Accelerating Augmented Reality Video Processing with FPGAs

A Major Qualifying Project

Submitted to the Faculty of

Worcester Polytechnic Institute

in partial fulfillment of the requirements for the

Degree of Bachelor of Science

4/27/2016

Anthony Dresser, Lukas Hunker, Andrew Weiler

Advisors: Professor James Duckworth, Professor Michael Ciaraldi

Abstract

This project implemented a system for performing Augmented Reality on a Xilinx Zync FPGA. Augmented and virtual reality is a growing field currently dominated by desktop computer based solutions, and FPGAs offer unique advantages in latency, performance, bandwidth, and portability over more traditional solutions. The parallel nature of FPGAs also create a favorable platform for common types of video processing and machine vision algorithms. The project uses two OV7670 cameras mounted on the front of an Oculus Rift DK2. A video pipeline is designed around an Avnet ZedBoard, which has a Zynq 7020 SoC/FPGA. The system aimed to highlight moving objects in front of the user.

Executive Summary

Virtual and augmented reality are quickly growing fields, with many companies bringing unique hardware and software solutions to market each quarter. Presently, these solutions generally rely on a desktop computing platform to perform their video processing and video rendering. While it is easy to develop on these platforms due to their abundant performance, several issues arise that are generally discounted: cost, portability, power consumption, real time performance, and latency. This project develops a Field Programmable Gate Array (FPGA) and System On a Chip (SOC) based solution to address these issues while simultaneously attempting to achieve the capabilities of the desktop solutions. To achieve this goal, the project performs object tracking by highlighting objects that moved in front of the user. Object tracking was chosen because it is a building block that can be used in other applications, and demonstrates the potential of the system. Cost compares at $500 for FPGAs versus over $1000 for desktop computers. Power consumption can be drastically reduced from 500-600 watts to under 10 watts for many video processes, allowing the platform to be portable and battery powered. Video processes on desktop platform generally have a baseline latency of 11-17ms due to frame buffers, but FPGAs can run a video processing pipeline end to end in the micro second domain. These statistics make FPGAs a clear candidate for augmented reality.

The developed augmented reality solution uses two OV7670 cameras mounted on the front of an Oculus Rift Development Kit 2 (DK2). A video pipeline is designed around an Avnet ZedBoard, which has a Zynq 7020 SoC/FPGA. Zynq is a Xilinx product family that has an embedded processor built onto the chip, in addition to a high performance FPGA fabric. The Zynq 7020 has two ARM Cortex A9 processors which run at 866Mhz. The FPGA fabric is customized from the Xilinx Artix-7 product family, featuring 53,000 LUTS, 100K Flip Flops, 5Mb of BRAM, and 220 DSP slices, which is about 2-3 times as large as an entry level Spartan 6 used in classroom environments. The DK2 is a second generation of Head Mounted Display (HMD) developed by Oculus VR. It uses a display from a Samsung Galaxy Note 3 and has a resolution of 960 by 1080p for each eye at 60fps. The OV7670 cameras are inexpensive 640 by 480p cameras that run at 30fps.

The first step of the video pipeline is to capture the raw video from the cameras and repackage it into the internal bus format used throughout the rest of the pipeline. The bus used is AXI Video Stream, a standard developed by ARM and used by most of the Xilinx video processing IP. After this, the video is collected in a frame buffer stored in the external DDR3 memory through a process called Video Direct Memory Access (VDMA). VDMA is applied because the camera input runs at 30 frames per second but the video processing and DK2 output run at 60 frames per second. The frame buffer takes up any slack between these two free running processes. Additionally, it provides a syncing stage for the video to ensure that the rest of the pipeline has synced video.

The video pipeline contains a 4 step process for highlighting moving objects in front of the user, utilizing a delta filter, a threshold filter, a median filter, and a Sobel edge detection filter. These filters progressively alter the camera input video in order to create a mask that can be overlayed on top of the original input in order to highlight moving objects. The filters are all implemented in programmable logic in order to reduce latency. The delta filter is the core filter of the process. It takes two consecutive frames of video and calculates the absolute difference between them to highlight changes between the frames. This reveals areas of the image that are different because objects have moved. The threshold filter takes the results of the delta filter and removes any intermediate values, eliminating noise that may have occurred due to subtle changes such as shadows or lighting. The median filter cleans up the results, removing noise left over from the threshold operation near regions that are likely to be insignificant. Lastly, the Sobel edge filter scans through the video and highlights the edges of the detected motion.

The video pipeline also contains a barrel projection system tuned for outputting to an Oculus Rift Development Kit 2. The DK2 lenses provide a full field of view to the user but in doing this create a pin cushion effect. Barrel projection applies the opposite effect by mapping the radius of a coordinate in polar space using a catmull rom spline curve function. The barrel projection algorithm was also implemented in programmable logic and leverages the Xilinx CORDIC IP to perform transformation between Cartesian and polar space.

The project was completed through a number of methods and work space environments. Many parts of the design were written from the ground up in Verilog, however Xilinx IP Cores were leveraged for parts of the design such as VDMA. Xilinx's High Level Synthesis environment, based on writing C/C++ code to describe hardware, was used for some of the video filters and up scaling. C code targeted for the ARM cores on the Zynq platform was used primarily for configuring the video pipeline at run time. Significant amounts of testing and simulation of the video filters was performed in MATLAB before writing them in Verilog. After writing Verilog, many components were tested in the Xilinx Simulator (XSIM) using Verilog test benches. Finally, design components were added to a main video pipeline project and tested on the hardware with the assistance of lab equipment. Unrelated to the FPGA work, some PCB design was performed in Altium to ensure a robust method for attaching the cameras to the ZedBoard.

Several aspects made this project very challenging. The first was the learning curve: none of the students working on this project had ever done any work related to video processing on FPGAs before this project. The VDMA system is a good example. It was one of the first systems to be added to the project and took a month longer than expected to complete due to constantly revisiting the research phase to learn more about the technology. This project was also working in new frontiers of FPGA application. As far as the team is aware, no one has documented designing a barrel projection algorithm on FPGAs before. The project overall is within a very new field and applying FPGAs to augmented reality is not a common task. Throughout the project, the limitations of the Xilinx development environment Vivado were constantly encountered. Source control was added to the project through complex TCL scripting to rebuild the project within Vivado each time it was retrieved from the Git repository. Challenges with meeting timings requirements were constantly encountered and the team is still not sure if the aggressiveness of the design or the capabilities of Vivado synthesis algorithms were creating timing issues. Finally, testing in hardware is challenging and in several cases was the only method available. Towards the end of the project, the video pipeline took over forty minutes to synthesize and implement, which made iterative development difficult. The other challenge to testing on hardware was the lack of feedback after a test fails. Usually video would not output from the system, and while it was easy to point to the change that broke the system, the reason this change did not work was excruciatingly difficult to identify.

This project successfully implemented a working end to end video pipeline and showed that FPGAs are viable platforms for developing augmented reality. Several items were recommended for future work. An implementation of a more useful, marketable application of augmented reality is highly recommended, such as overlaying rendered video onto a surface in a real world environment. There were a number of limiting issues encountered in the project. The way in which the HDMI output chip on the ZedBoard is connected to the Zynq limits the color space choices available and created color space issues within the project. Higher quality cameras with a higher resolution and higher frame rate would increase the quality of the augmented reality experience. In addition a camera with external fsync capability would greatly improve the performance of the video pipeline by eliminating the need for frame buffers and vastly reducing latency. Redesigning the video pipeline to run at a higher frame rate would further increase the performance of the system. Upgrading the project to the Oculus Rift consumer version would have similar performance capability improvement. Switching the project to using a transparent display such as the Microsoft Hololens would drastically reduce the bandwidth needed on the video pipeline and avoid working with full resolution color video. Full resolution color video would not be a necessary part of the pipeline since the user could see through the display to the real world. In addition, designing the system to be wearable was not within the scope of the project, but would greatly improve portability and marketability.

Contents

List of Figures

1 Introduction

The year 2016 has seen a great advance in the number of virtual reality (VR) systems being brought to market, with over 40 exhibitors demonstrating VR technology at the 2016 Consumer Electronics Show [1]. Virtual reality is an interactive computer system, which replaces some of a user's senses with other inputs in order to immerse them in a virtual environment [2]. A virtual reality system can take many forms, however the form most readers will be familiar with are commercial head mounted displays (HMDs) such as the Oculus Rift or HTC Vive as seen in Figure 1. At current time, these commercial systems are mainly used for gaming, however they have potential for many other uses, including the implementation of augmented reality systems. Augmented reality is an extension of virtual reality where virtual stimuli are superimposed onto the real world rather than creating a totally virtual world [2]. An example of where this could be useful is providing information to doctors performing surgery by adding useful information on top of a live video feed [2]. Another application would be in tourism where a system could show interesting information about a site a user is observing [3]. Unfortunately, this technology is still expensive as it requires the purchase of a desktop PC to power the system in addition to purchase of the virtual reality hardware [1]. This also increases the power usage of the systems, and decreases the portability.

Figure 1: The HTC Vive, a commercial VR display [4]

This project investigated implementing a virtual reality system on an FPGA instead of a desktop PC. FPGAs allow for efficient parallel processing of video, as well as lower power consumption and size compared to desktop PCs, making them an ideal platform for one of these systems. In addition, they offer the potential to greatly reduce latency between when an input is received and when the result of this input appears on the display.

This project investigated whether FPGAs are a viable platform for augmented reality. To achieve this goal, the system features cameras mounted on a HMD, which displays the output of the system. The camera video is passed through to the HMD after the system has processed and augmented the video feed. The system features two cameras in order to provide 3-D vision to the user. For video processing, the team decided to perform object tracking by highlighting objects that moved in the video. While this is not useful on its own, it is a building block that can be used in other applications, and demonstrates the potential of the system. Other goals of the project included ensuring the project operates in real-time and reducing latency as much as possible. These were important because latency in augmented and virtual reality systems can lead to nausea in users as what they are seeing conflicts with their other senses [5]. The project also aimed to reduce the cost and power consumption of an augmented reality system, which was done implicitly by using an FPGA rather than a desktop computer.

