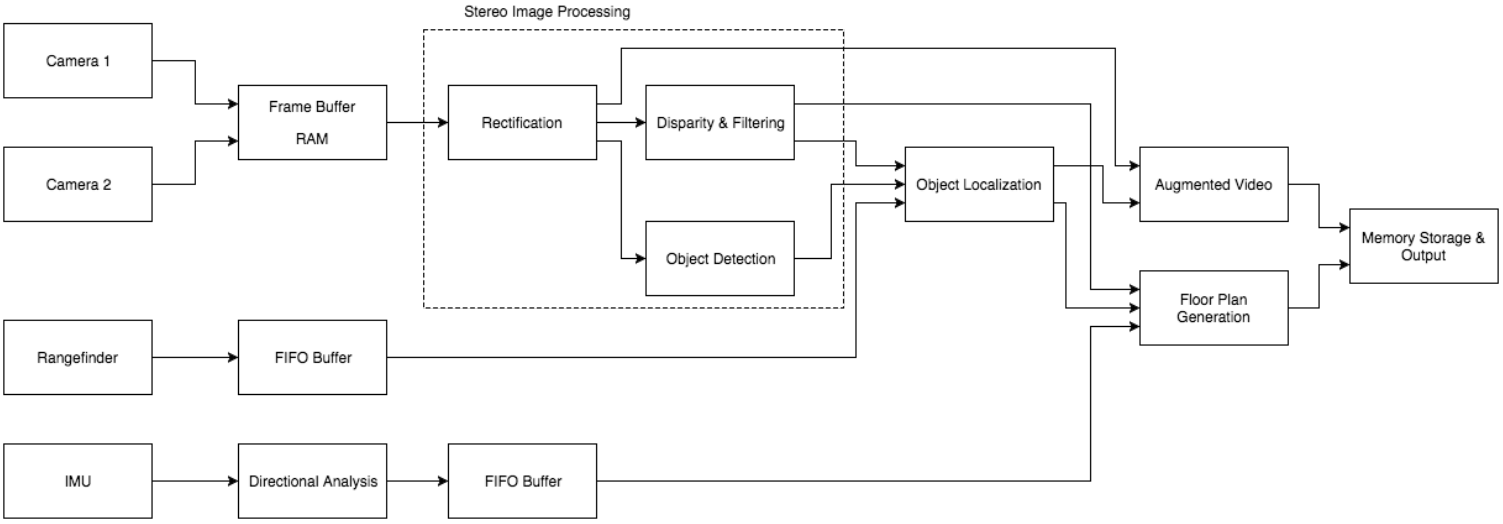


Figure 6: Functional Block Diagram

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

As our research has progressed over time, our project objectives have continually evolved. We originally envisioned the creation of a device that used laser rangefinders to create 3D maps of its surroundings, similar to that of a Carnegie Mellon University device created in order to volumetrically map abandoned mines [10].

As our research progressed, we believed that we could use a visual light and thermal imaging camera set to gather information on an area, and supplement that data with IMU and rangefinder readings in order to produce detailed maps of our sensor suites? surroundings. Eventually we came upon the concept of disparity mapping and generating depth information from image data, and decided that we would again like to shift the overall setup of our device to rely mainly on stereo image data. Due to our overall budget and the resources that have been made available to us, in the coming terms we plan to use an electronic rangefinder, IMU, and stereo camera pair to generate real-time SPLAM video and floorplan information. Although we were also originally planning on including a thermal camera in our sensor suite as well, we have decided to eliminate the module in favor of higher quality cameras due to its prohibitive cost, low resolution, slow sampling rate, and small field of view. More information on this decision can be found in Appendix item 4.3.

3 Methodology

3.1 Camera Testing

After obtaining two of the MT9V034 cameras chosen through the process referenced in Appendix item 4.3, several steps were taken to obtain test images from each camera. These steps are outlined in the following sections.

3.1.1 Camera Operation

In order to gather working images from each camera module, we first needed to understand what circuitry our camera module breakouts contained so that we could interface with them. The MT9V034 camera breakouts used have been purchased through Leopard Imaging Inc. Although these camera module breakout boards are intended to be used with Leopard Imaging's LeopardBoard ARM development board, the breakouts were found to contain only the supporting circuitry recommended in the MT9V034 datasheet, and we decided that they would be ideal for our application [4, 8].

Once the schematics of each camera module breakout were known, it was then possible to design a basic control interface for each camera. According to the MT9V034 datasheet, each camera module needs to be supplied with an external Master Clock and Output Enable signal in order to operate [8]. A simple Verilog module for the Nexys3 Spartan-6 FPGA board was created in order to supply the camera module with a 24MHz master clock signal, and a switch was used to toggle output enable. With this module implemented, the camera module's default outputs could then be observed. In order to interface the camera module with an FPGA, the breakout board shown in Figure 7 was also created to make the module's pins more easily accessible.

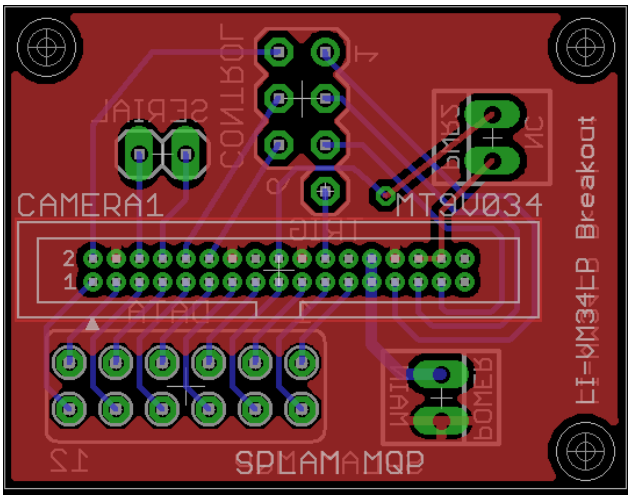


Figure 7: LI-VM34LP Breakout Board

By default, the MT9V034 camera module will continuously gather image data at 60Hz as long as it is supplied with an external clock signal and output is enabled [8]. Several output signals from the camera module are then used to transmit image data. Each image, or frame, is broken up into individual "lines" which correspond to a line of pixels that stretch the width of the frame. Since our camera module captures images at 752x480 pixel resolution, one frame will contain 480 lines of 752 pixels each. The camera module breaks up image data by frame and line, and camera data pins FRAME_VALID and LINE_VALID are toggled to indicate the transmission of a frame or line. The timing diagram shown in Figure 8 shows the operation of these pins while transmitting an image.