This project uses an Avnet ZedBoard, which is powered by a Xilinx Zynq 7020 system on a chip [6]. This chip includes both programmable logic and a dual core ARM Cortex A9 microprocessor. This is an advantageous configuration because it allows high performance CPU code to be run alongside custom hardware. Work initially developed on a CPU can be accelerated by offloading complex processes to the FPGA, and applications based on FPGAs can take advantage of the wide variety of drivers and devices that can be run by a CPU. The ZedBoard also contains 512 MB of off chip DRAM memory, and several peripheral ports [6]. It also features an HDMI port with a HDMI transmitter chip, which is used to output video [6]. In addition, the board has six PMOD ports, which are a general purpose digital I/O interface developed by Digilent [6]. This ZedBoard is fed input from two OV7670 cameras, which are inexpensive cameras that interface using an I^2C bus interface for control, a raw formatted single ended digital bus for video, and a connection to the PMOD ports on the ZedBoard. In addition, a custom PCB was used to map the ports on the camera to the PMOD standard, and ensure a reliable and robust connection. For the output device, an Oculus Rift Development Kit 2 (DK2) is used. This interfaces with the system using HDMI, and receives power through USB.

The paper starts by discussing the background behind virtual and augmented reality, and the hardware being used in this project. It also discusses prior work that has been done on FPGAs in terms of both VR/AR and general video processing. The following section gives a high level view of the system architecture and video pipeline. It is then expanded upon in the next section with more detail about how each system is implemented. Following that, the development and testing process is discussed as well as the results from this testing. Lastly, conclusion and future ways to expand upon the project are detailed.

2 Background

2.1 What are Virtual and Augmented Reality?

Virtual reality is the concept of creating a 3-D virtual world for a user to experience and attempting to convince the user they are actually a part of that world. This concept is a extension of simulation, 3-D video games, and 3-D movie experiences. Augmented reality is an extension of virtual reality, where instead of artificially creating a 3-D world for the user to experience, the real world is used. Either through capturing the world with cameras (as done in this project) or projecting video on a see through screen (as seen in the Microsoft Hololens and Google Glass) the real world is augmented in some significant way to change how the user experiences the world.

2.2 Applications for Virtual and Augmented Reality

There are many practical applications for virtual reality and augmented reality. Currently the most popular application for virtual reality is in the entertainment industry, using virtual reality platforms as a way to bring the user further into the experience. Video games such as Subnautica from developer Unknown Worlds and Time Machine from developer Minority Media allow the player to explore a world using the Oculus Rift as a screen and using the various other sensors on the system to move around in the world.[7][8] Additionally, movies have begun to support virtual reality systems in order to better the experience of 3-D movies. There are additional proposed applications in virtual reality ranging from education to construction.[9]

Augmented reality has many real world applications aside from the entertainment industry. Heads up display systems, which are an implementation of augmented reality, have been used in fighter jets for over 50 years and in consumer cars for over 20 years.[10] The applications in fighter jets are quite complicated, being able to track other aircraft as well as warn the pilot of incoming attacks. However, in order to achieve this, very large and powerful computers are used to process inputs from the space around the aircraft. For heads up displays in cars, applications such as displaying mileage and speed of the car are very simple and therefore can be performed on small, low power computing systems. However, as computers have become smaller, more powerful, and more power efficient, being able to do more processing for augmented reality is becoming common. Continental is just one of many companies experimenting with increasing the information that can be displayed to the user such as directions, road conditions, and other obstacles on the road. A marketing sketch of their product is shown in Figure 2.[11]

Figure 2: Marketing design for Continental Heads-up Display[11]

2.3 What is an Oculus Rift?

The Oculus Rift was designed as a platform to implement virtual reality. The device itself is a head-mounted display featuring two screens positioned in front of the eyes (one for each eye). There have been multiple release versions of the Oculus Rift; however for the purposes of this project, the only relevant versions are the Development Kit 2 (DK2) (the version used in this project and shown in Figure 3) and the Consumer Version 1 (CV1) (shown in Figure 4) which is the first consumer version of the product that had just been released as of the writing of this paper. Both versions feature the displays mentioned before, as well as IR sensors that can be used to detect the user's head movement while they are wearing the device. Additionally, as with all head-mounted displays such as the Oculus Rift, the device contains lenses between the user's eyes and the screens in order to provide the user with a full field of view. This distorts the images on the screens and therefore, in order to make the images look normal to the user, additional processing has to be done on the output video (see more in Section 4.6).

Figure 3: Oculus Rift Development Kit 2 (DK2)[12]

Figure 4: Oculus Rift Consumer Version 2 (CV1)[13]

The major difference between the CV1 and DK2 are the CV1 has built in audio capabilities, a higher resolution screen, and higher frames per second (FPS). Therefore, in terms of system implementation outputting to the Oculus Rift is similar between the two releases.

In order to implement virtual reality systems for the Oculus Rift, large, expensive desktops are normally required. The official minimum recommended specifications for the Oculus Rift (both CV1 and DK2) are a NVIDIA GTX 970 ($300-350), an Intel i5-6400 ($200), and 8GB RAM. A computer with these specs would average around $900 for minimum specifications to run the Oculus Rift and $1000 for a quality system.[14]

2.4 The ZedBoard and Zynq chip

The ZedBoard is a evaluation board developed by Avnet for the Xilinx Zynq-7020 All Programmable SoC (System on a Chip) (shown in Figure 5).

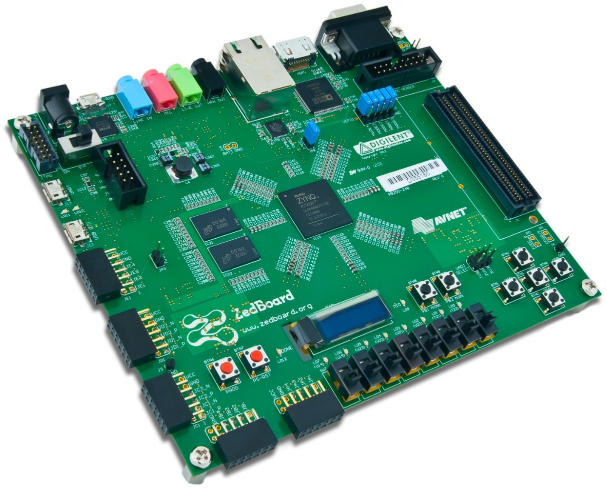


Figure 5: The Avnet ZedBoard

The significant part of these line of chips from Xilinx is that the chip contains both programmable logic fabric and physical CPU cores on the same die. This allows users to utilize the advantages of using both

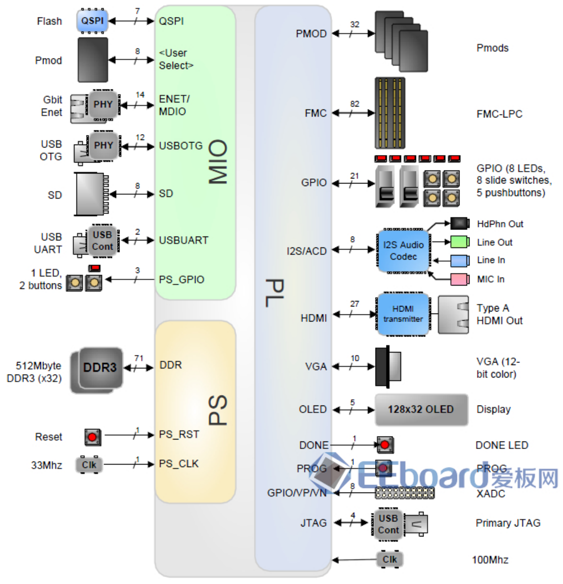


Figure 6: IO Diagram for the Avnet ZedBoard

types of computation. On the board used in this project, the Zynq chip contains two ARM Cortex A9 chips running at 866MHz, 85K logic cells, 560 Kbits of block RAM, and 220 DSP slices. In addition to the onboard bloack RAM, the ZedBoard features 512 MB of off-chip DDR RAM which interfaces through the ARM core.

The ZedBoard itself features 5 PMod ports (2 of which are differential, which is not important for this project), 8 LEDS, 8 switches, and 7 buttons. The board also features an HDMI port which is powered by an Analog Devices ADV 7511 transmitter chip and a VGA port [6]. The HDMI port was used in this project to connect to the Oculus, and the VGA port was used to debug video output in early stages of the project. The board contains other IO on board (as can be seen in Figure 6) however, only the aforementioned IO is used for this project.

2.5 Video Processing on FPGAs

Video processing on FPGAs is a well researched and proven area of hardware design. Techniques such as facial recognition, surface detection, and edge detection have all been successfully implemented on FPGAs.[15][16] Therefore, the video processing part of this project is well defined and known to be possible. However, in order to demonstrate a working system, a video processing system needed to be implemented to have a proof of concept. For the purposes of this project, object tracking was chosen as the video processing technique to be implemented (see subsection 4.4).

2.6 Related Work

Several systems have implemented augmented reality, including several that are currently in development. Two notable systems are discussed below.

2.6.1 Microsoft Hololens

The Microsoft Hololens is a mobile system currently being developed which augments the world for a user, allowing them to run Windows programs on the surfaces around them. The Hololens is a head mounted device similar to the Oculus Rift, however it utilizes a see through display which display augmentation as a ”holograph” on top of the real world (see Figure 7).[17]

Figure 7: The Microsoft Hololens Device

This see through display means there is no latency showing the user the physical world unlike Oculus Rift, where video of the world must pass through a pipeline before being shown to the user. This type of technology is aimed at improving the user experience by removing many of the latency issues that cause users to feel motion sickness. Additionally, it reduces the computation and power requirements for the system. The Hololens system uses a custom ASIC in order to do video output to the transparent screen, but uses a powerful CPU to do the computation.