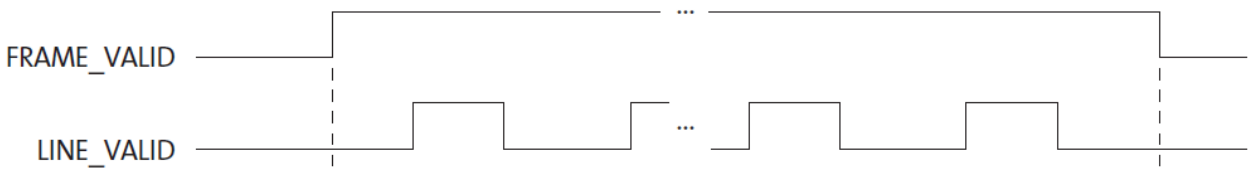


Figure 8: Frame and Line Valid [8]

Since the MT9V034 module transmits image data in parallel and each pixel contains 10 bits of resolution, 10 pins are used to transmit pixel values. Pixel data is transmitted in correspondence with LINE_VALID and output clock signal PIXCLK. When LINE_VALID is asserted, the pixel data pins are updated with values corresponding to pixels 0-751 of

the given line. Values for each pixel are written out on the falling edge of the camera's PIXCLK pin, allowing for each pixel's value to be read on each rising PIXCLK edge. A full LINE_VALID data transmission sequence will therefore contain 752 PIXCLK cycles, corresponding to the 752 pixels that make up the given line. A timing diagram of this data transmission scheme is shown in Figure 9.

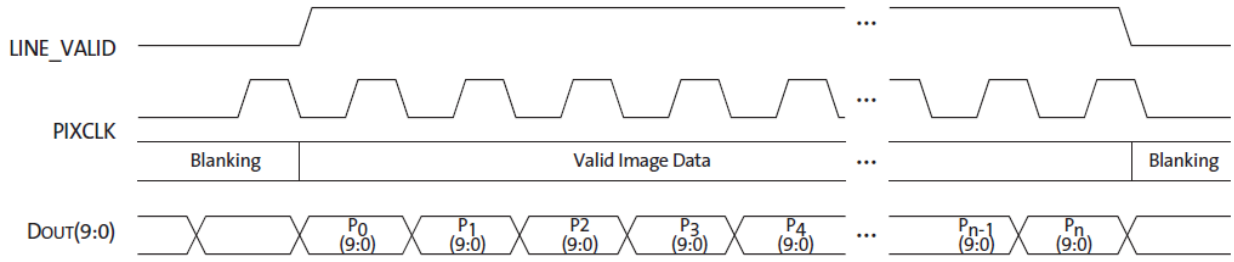


Figure 9: Line Data Transfer [8]

The default camera data transmission scheme was also examined using an oscilloscope, as shown in Figure 10, with channels 1-4 corresponding to camera PCLK, FRAME_VALID, LINE_VALID, and Data[0], respectively. In the case of Figure 10, the camera is initially powered off, resulting in an inactive PCLK signal during the beginning of the recording.

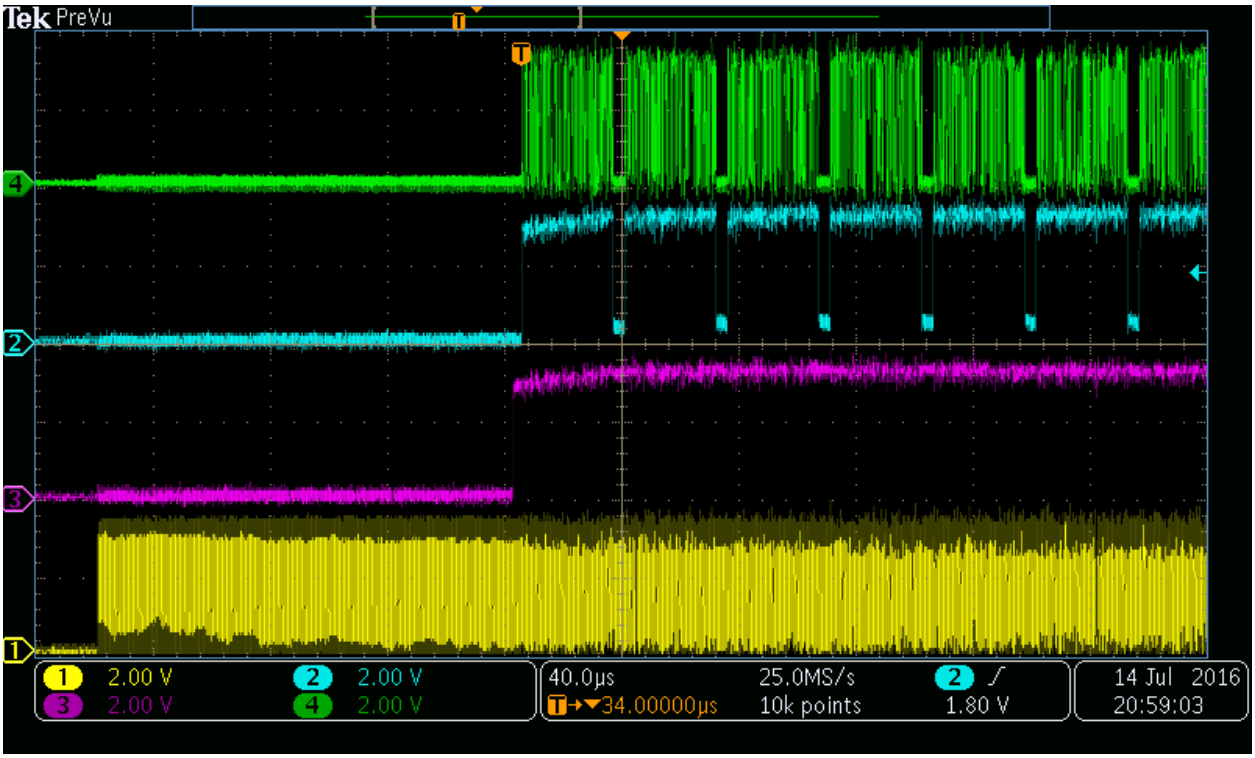


Figure 10: Camera Data Transfer

3.1.2 I^{2}C Control

The MT9V034 Camera module's mode of operation can be configured using a standard I^{2}C control interface. I^{2}C, or Inter-Integrated Circuit, is a bidirectional serial interface that allows for a master device to read from and write to several slave devices sharing the same data bus. An I^{2}C interface will use a Serial Data Line (SDA) and Serial Clock Line (SCL) that are normally pulled to 5V. When one connected I^{2}C device wishes to communicate with another, it will pull the SDA line low while leaving the SCL line high. The master device will then begin clocking the SCL line, and SDA will be used to transfer 7 bits representing the address of the desired slave device, along with an 8th bit representing whether it would like to read from or write to the device. An example of this transfer is shown in Figure 11. A second 8 bit sequence representing a specific register within the slave device may also be transmitted following the device address. For example, if the master device wishes to write to slave device 0x40 at register 0x00, it will transmit 0x41 (address 0x40 and WRITE),

followed by 0x00. If the slave device receives this transmission, it will acknowledge by pulling the SDA line low. At this point, the master can then transmit the value that it wishes to write to the given slave address and register. If the operation were a read rather than a write, the slave would transmit a value back to the master.

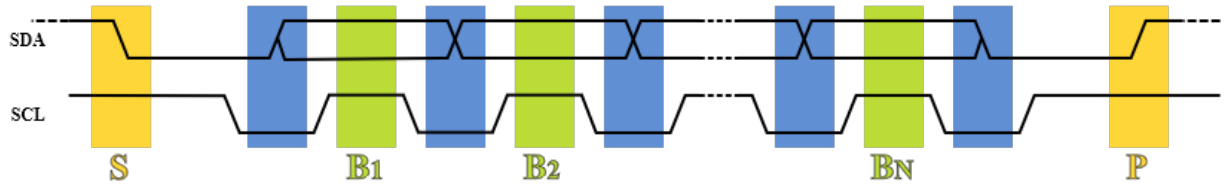


Figure 11: Example I^2C Data Transfer

Based on the LIVM34LP camera board schematic, each breakout board have been configured so that its camera is accessible at I^2C address 0x58 [4, 8]. Note that since both cameras come configured with the same I^2C bus address, a pullup resistor must be added to one of the cameras I^2C address lines so that both are individually accessible on a shared bus.

Although the MT9V034 camera control registers are closed source, the previous model's registers are available in the camera module datasheet, and have been found to work with the current model thus far [7]. As a baseline, the camera module was sent a read request at address 0x00, which should return 0x1324 for the MT9V034 camera module. An oscilloscope screenshot of this request is shown in Figure 12, with the first packet consisting of a request to address 0x00 of device 0x058, and the second packet consisting of the camera's response of 0x1324.

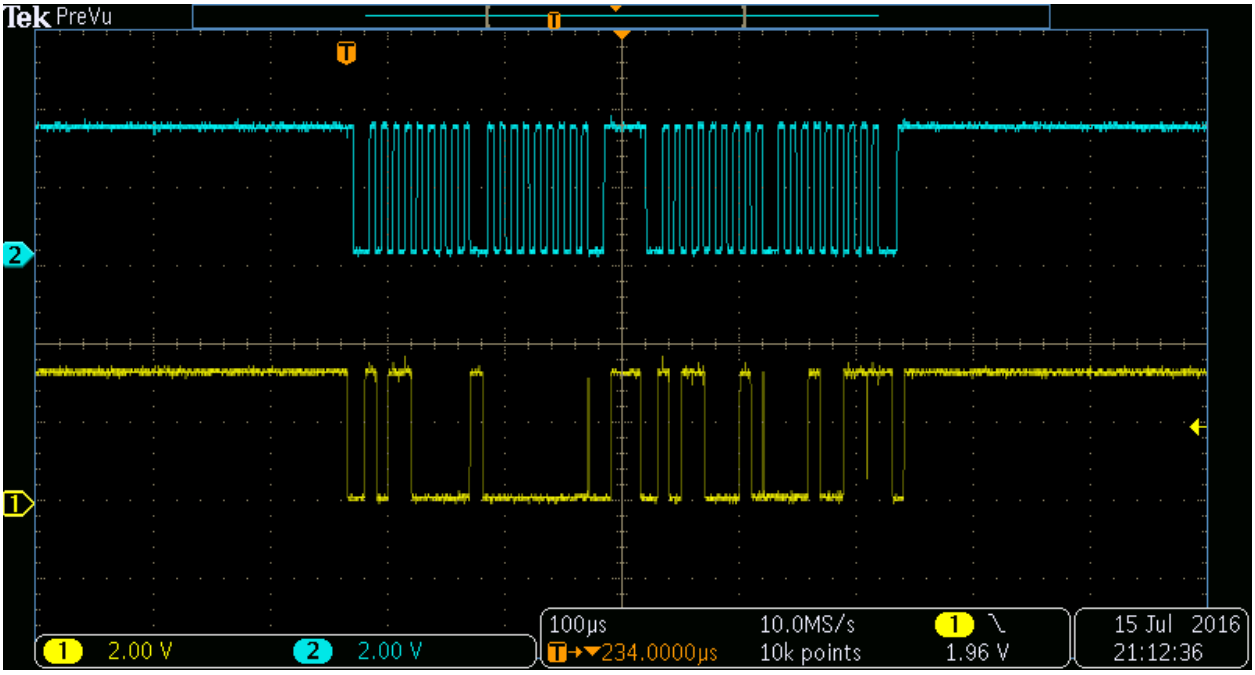


Figure 12: Example I^2C Transfer with Camera

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

According to the previous camera iteration's datasheet, the camera module's operational mode can be set through control register 0x07. By default, this register will be set to a value of 0x0388, which corresponds to master mode with parallel output and simultaneous readout of pixel data enabled [7]. In order to put the camera in trigger mode, the control register needs to be written with value 0x0198, which allows for the same functionality as before with the exception of having continuous shutter mode replaced with an external trigger. For reference, a table with bit descriptions for the camera control register can be found in Appendix item 4.4 [7].

After modifying the state of this register, a button was 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 13. This oscilloscope screenshot can be seen as an example of how the camera is no longer in continuous operation, since FRAME_VALID only asserts itself in response to a TRIGGER input.

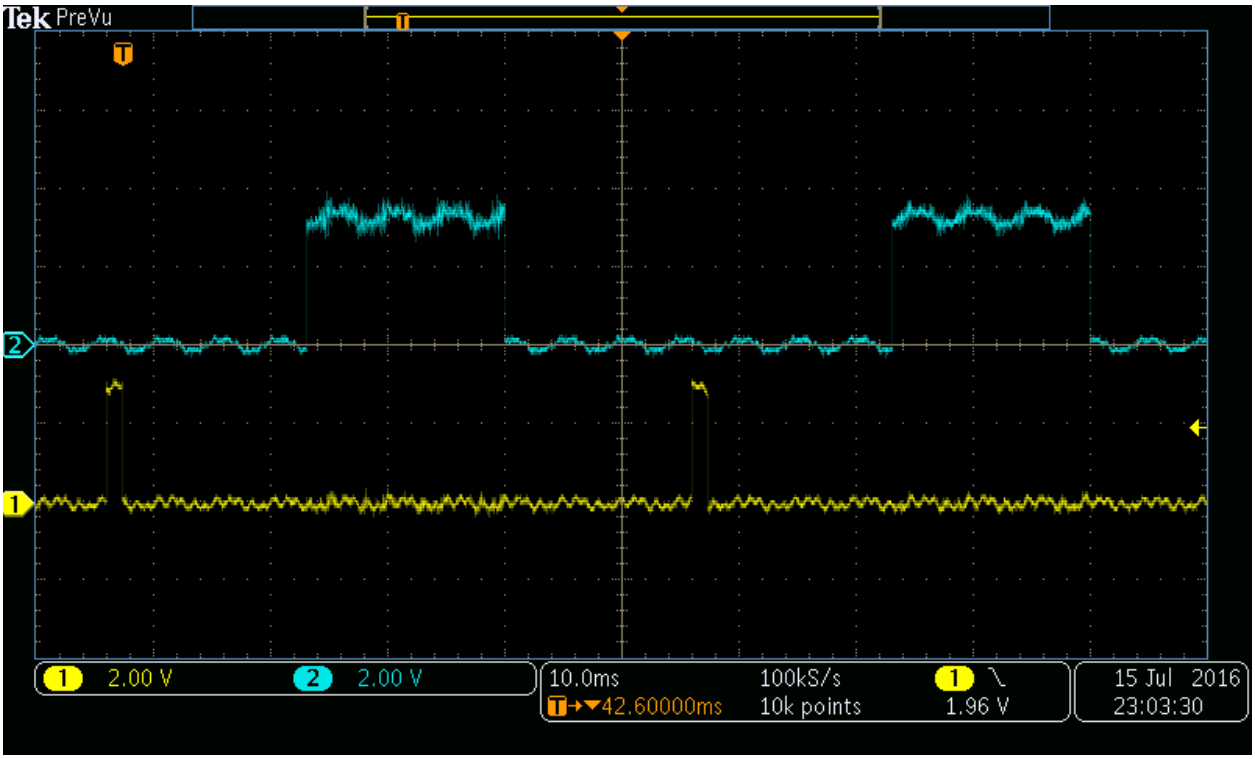


Figure 13: Camera Trigger and FV in Trigger Mode

3.1.3 Data Management

After successfully creating a camera control interface and placing the MT9V034 camera module in trigger mode, it was then possible to begin viewing images from the module. Since each image contains 752x480 pixels with 10 bits of resolution per pixel, a full camera image will consume 3,609,600 bits, or 440.6kB, as shown in Equation 1.