2.6.2 LeapMotion

LeapMotion, as pictured in Figure 8, is a camera device that can be mounted to the Oculus Rift and interfaces with the computer powering the Oculus Rift in order to provide a augmented reality experience for the user. Originally, this device was meant to track user hand and arm movements to allow the user to use their hands as control mechanisms for a computer. The LeapMotion device is a simple camera and by mounting it to the front of the Oculus Rift a computer can process this input and display it to the user. [18]

The Leap Motion works with the Oculus Rift SDK when used in augmented reality applications and must be connected to a desktop in order to function. This means the device is only useful for augmented

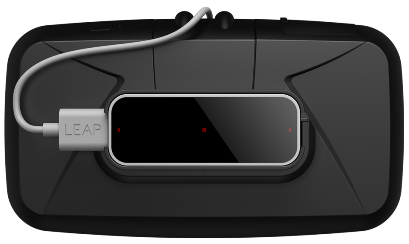


Figure 8: The Leap Motions Device Mounted on an Oculus Rift

reality applications that allow a user to be near a desktop computer, limiting the mobile capabilities of the device. [19]

3 System Design

The system was designed to process video with as little latency as possible. For this reason, the team decided to try and put the majority of the modules in programmable logic (PL) rather than on the processing system (PS). This is because the PL is very efficient at doing parallel processing in real time. There is also a latency associated with moving video into and out of memory so it can be modified by the CPU, so these transactions needed to be minimized. When video comes from the camera, it must be either processed or buffered immediately as the camera will not stop capturing data. For this reason, the first goal of the system was to buffer the video in memory. This reduces the pressure on later video processing blocks since there is not a risk of losing parts of the frame. Next, video was processed to add the augmented reality component. It was decided to do the processing here so that it was running on the original resolution video from the camera, which reduces the processing power needed compared to processing the upscaled image. Last, the video had to be converted and output so that it could be viewed by the user.

The overall system design is divided into several functional blocks as seen in Figure 9. These include two hardware blocks, displayed in light blue, several blocks implemented in programmable logic on the Zynq chip, displayed in green, as well as a memory interface which uses the ARM processor on the Zynq chip, displayed in dark blue.

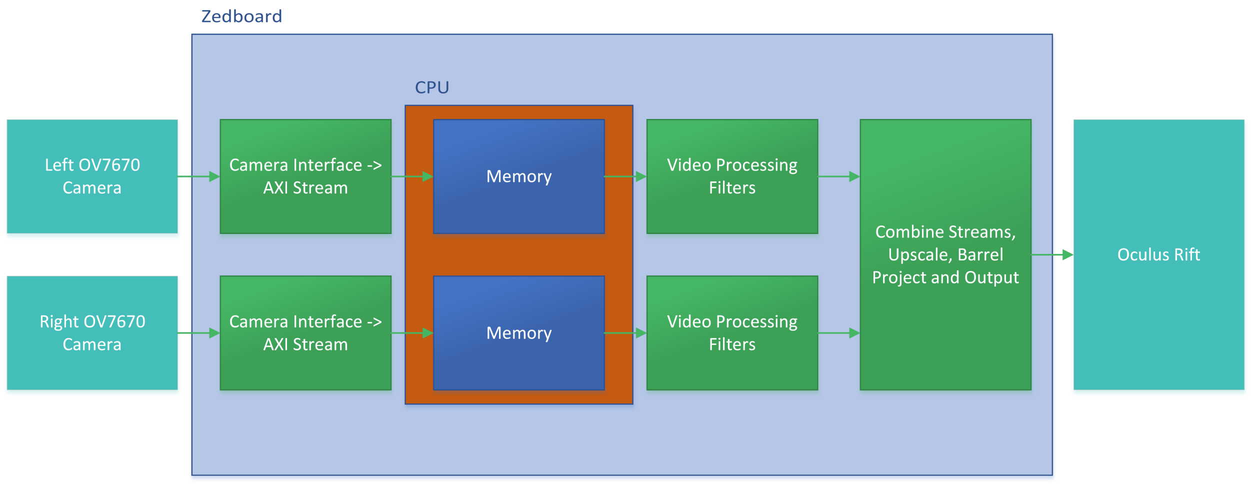


Figure 9: Overall System diagram

The first stage in the overall system is the input, which takes place in the OV7670 and Camera Interface to AXI Stream blocks. The first component is the OV7670 which is an inexpensive camera used to capture video input. This camera was chosen mainly for its price, and for the ease of interfacing with it. It is connected over a PMOD interface using a custom PCB to map the pins. It captures video at a resolution of 640x480. It is programmed, and data is captured through a camera interface block which is written in

VHDL. The block is responsible for programming the camera's on board registers through an I^{2}C interface and capturing video data from the camera. It captures the data 8 bits at a time and combines two groups of these 8 bits together into one 16 bit pixel. Last, the video signal is sent through a Xilinx IP block which converts it into an AXI stream. This allows stock Xilinx IP blocks to interface with the video. The input to the system is discussed in more detail in Section 4.1.

During the next step in the pipeline, the video is saved to a frame buffer located in the DDR3 memory, which is external to the Zynq chip. This is done in order to allow the video processing modules to block if needed while processing data. Since the camera data is being stored they don't have to worry about parts of the frame being lost if they don't process it quickly enough. The memory buffer is triple buffered, which eliminates any tearing that may result from the frame changing while it is being read or written. On the Zynq chip, the memory access is controlled by the on board ARM Cortex A9 CPU, meaning that the programmable logic and CPU must work together in order to buffer video. On the programmable logic side, the module uses the Xilinx video direct memory access (VDMA) IP. In the CPU, bare metal C code interfaces with this block and controls memory access. More details about the buffering and VDMA is in Section 4.3.

After being retrieved from the video buffer, the video is sent through various processing filters in order to augment the video with additional information. While in this proof of concept project, this module only highlights the difference between frames of video, other processes could be added here in the future that are specific to a certain application of the system. In this implementation, the video is passed through a delta filter, a thresholding filter, a median filter, and finally a Sobel filter. The delta filter, thresholding and median filter are written in Verilog, while the Sobel filter was written in HLS using the HLS video library. For more information on the video processing see Section 4.4.

The last step of the system is to prepare the video for output and then output it to the DK2. Before the video can be output it has to be upscaled from the input resolution of 640x480 per eye to the output resolution of 1080x960 per eye. The video must then be barrel projected to correct for the pincushion distortion caused by the Oculus Rift lenses. Last, the streams from the two cameras, which have been processed separately, are combined. These processes are discussed further in Sections 4.5 and 4.6.

After the output processing, the video is ready to output to the DK2. The DK2, as discussed in Section 4.8, takes in video over HDMI, so the video must be converted from an AXI Stream to HDMI. In order to convert the video, the AXI Stream is passed into a Xilinx IP block called AXI Stream to Video Out. This block also takes video timings from a Xilinx video timing controller (VTC), which is programmed by the CPU. After the stream is converted to a video out signal, it is output to HDMI using an Analog Devices ADV7511 on the ZedBoard. This chip takes the video signal and converts to HDMI, before directly outputting to the HDMI port. The ADV7511 is programmed over an I^{2}C interface controlled by the CPU.

More information about the video output process can be found in Section 4.7.

4 System Implementation

This section will go into detail about how each part of design works. Each technology is described from a general standpoint, then how it was applied to the requirements of this project.

4.1 Camera and Camera Interface

The work done with the camera for this project consisted of three parts; altering the camera settings (registers), altering the camera capture logic, and translating the camera timing signals (href, vsync) to work with AXI Streams and Video Direct Memory Access (VDMA) (see Section 4.3).

4.1.1 Camera Settings

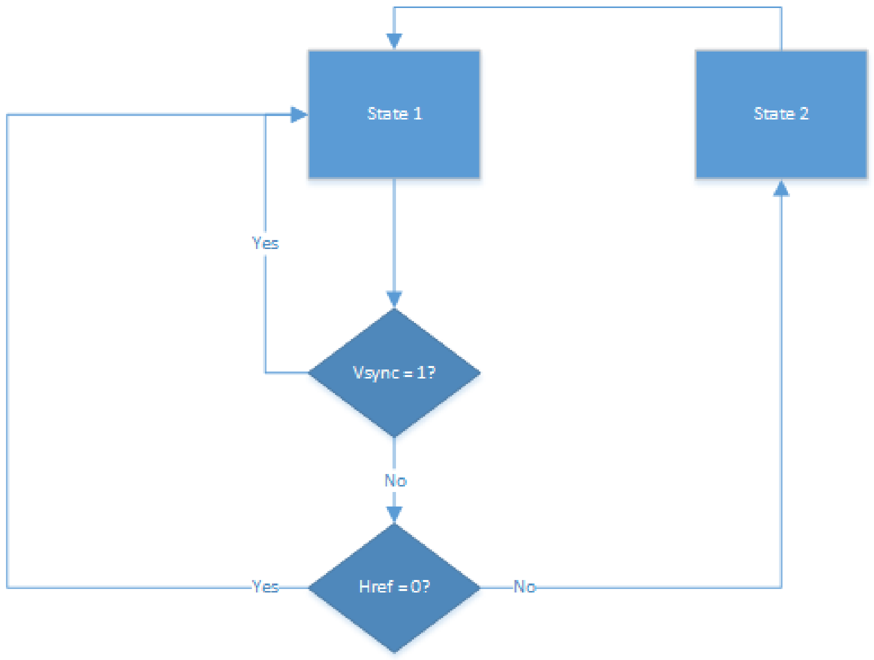
The OV7670 has 202 registers that provide various settings for how the camera functions [20]. Even though the functions of some of these registers are documented, many are documented as "reserved" but still impact how the camera performs. Undocumented registers were difficult to work with as the proper settings for this project had to be found through trial and error.