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

In order to send a camera image to a computer or monitor for viewing, several steps need to be taken. Although it would be ideal to transfer the image directly from the camera to a

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

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

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

since the truncation will correspond to a 4/1024 reduction in the range of values that each pixel can map to.

With the inclusion of the external FIFO module, it is now possible to capture and store an image for future reading, and to read out image data in chunks. Keeping this in mind, the system shown in Figure 14 was created for capturing, storing, and transmitting camera images to a computer for external analysis. In order to reduce development time, an external microcontroller was used for controlling the camera module's I^2C interface and placing the module in trigger mode. Various buttons and switches on the FPGA were then used for controlling the camera output and trigger, allowing for a user to trigger an image for storage on the AL422B FIFO. Once the image has been stored on the FIFO, the FPGA is capable of reading the image line-by-line into an internal buffer. An internal System on Chip (SoC) is used to control FPGA reads from the FIFO into this internal buffer. An image dump will begin when the SoC microcontroller signals to the FPGA to read a new line of pixels into its internal 8 bit by 752 address pixel buffer. The FPGA will then signal to the microcontroller when this buffer has been filled, and the microcontroller will print out the value of each pixel in the buffer to a connected computer over a Universal Asynchronous Reciever/Transmitter (UART) port. When the microcontroller finishes printing out the value of each pixel in the line buffer, it will signal to the FPGA to read in a new line of pixels. This process will repeat for each of the 480 lines of pixels in the image, allowing for the transmission of an entire image's worth of data from FIFO to computer. The Verilog implementation of the top module and line buffer for this interface can be found in Appendix item 4.7.1.

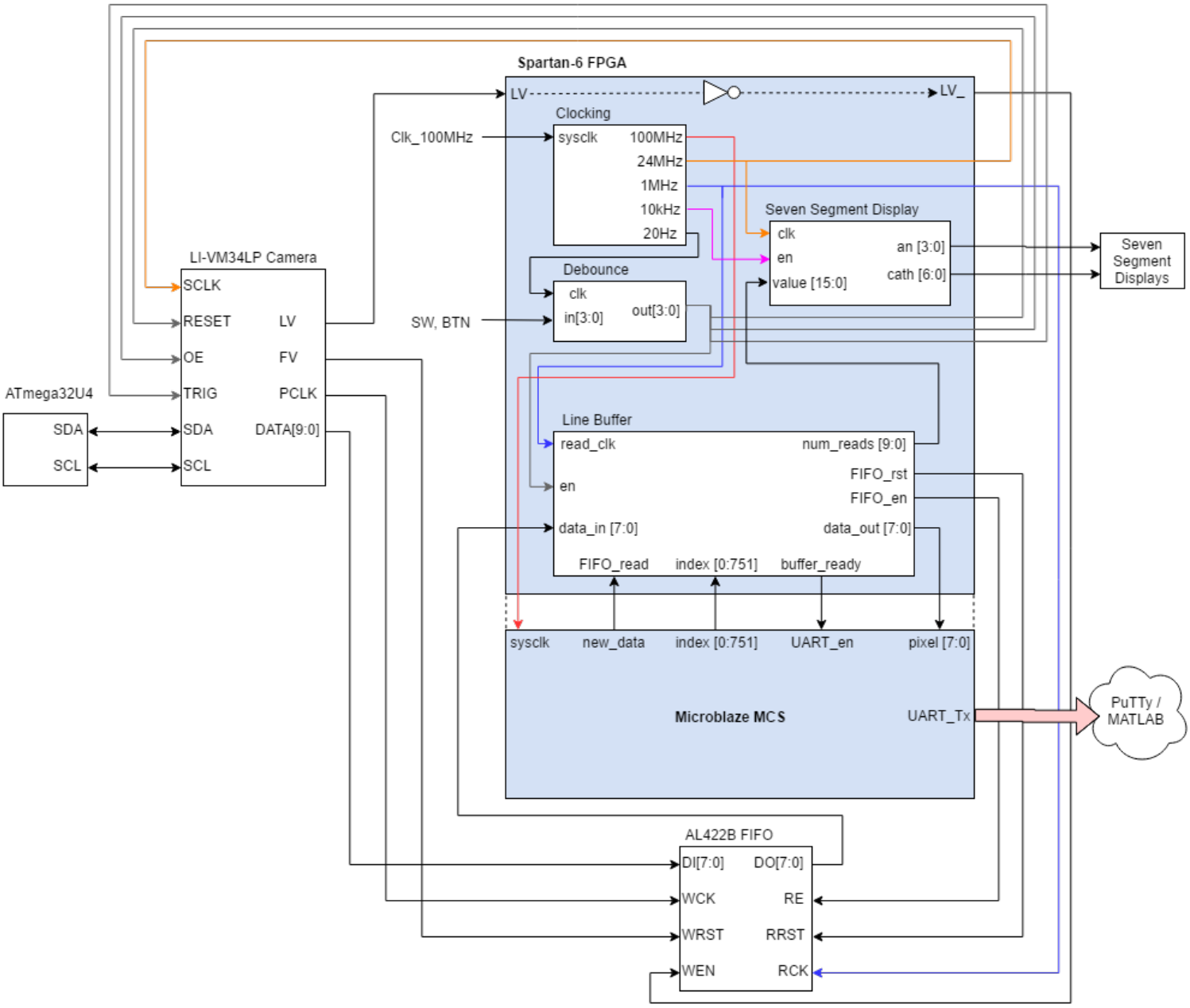


Figure 14: Camera Test System Block Diagram

An example of the transmission of one line of pixel data from the FIFO to the FPGA is shown in Figure 15. The green, purple, blue, and yellow lines in this image represent pixel data, FIFO read enable, read reset, and read clock, respectively. Since the FPGA reads in one line of pixel data at a time, this process will take 752 read clock cycles, as measured in Figure 15. In order to simplify debugging, an internal counter and seven-segment display