The main register settings that were modified were the video output format (COM7, COM13), to convert to YUV output, and data output range (COM15), to use the full scale output of 00 to FF. Most other register settings were either left as default value or changed to recommended values from online references (see Appendix D for more details). Another key register is COM17, which provides an internal test pattern generator that can be used for testing color and timings. For a detailed list of all register settings see Appendix D.

4.1.2 Camera Capture

The initial camera capture logic was used from a similar project which built a system for outputting from the OV7670 camera module to a VGA display [21]. This interface was used to capture RGB data from the camera. The HDMI output for this project uses YUV, therefore the capture logic was redesigned to use YUV.

The camera logic uses a 2 state state-machine to keep track of what data is currently being sent. The first data packet sent after href goes high is dictated in the COM13 camera register, which is configured to send the U chroma value first. The data from then on will be in a known sequence of U (first value) ,Y,V,Y. However, because the video-in-to-AXI IP only expects a pixel package of either YU or YV, the camera capture logic does not need to keep track of whether it is outputting U or V. Figures 10 and 11 show this behavior.



State	Function
0	• C = data • WE = 0
1	• Y ¡= data • WE = 1

Figure 10: State Machine for Camera Capture

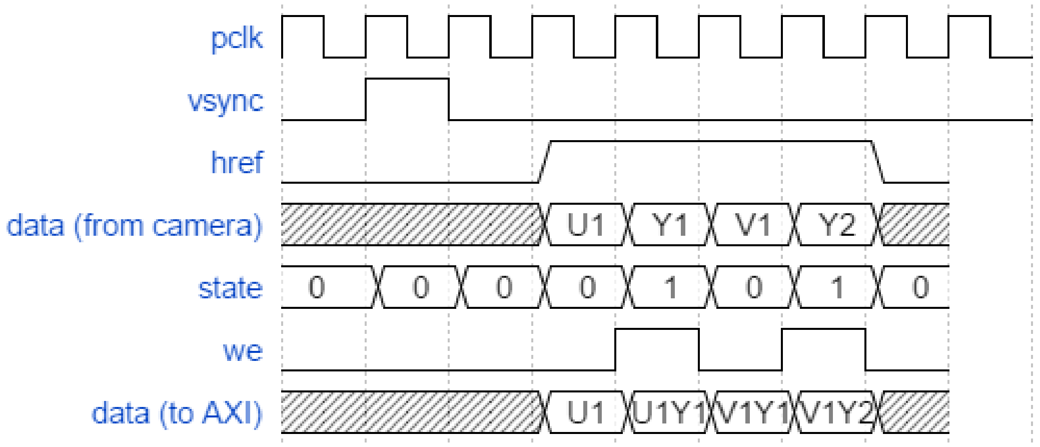


Figure 11: Timing Diagram for Camera Capture

The state machine increments its state on every rising clock edge of PCLK. It resets to state 0 when vsync is high (new frame) or href is low (data for the row is not active yet). After which, on state 0, it assigns the data to an internal register C and sets the write enable to low, and on state 1 is assigns the data to an internal register of Y and sets the write enable to high. The output is then assigned to be C,Y according to the data expected from the video-to-AXI stream.

4.1.3 Camera Input Timings

The video IP used for DMA and Video Processing uses the AXIS-V format (see section 4.2.4). The Video In to AXI4 Stream core is designed to convert standard computer video inputs such as DVI, HDMI, or VGA to the AXI format. Unfortunately, the camera does not conform to one of these computer video standards. The camera outputs only a few timing signals: Pclk, Vsync, and Href. The video core needs Pclk, Active Video, Hblank, Hsync, Vblank, and Vsync. Figure 12 shows how these timing signals work. In the case of syncing, there are options to use only the blank signals, only the sync signals, or all 4 signals for timing detection. Some adaptation between signals was needed in order to create the timing information for the Video In to AXI Stream core.

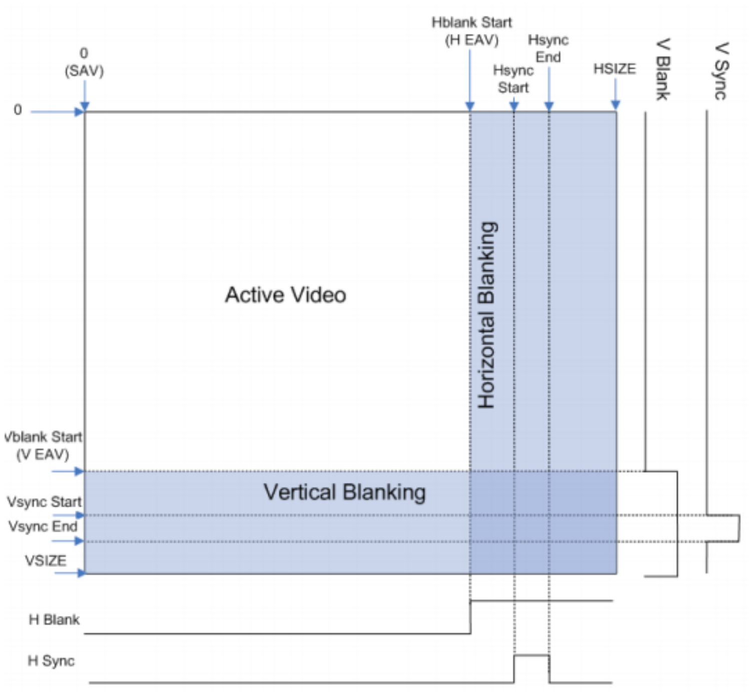


Figure 3-2: **Definition of Timing Variables – Falling Edge of Blanks**

Figure 12: Video Timing Signals [22]

The adaptation used in the design alters and re-purposes the camera timing signals to conform to the inputs of the video in core. For Pclk, due to the methods used for camera capture, half of the camera's pclk is used. This goes back to how the camera caputre logic only produces a pixel every other clock cycle. For Active Video, href is used. The href signal is active during the valid data of each row, so it is an acceptable replacement for Active Video. For hblank, the inverse of href is used. This signal is high whenever the data is not valid. To get hsync, an edge detector is placed on href. Hsync rises for a short pulse during the horizontal blanking period, after a row is finished. Only one high clock cycle of href is allowed to pass. For Vsync, the camera's vsync is used. Vblank is not used by the video-in core and is simply passed through for timing detection, which is not used in this project. The mapping of camera signals is shown in Figure 13. These connections are duplicated for both cameras in both sides of the pipeline

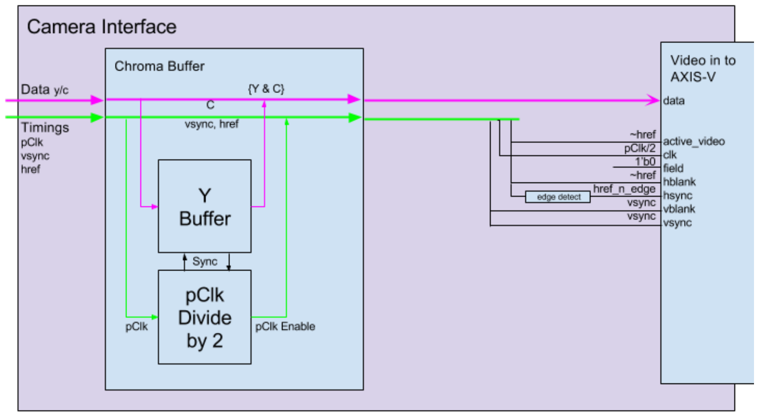


Figure 13: Video Input Block Diagram

4.2 Memory Interface and AXI Streams

In order to move data between different processes on both the programmable logic (PL) and programmable system's (PS) on the FPGA, various types of Advanced eXtensible Interface (AXI) buses are used. The AXI standard was developed by ARM under the Advanced Microcontroller Bus Architecture (AMBA) and contains several similarly-named but functionally different bus standards: AXI 3 and 4, AXI4-Lite, AXI4 Stream, and AXI4 Video Stream. These buses are integrally involved in most Xilinx Intellectual Property (IP) cores made for processing sequential data, including the Video Direct Memory Access core (VDMA) discussed in section 4.3.

4.2.1 AXI 3, AXI 4, and DMA Transactions

AXI 3 and its generational successor AXI 4 are both standards intended for transaction based master-to-slave memory mapped operations. In this design, they are used for the transfer of data between the video pipeline in the Programmable Logic (PL) side and the main memory on the PS side. Some device controls also use AXI 3/4 as a trunk before being reformatted into individual AXI4-Lite interfaces. AXI 3 is hard wired in the PS since the PS is an ”off the shelf” ARM Cortex A9 processor and was not modified to upgrade to the newer AXI4 standard. Most interconnects on the PL side are done in AXI 4, with the exception of interconnects made to the PS.

Through AXI, a read or write to main memory is implemented as a periodic transaction originating from the PL side of the design. A Direct Memory Access (DMA) controller fabricates an AXI compliant transaction and outputs it on the master bus. In the design, the DMA controller is a Xilinx IP core: the Video DMA (VDMA) controller. Data flows from the master bus to the PS memory controller. The PS memory controller is an arbiter for all the master buses on the PL and multiple other master interconnects for the ARM cores, caches, and snooper. When a transaction appears on any of these interfaces, it will be executed upon main memory.

In the case of writing to main memory, an AXI transaction contains the memory address to write to and a payload of configurable bit width/burst length. In the case of reading, the bus master (i.e. the DMA controller) provides an address and gives the slave device permission to use the bus to reply back with the data.