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

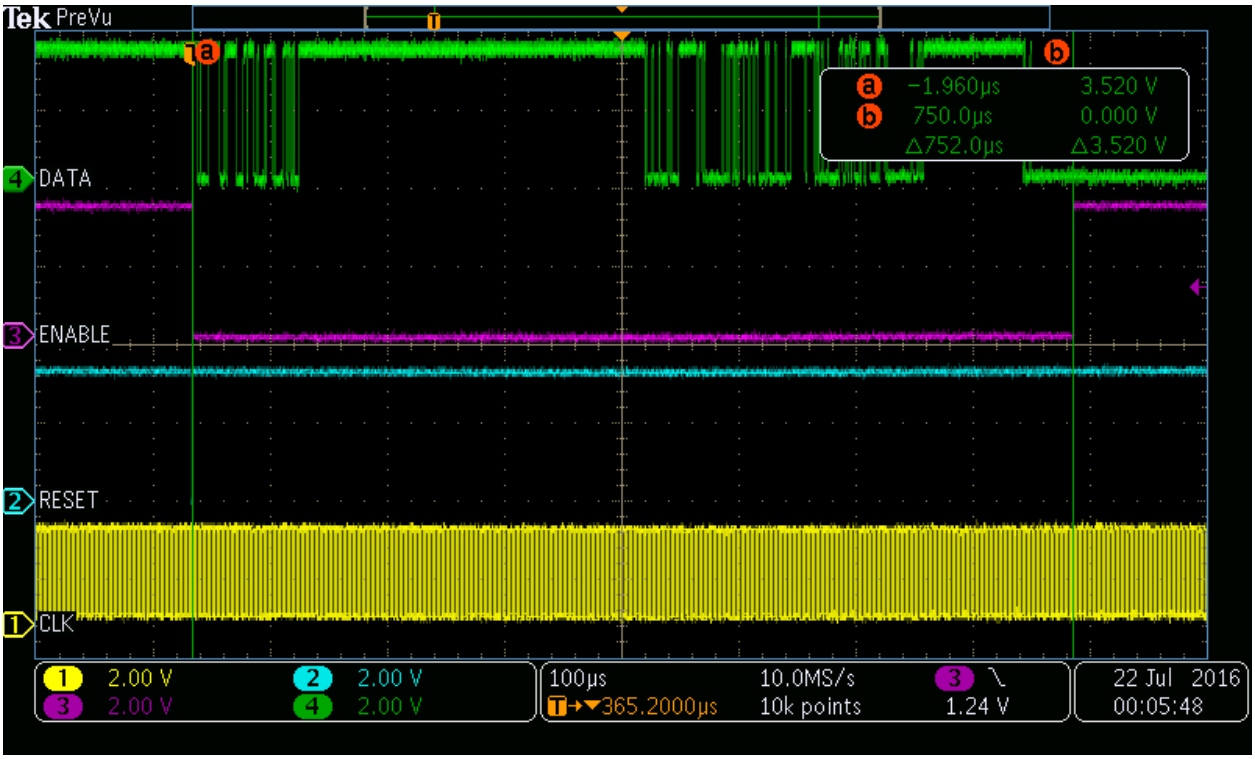


Figure 15: Transferring Line Data from FIFO to FPGA

3.1.4 Transmitting Images Over UART for Analysis

Once the FIFO and FPGA line buffer interfaces were created, the source code found in Appendix item 4.8.1 was implemented on a Microblaze SoC in order to transmit camera line data from the FPGA's internal line buffer over UART. An example of the microcontroller's UART output is shown in Figure 16. The microcontroller will print the value of each pixel followed by a newline and carriage return, starting with the top left pixel in the acquired image.

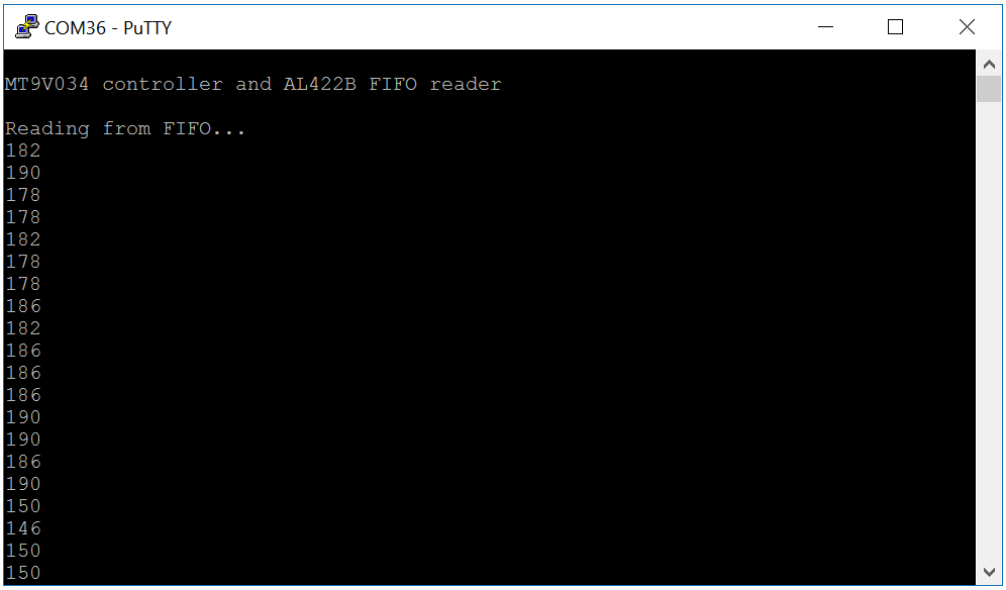


Figure 16: Reading FIFO Data

After the image is received through PuTTy, the MATLAB script found in Appendix item 4.6.1 is used to parse the corresponding logfile into a greyscale image. An example image created through this process is shown in Figure 17. Note that the sub-optimal quality of this image is due to signal interference and degradation in the test setup's long wiring, as shown in Figure 18.