As might be implied, the AXI bus contains many signals and control flags. While there are timing diagrams for all the eccentricities of this bus, fortunately all of this has been abstracted away by Xilinx IP. All the interactions using a full AXI bus in the design are covered by Xilinx IP, and thus when considering using this bus one need only see to the proper configuration of IP and that the throughput and latency are sufficient. One need not worry about low level timings and interface operations.

4.2.2 AXI4-Lite

AXI4-Lite is a substandard of AXI4 that is intended for the configuration of memory mapped peripherals. Peripherals in this sense are any IP or hardware that is controlled by a processor such as a Microblaze or the PS system. AXI4-Lite lacks the extensive flags and timing conventions of full size AXI buses, using simpler logic that is feasible for frequent implementation in large designs with many peripherals. Thus it is not compatible with some of the more complex memory transactions (i.e. Bursts, Scatter-Gather) that full AXI buses are capable of. In this project, AXI-4 lite is used to configure Xilinx IP such as the Videe Timing Controller.

4.2.3 AXI4 Stream

AXI4 Stream (sometimes abbreviated AXIS) is significantly different from the previous two AXI buses in that it is not memory mapped. AXIS is a Master-to-Slave bus standard for controlling the flow of periodic data between modules, typically in a pipeline. It uses two flags: tReady and tValid to apply fore-pressure and back-pressure respectively, ensuring that modules connected together are operating at the same speed. In order to create an AXIS bus, only a clock, a data bus (tData), tReady, tValid, and reset are explicitly needed. Some additional control signals are optional, and most pipelines do not need them. Memory addresses are not needed in AXIS since the topology determines the source and destination of data. Due to this combination of design choices, it is fairly trivial to implement an AXIS interface for any logic that is designed to fit on a processing pipeline. In the design, AXIS is used in several niche cases such as for CORDIC rotations. Most

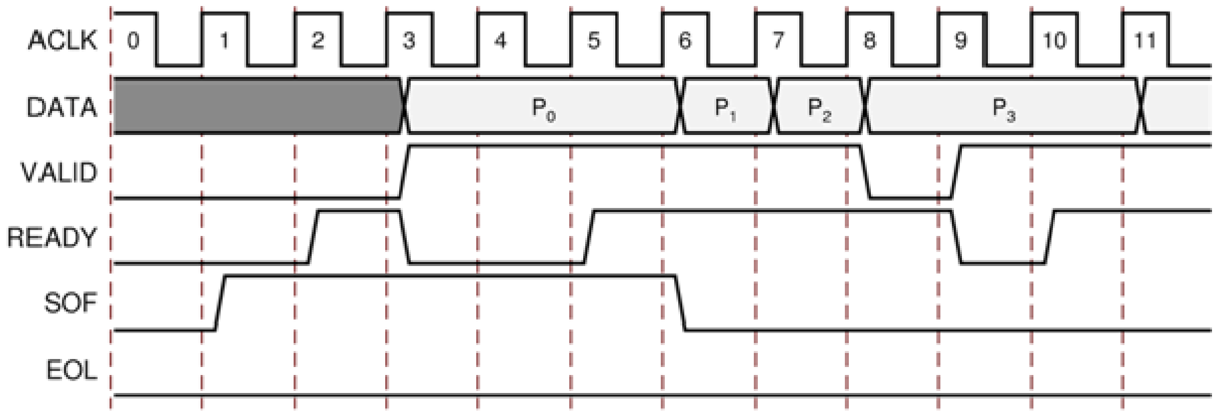


Figure 14: AXIS-V Ready/Valid Handshake [22]

of the design is based on AXI Video Streams.

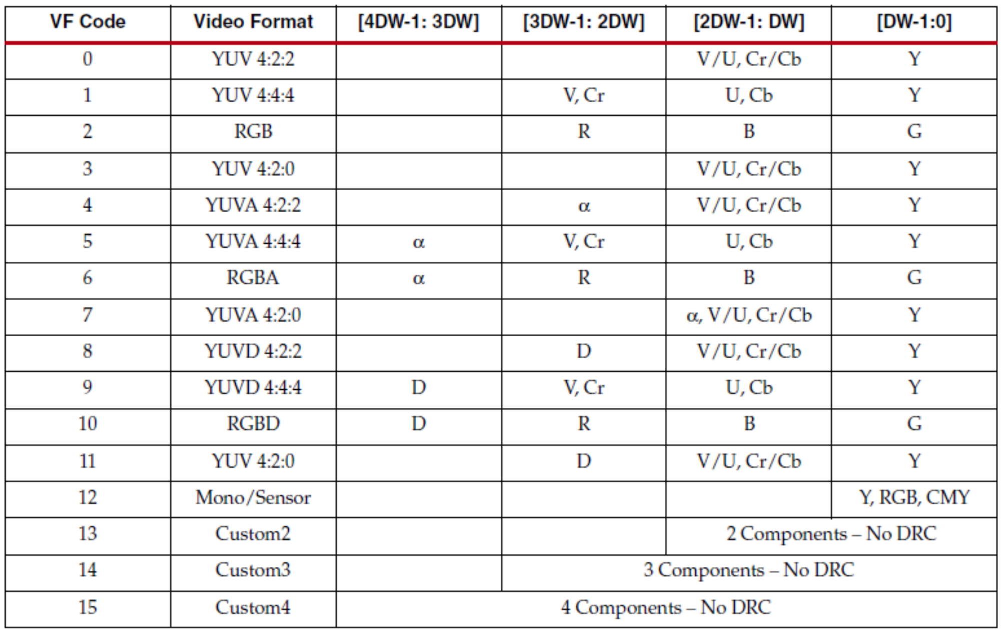
4.2.4 AXI Video Stream

AXI Video Stream (sometimes abbreviated AXIS Video or AXIS-V) is a standard for the payload of an AXI Stream and elaborates on the uses of two extra control signals: tUser and tLast. It specifies a large number of different standards for different video formats and resolutions, up to totally custom combinations. It also defines End of Line (i.e. hsync) and Start of Frame (i.e. vsync) using the tLast and tUser flags respectively. This standard is used throughout the project for the passing of video data between the various IP Cores, modules, and DMA controllers. In contrast to AXI4, it is significantly more import to understand this bus since it is much more feasible to write custom modules based on AXIS-V than full-fledged AXI4. Figure 14 shows the handshake between master and slave. Figure 15 shows some of the encoding this standard specifies, including the byte order for the most common formats.

Compared to the memory mapped standards, it is trivial to implement AXIS-V. In fact, in the design many of the custom designed HDL modules use the AXI4 Video stream bus format to communicate with each other and existing Xilinx IP.

4.2.5 Using AXI to design the system

A combination of AXI bus standards are used to create a complete video system. A diagram of clocking and data flow is shown in Figure 16. There are three independently clocked pipelines in the system. The design is split into two overall halves, separated by DMA. The first half is associated with the clock and data flow of the cameras. Each of the cameras have their own free running clock and logic related to the cameras operate on these clocks. The logic is also operating under positive back pressure (i.e. pushing). The pixels



VF Code	Video Format	[4DW-1: 3DW]	[3DW-1: 2DW]	[2DW-1: DW]	[DW-1:0]
0	YUV 4:2:2			V/U, Cr/Cb	Y
1	YUV 4:4:4		V, Cr	U, Cb	Y
2	RGB		R	B	G
3	YUV 4:2:0			V/U, Cr/Cb	Y
4	YUVA 4:2:2		α	V/U, Cr/Cb	Y
5	YUVA 4:4:4	α	V, Cr	U, Cb	Y
6	RGBA	α	R	B	G
7	YUVA 4:2:0			α, V/U, Cr/Cb	Y
8	YUVD 4:2:2		D	V/U, Cr/Cb	Y
9	YUVD 4:4:4	D	V, Cr	U, Cb	Y
10	RGBD	D	R	B	G
11	YUV 4:2:0		D	V/U, Cr/Cb	Y
12	Mono/Sensor				Y, RGB, CMY
13	Custom2			2 Components – No DRC	
14	Custom3		3 Components – No DRC		
15	Custom4	4 Components – No DRC			

Figure 15: AXIS-V Format Codes and Data Representation [22]

streaming in from the camera cannot be stopped. Logic must always be ready to receive data. The second half of the design is tied to the HDMI clock, derived from the system clock. The logic is also operating under negative fore pressure (i.e. suction). The HDMI output cannot be stopped if pixels are not ready. Logic must almost always be ready to send data - there is a small FIFO that can absorb momentary halts. These timing constraints are the basis for needing VDMA.

Different types of AXI are used together on the design. In the first half of the design, data from the cameras is interpreted from its raw format and packaged into AXIS-V. During this conversion, the data is also re-clocked from its 24MHz clock to the internal stream clock of 148.MHz. The Video in to AXIS contains a FIFO for this purpose. While this does change the actual clock that the data is associated with, the amount, rate, and pressure constraints of the data stay the same. Many of the 148.5MHz clock cycles are not actively transferring data, and this builds in plenty of overhead for ensuring that data is always processed faster than the cameras output it. The AXIS stream is then passed into the DMA controller to be sent to a frame buffer in main memory over AXI4. The VDMA has a connection to main memory through the AXI interconnects, which are arbiters of all DMA traffic and convert between AXI4 and AXI3. In the second half of the design, data is pulled from the DMA controller as AXIS-V, processed to perform any augmented reality, stereo combined, and then output via HDMI. While examining this design it is important to note how all timing constraints converge on the DMA controller. Video is being forced into the frame buffer from the cameras, while being forcibly extracted from the frame buffer by the HDMI output. Each of these has its own clock where data needs to be taken in/output on every cycle, and all surrounding logic

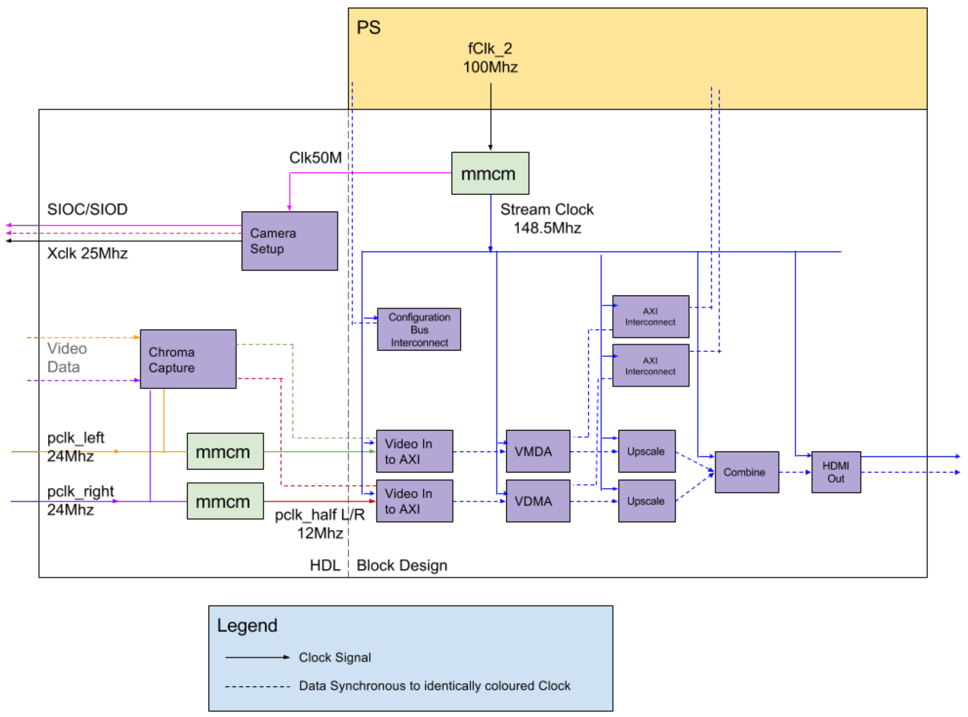


Figure 16: Clock architecture on video pipeline. Video processing omitted

must be able to operate at this speed. The frame buffer created in main memory is able to pick up any error in the synchronization between each of the cameras and the HDMI output.

4.3 Video Direct Memory Access

Video Direct Memory Access (VDMA) is a crucial part of the video pipeline. It relieves the fore pressure and back pressure between the camera inputs and the HDMI output by creating frame buffers.

4.3.1 Frame buffers

All VDMAs used in the design implement a frame buffer in the DDR memory. The design requires two frame buffers, each of which should be at least capable of buffering an entire frame from each of the stereo pipelines. Considering that the cameras are using the YUV4:2:2 format, each pixel is represented by 2 bytes on average (chroma component sharing) and each frame is 640x480 progressive scan. Thus in order to buffer a single frame, 614.4KB of data must be stored. Considering 30 frames per second, this extends to a memory bandwidth of 17.58Mb/s. While these figures are not feasible to implement in BRAM, they are minuscule compared to capabilities of the external memory: 512MB of RAM, accessible from the PL at speeds up to 1.2Gb/s when reading at small burst sizes similar to those used in the design. Thus DMA is the ideal choice

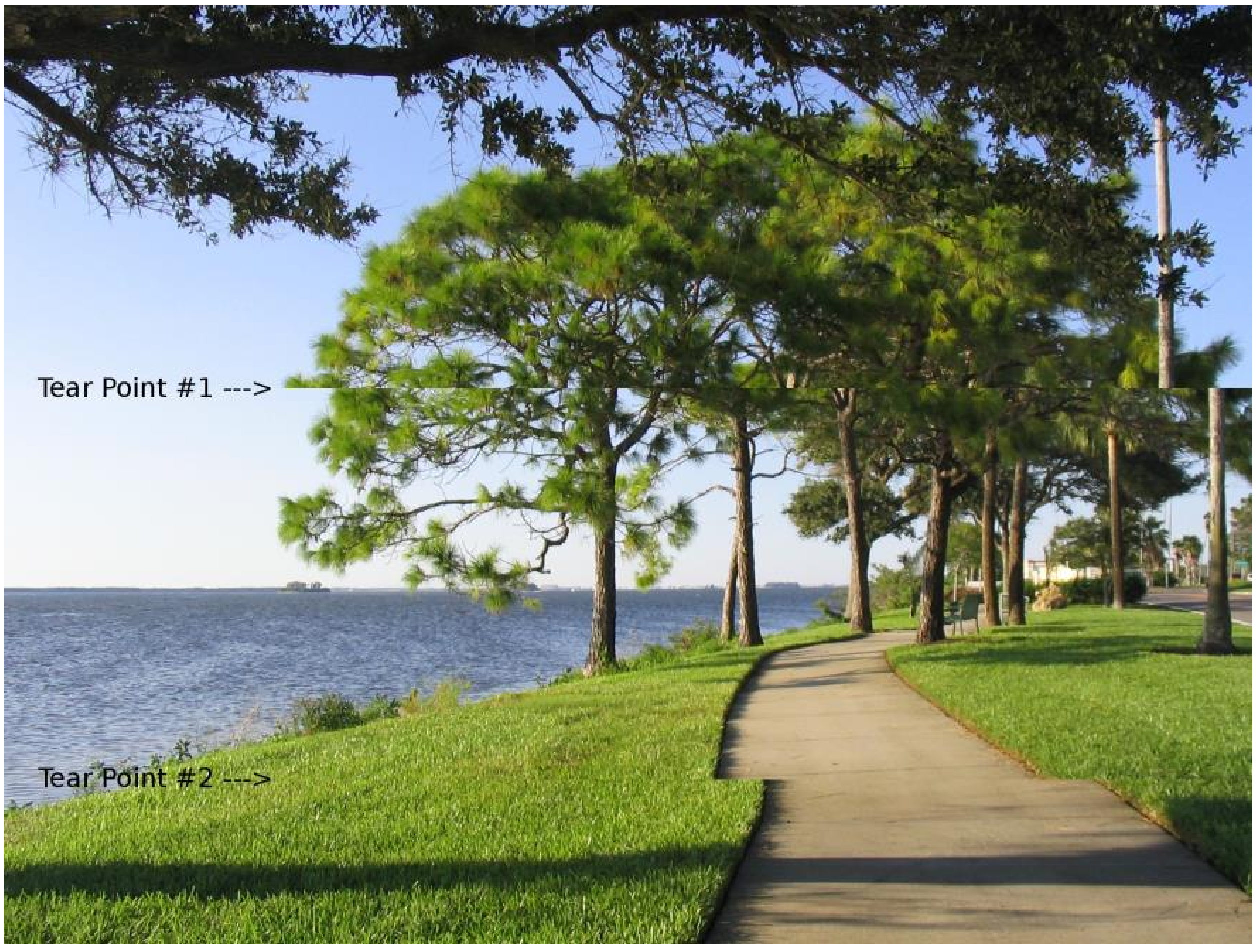


Figure 17: Tearing [23]

for creating frame buffers in the project. While single buffers would be functional, the current DMA setup implements a triple buffer to avoid tearing and to allow frames to duplicate if necessary (i.e. 30fps in, 60fps out). This is closer to the default settings for the VDMA controller than single frame buffering.

In single buffering, the video input would trace along the frame buffer overwriting pixels as they are read in from the camera. The output would also trace along the frame buffer, although it may do so asynchronously to the input. This implies that at some point, the pointers of each trace could cross over each other. If this happens, an artifact called tearing will occur. In the middle of the output frame, a horizontal division will appear over which one frame appears and under which a different frame appears. The frame that appears under the tear depends on which trace is running faster: it could be an earlier or later frame relative to the image above the tear. This is very noticeable if the input is horizontally panning (i.e. the user turns their head side to side). See Figure 17 for a demonstration of tearing when frames are horizontally panned images of each other.

In double buffering, two buffers are used. One is copying in video input pixel by pixel, while the other is reading out video output pixel by pixel. When the video output has completed reading an entire frame and the video input has finished writing its frame, the two buffers switch roles so that the output reads the fresh input and the input begins overwriting the memory previously used by the output. This scheme will not work in this project because it requires the video input to be frame synchronous with the output. The video input must start writing to the frame buffer after the swap occurs and it must finish before the next swap occurs. This makes sense in video rendering applications where the video input can be controlled by vsync, but not in cases where the video input is a freely self-clocking camera. In this case specifically it will not work because the camera input must run at a slower speed than the output frame rate. If a synced input is not possible, tearing once again occurs, except it is written into the frame buffers as a result of the input trace changing buffers mid-frame as opposed to being an artifact of the reading trace in the single buffer example.

In a triple buffer, three frame buffers are used. At a given time, the input and output each operate on one buffer, while the third buffer contains an intermediate frame. By having three frame buffers, the input and output do not need to be synchronized for clean, tear free video. Suppose the output is running at a faster frame rate than the input as is the case in this project. When the input finishes copying in a frame, it switches to writing to the unused buffer. The freshly finished frame is scheduled to be the next frame the output will use. Meanwhile, the output is tracing through its frame buffer. When it gets to the end of the current frame, it may switch frame buffers. This depends on if a new frame is available from the video input yet. If no new frame is available, the same buffer is read again. A frame is duplicated. If a new frame is available, the frame buffer switches to the new frame and outputs this data instead. Suppose that the input was a faster frame rate than the output. The input will always switch buffers at the end of a frame, however the impact of this depends on if the output has begun reading the last frame or is still reading an older frame. The input cannot switch to the buffer that the output is using, so it may overwrite frames the output never read from. This isn't an issue per se. The output will always begin reading a very recent frame from the input, however in extreme cases many input frames may be overwritten by the time the output gets to the end of its frame.