Figure 17: Notebook With Grid and Oscilloscope Leads

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

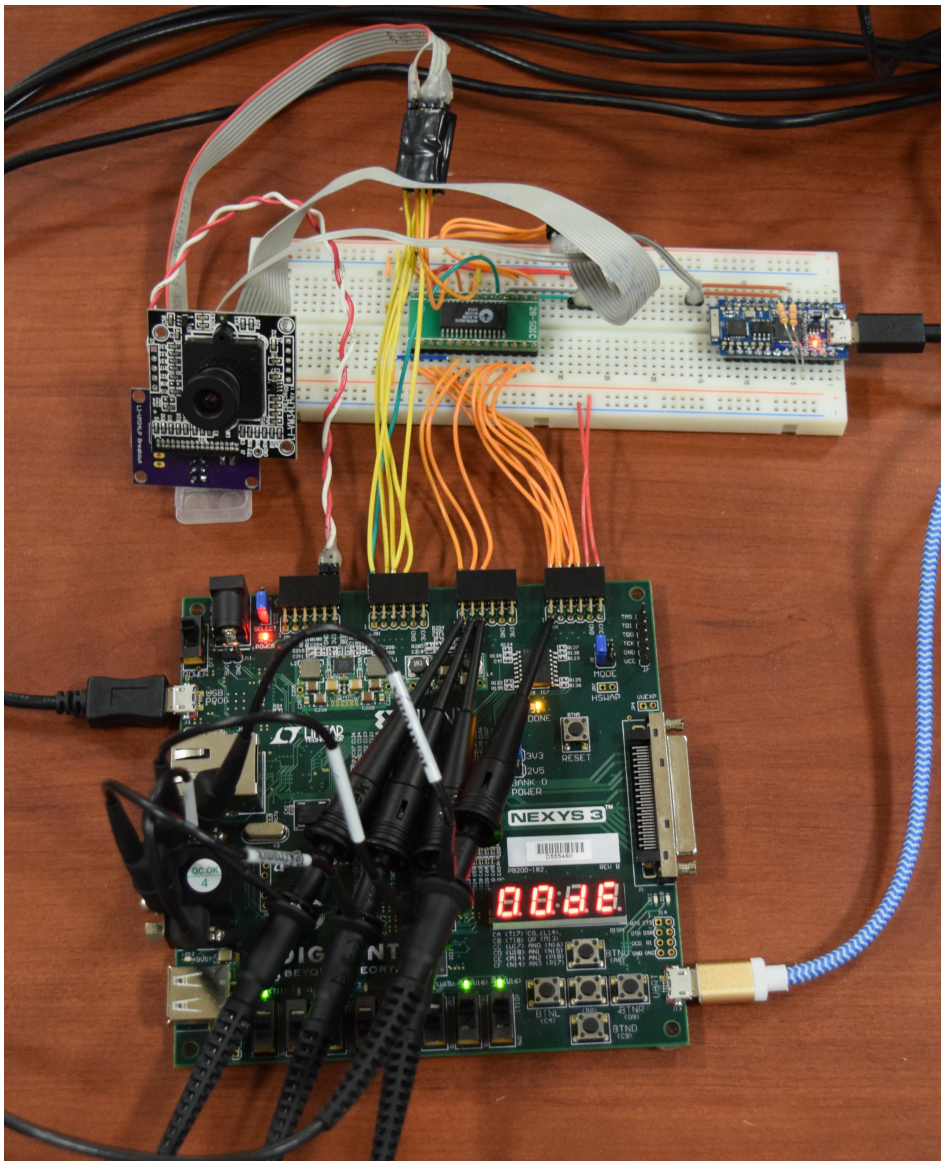


Figure 18: Camera Test Setup

3.2 Final Camera Implementation

After successfully gathering image data from a single camera module, an interface needed to be created for controlling both cameras at once using the ZedBoard. Interfacing each camera module directly to the ZedBoard's GPIO is not feasible, since the pair would consume every available PMOD pin on the board, leaving no additional pins for the IMU or rangefinder[2]. One solution originally investigated was the use of the ZedBoard's FPGA Mezzanine Card (FMC) connector, since it contains 68 available GPIO pins and would be more than adequate for interfacing the stereo cameras with the board. However, the FMC connector has been configured to provide logic voltage levels of only 1.8 or 2.5 volts without modification to the ZedBoard. Since each camera module is only compatible with 3.3 volt logic, the FMC connector is therefore not be feasible for our designs.

This leaves the final option of reducing the overall pin count required by the cameras and interfacing the combined camera setup with the board's PMOD pins. One significant method of reducing the necessary pins required is to include an individual AL422B FIFO per camera. Based on the testing described in the previous section, it has already been determined that these FIFO modules are compatible with the MT9V034 cameras, and are capable of significantly reducing memory requirements on the FPGA. A second major advantage of including these FIFO modules in the camera interface is that their data output lines may be placed in a high-impedance state. This means that the individual data output lines of each FIFO module can be connected in parallel, with a single FIFO driving the lines at a time. Since the bulk of each camera module's required pin count lies in its data lines, the ability to connect these lines in parallel reduces the overall camera GPIO requirements by 8 pins. Since each AL422B FIFO module is capable of being read from at a clock speed of up to 50MHz and the maximum master clock rate of each MT9V034 camera module is 27MHz, the inclusion of the FIFO modules also won't cause a significant decrease in the overall speed of the stereo camera system [1, 8].

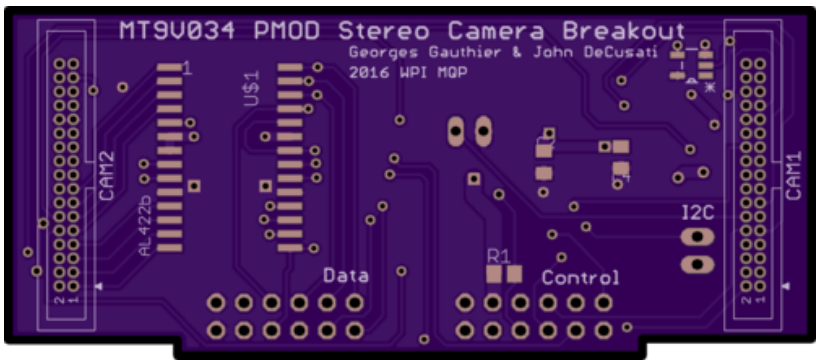
[2] $[2 * (D[9:0] + TRIGGER + OE + RST + SCLK + PCLK + FV + LV)] + SDA + SCL = 36\ pins$

Along with the shared camera data lines between each AL422B FIFO module, it is also possible to connect several other signals in parallel. Since each camera image capture should be triggered at approximately the same time in a stereo imaging setup, it is already desirable to connect both camera TRIGGER lines together. The RST, OE, SDL, SCA, and SCLK lines of each camera module can also be tied together in pairs of two, and the OE lines can simply be held at 3.3 volts. Lastly, since each camera LV signal must be inverted for use with the AL422B FIFOs, a discrete inverter IC may be used to save on FPGA GPIO. Overall, these modifications will save a total of 25 pins, as shown in Equation 2.

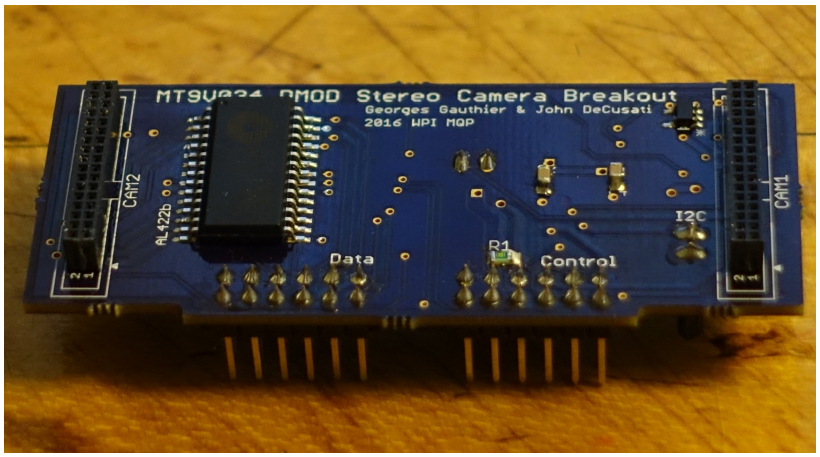
$$\begin{aligned} 36\ Pins - (8\ Data + 4\ truncated\ bits) - (TRIGGER + SCLK + RST) \\ -2 * (OE + PCLK + FV + LV) = 13\ pins\,(!) \end{aligned} \tag{2}$$

Note each FIFO must be controlled individually, requiring an additional Read Reset (RRST) and Read Enable (RE) pin per camera, as well as a shared Read Clock (RCK) line. This brings the total pin count required by the stereo camera setup to 16 pins plus two I2C pins, which is conveniently the number of GPIO available in two PMOD headers. This setup was implemented as shown in Appendix item 4.5, and the final stereo camera breakout board shown in Figure 19 was then created.

A Verilog module was created using a modified version of the MT9V034 camera test code found in Appendix item 4.7.1 and a VGA controller in order to test the stereo camera breakout board. A switch input is used to select one of the two camera modules for image aquisition, and a binned 60x92 pixel set from the center of the camera's image is buffered locally for VGA display. The image is then independently written to the display according to internal VGA timing. This process is repeated at a high rate of speed, allowing for a realtime video stream from the selected camera to be displayed. The assembled stereo breakout board used in this test is shown in Figure 21.



(a) PCB Top



(b) Assembled PCB

Figure 19: Stereo Camera PMOD PCB

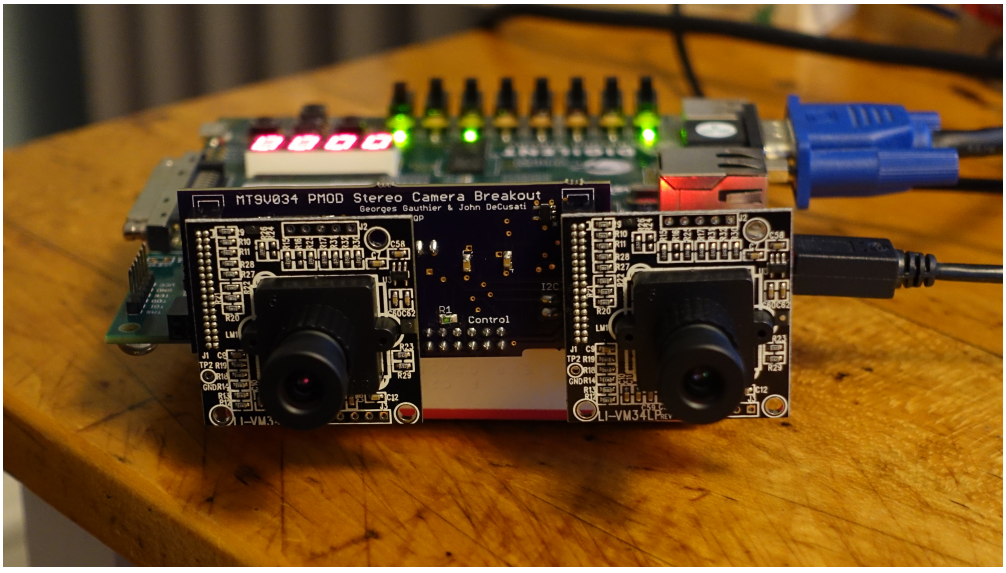


Figure 20: Stereo Camera Breakout Under Test

After attempting to manually focus each camera using the VGA module described above, the code used in Section 3.1.4 was used to transmit image data from the stereo cameras to a computer for further analysis. As you can see from the example image in Figure **??** below, the new stereo camera setup is far less susceptible to data loss in comparison to the previous version.

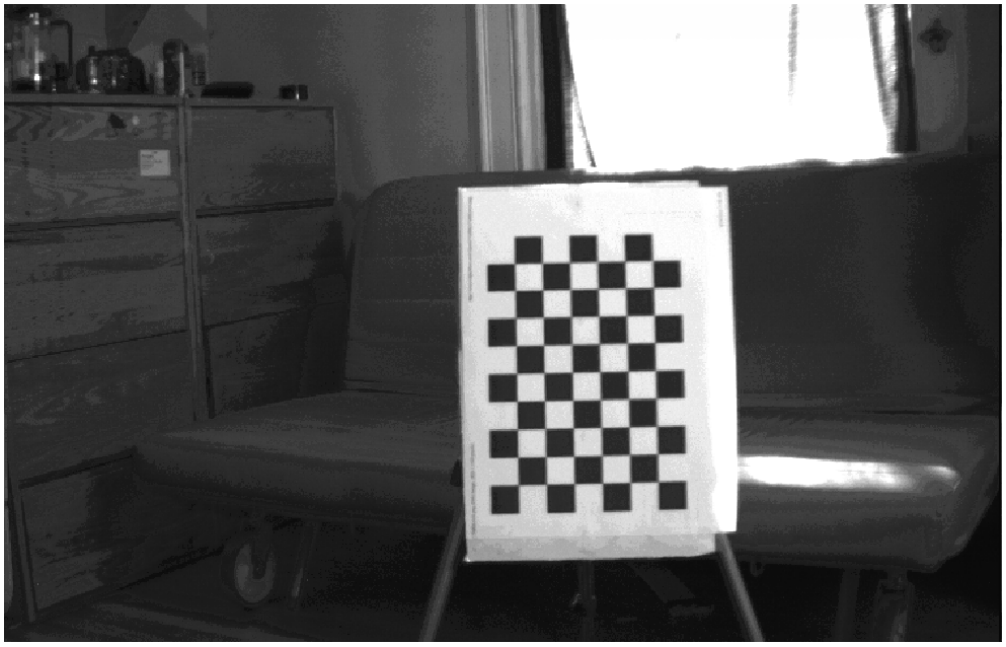


Figure 21: Stereo Camera Breakout Sample Image

For further comparison, please refer back to the test image acquired using the original camera test setup, as shown in Figure 17.

References

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

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

[3] 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`

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

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

[6] 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.

[7] 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`.

[8] 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`.

[9] 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.

[10] 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.

4 Appendix

4.1 Useful Resources

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

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

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

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

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

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

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

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

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

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

4.2 Component Selection

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

† Note that we originally planned to purchase a flir lepton thermal camera module and accompanying breakout board to support two stereo ov7670 camera modules. After experimenting with the ov7670 camera module on our FPGA board, we began to realize that these camera modules are highly limited due to their low frame rate and poor documentation, and realized that we wanted to search for a different camera module. In addition, at a price of $223 for a thermal camera with an 80x60 degree resolution, 25 degree fov, and 7-9Hz image sample rate, we believe that we are much better off spending our money on better camera modules that will be more usable for our task. For more information see Appendix Item 3.

4.3 Camera Module Decision Matrix

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 0-10
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**	**750x480**	**$73**	**No**	**Parallel**	**Global**	**55**	**9**

For purposes of comparison, the thermal camera module we were evaluating is also shown below.

Flir Lepton	9	80x60	$175	Yes, $40	SPI/ MIPI	N/A	50	3

Shown above is our decision matrix for choosing a camera module. Fields marked in green indicate a positive ranking, while red indicates a negative ranking. Based on the individual

rankings of each item's field, we gave our camera modules an overall ranking of 0-10 in the right hand column, with 10 being an extremely high ranking and 0 being an extremely low ranking.