4.3.2 DMA Usage in Design

In order to implement a design meeting the video buffering requirements, the Xilinx AXI Video DMA controller is used. This controller supports an AXIS-V Input, an AXIS-V Output, and an AXI4 Master interface. It also has special I/O for communicating its status to other peripherals. What makes this controller unique from the other Xilinx DMA controllers is this controller fully implements the AXIS-Video standard and is aware of frame buffers, as opposed to plain FIFOs or fully manual DMA transactions implemented by other designs. It maintains its own internal pointers to memory that automatically move between one or more buffered frames as video input and video output send/request data. Again, it is frame aware, not just a FIFO, and it knows how to do multiple frame buffering. To enable this functionality, some configuration is required in hardware and significant configuration is required at run time which is setup in software and sent over AXI4-Lite. These processes are described further in Section 4.3.3.

Two VDMAs are used in the Design, with one VDMA for each camera. The VDMAs are used to create a frame buffer so that differences in the running speed and frame position of the camera and HDMI output can be handled. The VDMA is placed immediately after the Video In to AXI Stream Core and stores YCbCr 4:2:2 640x480p video into a triple buffer. The genlock mode is set to dynamic master/dynamic slave, so that the write half of the VDMA controls two frame buffers and tells the read half of the VDMA which frame buffer to read from. At run-time, software configures the VDMA for operation. The Frame buffer locations are set for each VDMA in main memory, the resolution is configured, and the VDMA is set to run as soon as video input appears.

4.3.3 Hardware configuration of VDMA

The hardware configurations for VDMA are not as intuitive as they may seem but once understood can be leveraged to perform many useful DMA operations. These settings are entered through the IP customization interface in the Vivado development environment. Most of them have an effect on which operations can be performed with the unit after it is synthesized since disabling certain functions will remove the requisite logic from the DMA unit's HDL. The VDMA module must also be properly connected to the video pipeline and to the PS in order to access main memory.

One of the first settings is enabling or disabling the read or write channels. By default, the VDMA unit has both a read and write channel, which makes sense in the context of a frame buffer. Data copied into main memory is read back on demand. In applications where video is rendered by the CPU into a location in main memory, VDMA may be used with only a read channel to copy data out. Alternatively, in an application where the CPU performs some terminal video processing, VDMA may be used with only a write channel to copy video into a location in memory. In the project a frame buffer was needed so all VDMA units have both channels. There is also an option inside the IP configuration panel to select the number of frame buffers, which was set to three for this project. This is cosmetic and merely initializes a register. Almost all implementations of run time software will overwrite this register with a value stored in the configuration program, but there is no harm in having the intended value entered here.

Within each of these channels there are options to set burst size, stream data width, and line buffer depth. There are FIFO buffers built into the VDMA unit since the VDMA unit needs to packetize its transactions with main memory. The VDMA unit must accumulate at least one packet's worth of data in order to start a write transaction. In order to guarantee that video is available to subsequent modules on the pipeline at all times, video must be prefetched several hundred pixels in advance to hide the latency of accessing main memory. The channel settings affect how much data each transaction will contain and how aggressive the VDMA will be about pre-fetching video from main memory. Larger burst sizes and bus widths are more efficient and must be used for a high performance video system (i.e. 4K video) since the overhead associated with transactions can quickly add up. Larger FIFO sizes are also needed on high performance video pipelines since the latency of communicating with main memory can quickly exceed the time in which the FIFO can be emptied by video processes down-pipe. For this project, default settings were used since the video processing does not come near the performance limits of the Zynq system. A burst size of 16 pixels with a 16 bit bus width and a line buffer depth of 4096 were used.

Fsync has different purposes on the read and write channels. On the write channel, fsync is extremely important and not set by default. An fsync signal indicates the separation between frames of video. The VDMA needs an fsync source in order to separate frames of incoming video into their proper frame buffers. Without this, frames will start at an arbitrary point in the frame and will not be properly aligned within the frame buffers. An arbitrary start of frame will be selected based on the assumption that the first data the VDMA receives is from the first row and pixel. In most applications, the s2mm tUser option is the correct setting. This will use the AXIS-V format's built in Start-Of-Frame signal to determine the vertical sync position. On the read channel, fsync serves a completely different purpose. Often, a device outside of the FPGA will be clocking frames. For example, a device combining two DVI or HDMI inputs that does not have its own frame buffers will use an external fsync to synchronize its inputs. The VDMA has a line for an external fsync so that the video coming out of the video pipeline will be synchronized with the external device.

Genlock is a very pivotal setting that affects how the VDMA will sequence through frames. Dynamic master and dynamic slave are used together if it is desired for the VDMA to internally perform a triple buffer operation such as the one described in Section 4.3.1. Dynamic refers to the fact that the read and write channels of the same VDMA unit will use an internal bus to communicate with each other about what to do. Read and write may take either role (master or slave), depending on preference. Master will own all but one of the frame buffers, and slave will be told which remaining frame buffer to use. Dynamic-Master/Dynamic-Slave for write/read respectively is the selection used in this project. Master can be used in a situation where only a read or a write channel is being used, or two separate VDMAs units are being used each with one channel. In slave mode, VDMA will be programmed with a list of possible frame buffer memory addresses and will select a buffer as indexed by a hardware interface. Slave/slave is a mode that was attempted in this project when the CPU was performing video processing and needed explicit control over the frame buffer usage. Unfortunately, this method was unsuccessful and removed when all the video processing was redesigned to work in the programmable logic.

Unaligned transfer is an advanced setting which enables the generation of some additional hardware that utilizes the advanced functions of the AXI3/4 bus to perform memory transaction that are not aligned with 16, 32, or 64 byte lines of memory. Accessing memory starting at bytes aligned to a certain minimum power of two is optimized on almost all computer architectures. Most applications don't need to use specific bytes and a wisely chosen address for the frame buffers and burst size will sidestep this problem. The unaligned transfer function was not used in the project.

The VDMA unit must also be be connected to the PS system. In Vivado, the PS is represented as an additional IP module "Zynq7 Processing System". This module almost has to be used in the block design layout tool in the Vivado development environment, which automates wiring up complicated buses and packaged IP heavy projects. Xilinx Product Guide 082 has more information on this module. In short, the Zynq IP module is the interface used to tell the implementation tool chain to connect nets to features of the PS system, and it also includes many settings that will affect PL and external facing resources of the PS system (i.e. clocks, interrupts, specialty buses). The VDMA's AXI4 Master interfaces need to be connected to the High Performance (HP) AXI3 slave ports on the PS system. Due to the difference in protocol version, an AXI memory interconnect is needed between these devices. An AXI4-Lite bus needs to be connected from the VDMA units to the AXI3 GP Master ports on the Zynq7 module. Again, due to protocol version differences this needs to pass through an AXI memory interconnect. The clock and resets nets need to be setup and routed for these AXI3/4/Lite buses. An 148.5MHz clock derived from the PS's fclk is used for all buses (stream interfaces included). Use of one clock helps synthesis by simplifying timing constraints. These fclk clocks have special resets, and a "processor system reset" IP core is needed to format and distribute two separate resets to all IP on these AXI3/4 buses. One reset is used to reset interconnects, and a separate reset is used to reset the peripherals connected to these interconnects. Most of the wires and IP up to this point are all automatically created and routed with the Vivado Designer Assistance's connection automation tool. Once these connections and hardware are created, the actual video stream (i.e. AXIS-V) inputs and outputs need to be wired up. This is application specific. For testing only, a test pattern generator is wired into the input, and the output is connected to a working HDMI output stage. For design implementation, the input is connected to the cameras, and the output is connected to the first module in the video processing pipeline.

4.3.4 Software configuration of VDMA

Most of the VDMAs settings are configured at run time by software running on a processor. In simpler projects, a Microblaze soft core processor can be used to perform this configuration. In this project, the ARM cores on the Zynq perform these operations. The source of the code used for software configuration was adapted from an Avnet example project, and due to the license agreements and publishing restrictions on this code it is not included with this report. This code can be found on the Avnet website under design name ”FMC-IMAGEON HDMI Video Frame Buffer”. The settings configured are important but generally easy to understand, and they can be leveraged to achieve some fairly complicated operations with the VDMA IP. The following settings are configured in software, some of which may overwrite options set in IP configuration tool in Vivado.

- Input, Output, and storage frame size/resolution
- Stride setting (the size of a pixel in bytes)
- Parking settings (stopping after a certain number of frames)
- Genlock mode and source for each channel
- Number of frame buffers
- The base address in main memory of each of the frame buffers (independent for each channel)
- Start/stop of VDMA
- Checking of error flags and dumping of registers for configuration verification

All of the functions used for this configuration are generated by the Xilinx SDK through a board support package (BSP). BSPs will generate an API of functions and constants for hardware modules that are included in a hardware description file (HDF). A HDF is automatically exported from Vivado when creating an SDK environment associated with a hardware project. The HDF contains information about what Xilinx IPs have been created in a project, how they are connected to the processors, what the base address of their registers are, and what their IP customizations are. This process can be extremely finicky if any of these files/packages are not updated when hardware changes.

4.4 Image Processing

The image processing steps implemented in this project are based on a previously implemented object tracking algorithm [24]. These processes are meant as a proof of concept for an advanced image processing technique that could be implemented in this sort of project. This section provides an in depth look at the techniques and implementation of the video processing steps involved for this project.

4.4.1 Grayscale

All the video processing done in this project uses grayscale images. This is important because in grayscale, objects in an image are more pronounced from each other and easier to separate. Also, many video filters are not applicable to a color image. Luckily, in the image format this project uses, YCrCb, the Y component of the image represents the grayscale image. Therefore, to convert to grayscale, the Y component can be processed separately, whereas in RGB, additional calculations have to be done on the image in order to get the equivalent grayscale image. This reduces the calculations the system needs to perform and also eliminates one source of potential error.