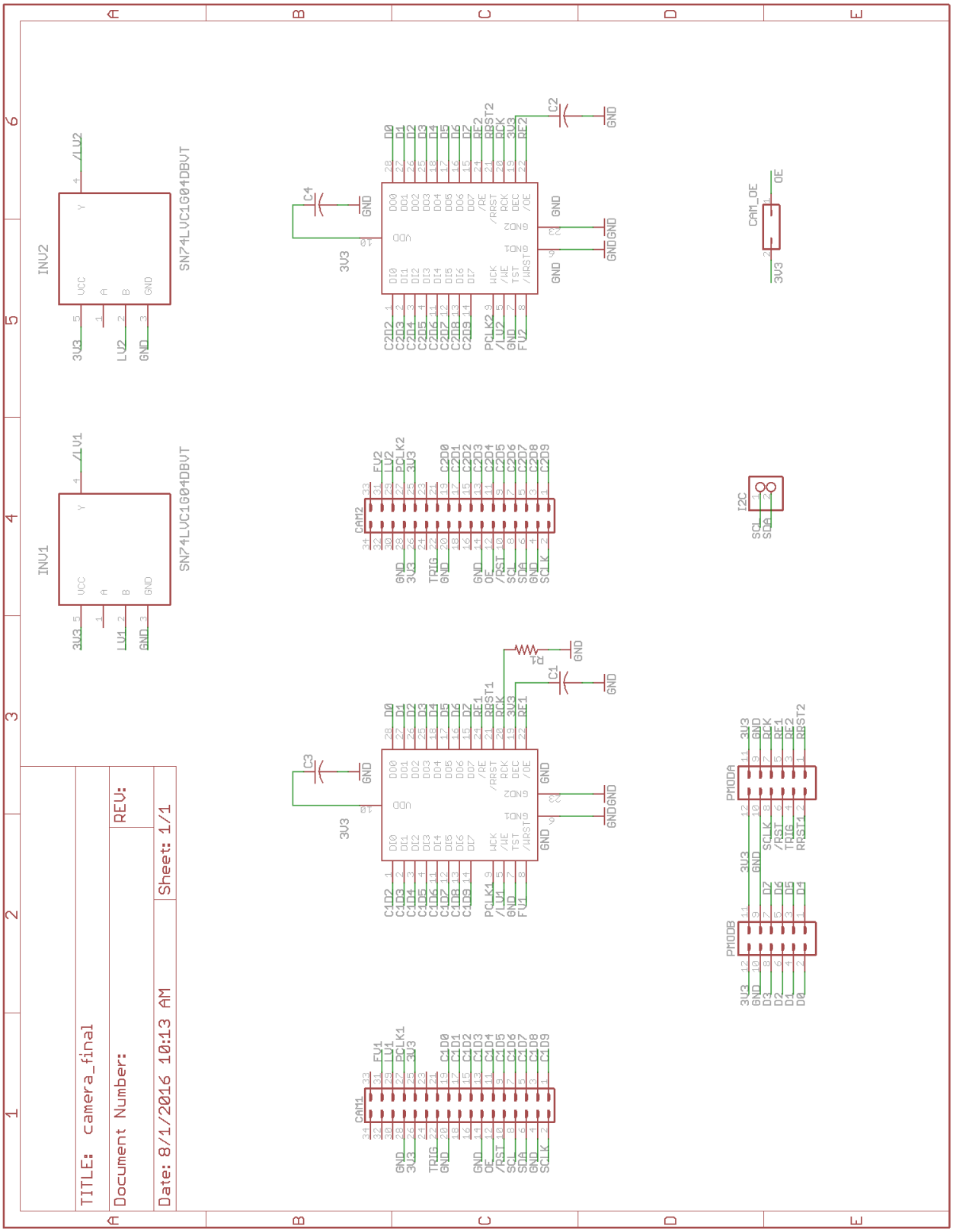
Based on our decision matrix, we believe that it would be worth both our time and money to use the MT9V034 camera modules for our stereo camera interface. These camera modules are the only low-cost global shutter option we've come across, and would be ideal for taking images in a sensor suite that is susceptible to motion. The MT9V034 also uses a parallel data interface and relies on an external clock and shutter trigger, making the module ideal for interfacing with an FPGA-based stereo imaging setup.

4.4 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 [7]

4.5 Stereo Camera Schematic



4.6 Matlab Code

4.6.1 Camera Image Parsing

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

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

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

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

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

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

```
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)
```

4.7 Verilog Code

4.7.1 MT9V034 and Al422b Test Code

Top Module:

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

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

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

```
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
127 // debounce the microblaze input sw
128 wire read_en;
129 btnlatch fifoRead_en(
130     .clk(clk_20Hz),
131     .btn(MICRO_SW),
132     .btn_val(read_en)
133     );
134
135 // camera initialization sequence
136 reg [11:0] init_count = 12'h000;
137 always @(posedge clk_24MHz) // cam sysclk before ODDR2
138 begin
139         if (cam_rst) // if cam_rst is pressed, redo the initialization
              sequence
140                 init_count <= 12'h000;
141         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
142                 init_count <= init_count + 1'b1;
143 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.
//////////////////////////////////////////////////////////////////////////////////
```

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

```
55                                          next_state = read;
56
57                                  prev_state = read;
58                          end
59                  done:
60                          begin
61                                  next_state = ready;
62                                  prev_state = done;
63                          end
64          endcase
65
66  always @(posedge fifo_rck)
67  begin
68                  if(reset_pointer)
69                          begin
70                          fifo_rrst <= 1'b0;
71                          num_lines <= 16'h0000;
72                          end
73                  else if(state==ready) // allow for MCS to read from
                        pixel_line
74                          begin
75                          //pixel_value [7:0] <= pixel_line[pixel_addr];
76                          fifo_rrst <= 1'b1; // make sure read addr doesn't
                             get reset
77                          end
78                  else if(state == init) // prepare to read new data from
                        the AL422 into pixel_line
79                          begin
80                          pixel <= 10'b00_0000_000;
81                          num_lines <= num_lines + 1'b1;
82                          buffer_ready <= 1'b0;
83                          fifo_oe <= 1'b0; // allow for read pointer to
                             increment
84                          end
85                  else if(state == read) // read data in from the AL422
86                          begin
87                          if(next_state == done)
88                                  fifo_oe <= 1'b1; // turn off read enable
89                          if(prev_state != init) // one cycle delay between
                             init and valid data
90                          begin
91                                  pixel_line[pixel] <= fifo_data;
92                                  pixel <= pixel + 1'b1;
93                          end
94                          end
95                  else if(state == done)
96                          begin
97                          buffer_ready <= 1'b1;
98                          end
99  end
100
101 // display number of lines written on 7seg display
102 assign current_line = (num_lines);
103 // allow for MCS to read stored pixel line at given addr if state==ready
```

```
104 assign pixel_value [7:0] = pixel_line[pixel_addr];
105
106 endmodule
```

4.8 C Code

4.8.1 Camera Image Parsing

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

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

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

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


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

    XIOModule gpi;
    XIOModule gpo;

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

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

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

        if((GPI2&SW_READ)!=0){
            swState = 1;
            if (row >= 480) GPO1 &=~(LED);
```

```
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
95                          XIOModule_DiscreteWrite(&gpo,2,pixel_position);
96
97                          // make sure read_sw hasn't been turned off
98                          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;
}
```

4.9 LaTeX Coding Examples

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

4.9.1 Figures

Figure 22: A Test Figure

Using the `\ref` command, I'm able to reference Figure 22 by calling `\ref{wpiLogo}`.

4.9.2 Code Snippet

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

Note that you can also use

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

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

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

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

```
//C code example
#include "someheader.h"
```

```
3  XIOModule gpo;
4
5  void GPOWrite(u8 value){
6      XIOModule_DiscreteWrite(&gpo,1,value);
7      xil_printf("Wrote %d to GPO\n\r",value);
8      return;
9  }
```

4.9.3 Using the bibliography

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

As an example, I can cite *Serveball* at the end of this sentence by calling `\cite{serveball}`.[5]

To cite multiple references, call `\cite{ref1,ref2}`.[5, 9]