4.4.2 Delta Frame

The delta frame filter finds the absolute differences between two frames. For every pixel i in the output frame, the value is the absolute difference of the value of pixel i in frame 1 and pixel i in frame 2.

$$OutputFrame[i] = |Frame_1[i] - Frame_2[i]| \tag{1}$$

These differences highlight the parts of the image which have changed between the two frames. Figure 18 shows this process in detail. Frame 1 and 2 represent the two frames being processed. The object in frame 2 is slightly shifted right from that shown in frame 1. The combined frame would be the resulting frame if they were simply combined. The resulting figure shows the result of the delta frame calculations. It is important to note that the black figure would not turn white, but it would be separated from the black background as shown.

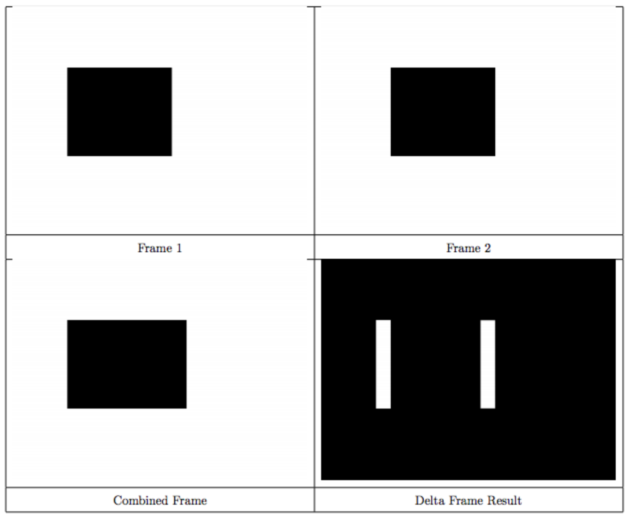


Figure 18: Delta Frame Process

This process is useful for eliminating background from moving objects in a video. Figure 19 shows this process being performed on real video of a hand moving through the frame. In the before picture, there is a lot of background in the image (laptop, wall, etc). However, because they are not changing from frame to frame, they are removed in the delta frame process because they are subtracted from themself, and what is left is only the moving hand (more can be seen at `https://youtu.be/2wbC4cexFb0`).

This processing is most useful when performed on two consecutive frames. By doing this, the output will highlight the objects in the frames which have moved between the two frames. This is highly dependent on

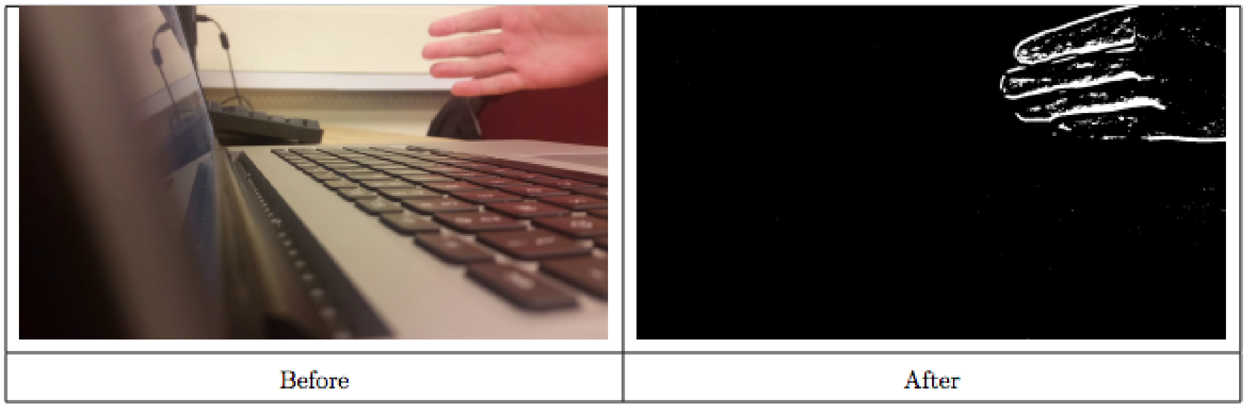


Figure 19: Delta Frame Calculations on a Video of a Moving Hand

the frame rate at which the video being processed. At low frame rates, there could be large changes between two frames, which would result in a convoluted output frame. However, at high frame rate, smaller changes can be detected, which means a more meaningful result on fast moving objects.

4.4.3 Thresholding

Thresholding is the process of setting all values below a cutoff value to 0. This reduces the amount of noise in the image that can be caused by small differences in the frames before the delta frame calculations. These differences can be caused by lighting or resolution. If the cutoff value for the threshold is too low, some of the noise that should be getting removed stays and can change the processing later on in the pipeline. If the cutoff value is too high, some of the values that are relevant to data processing can be removed, and therefore the desired video might not output. This means choosing a threshold value is an important step. There are processes for dynamically choosing the threshold (detailed in [24]), however, these add additional logic and unreliability to the system for a small improvement in desired output. In addition to removing all values below a threshold, is order to assist in processing later on in the pipeline, all values which are not removed are set to the max value, 255.

4.4.4 Median Filter

A median filter is another way to remove noise from an image. Median filter works by assigning each pixel to the median value of all its adjacent pixels. This processing is similar to averaging the pixels of the image, however is less sensitive to large differences in values. This is important because previous to this step, thresholding was performed on the image. This means there will only be values of the lowest value or the max value, so there will be many of values of 0 and 255, but none in between (for more information see Section 4.4.9).

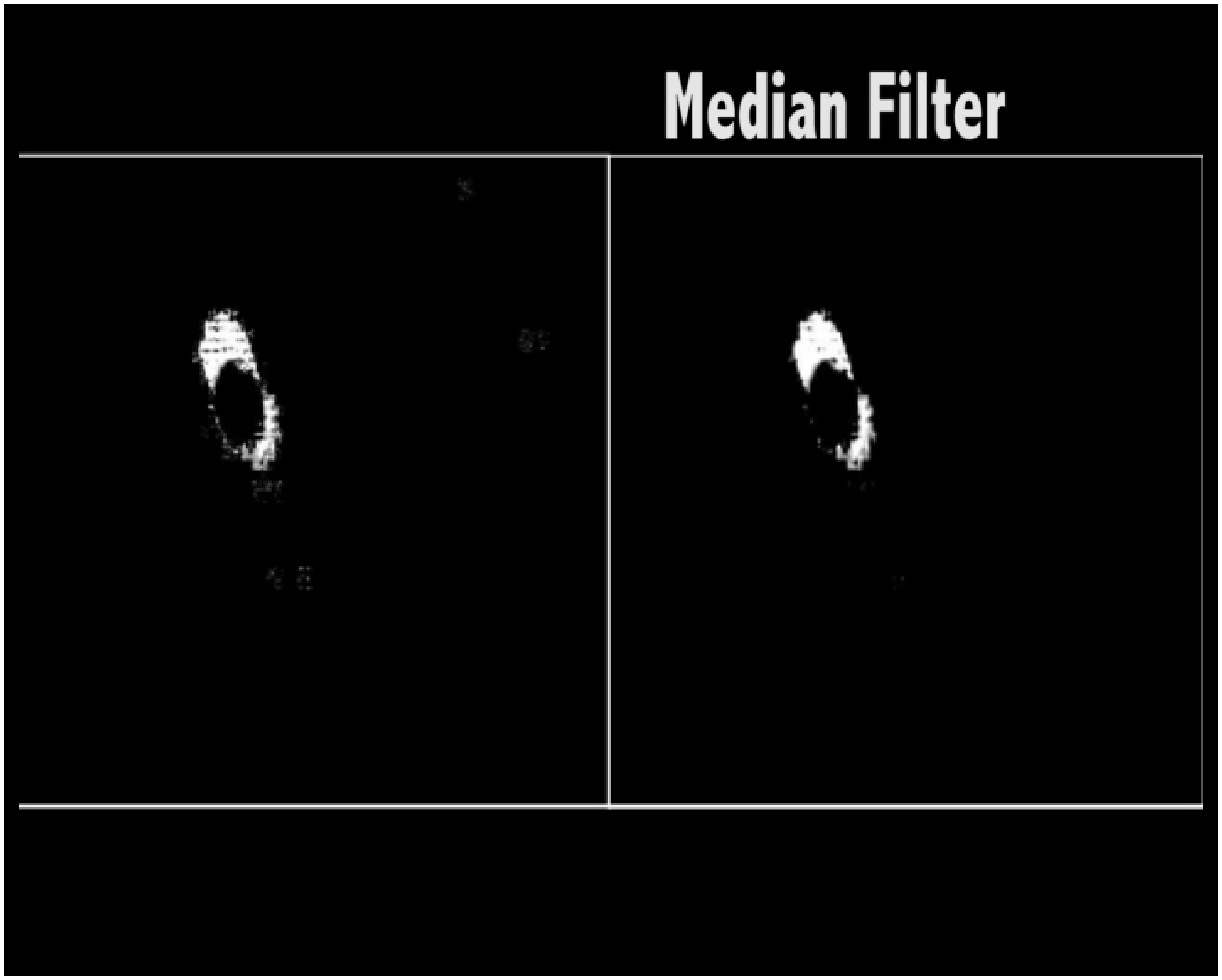


Figure 20: Median Filter (Before on the Left, After on the Right)

$$G_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \quad G_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

Figure 21: Sobel Masks[25]

Figure 20 shows the results of the median filter applied to the result of delta frame and thesholding. As can be seen in the figure, the image on the left has more noise that was not caught by the thresholding, that should be removed before processing is continued. The median filter also can remove negative noise as can be seen in the white object in the image. The median filter inserted white into the object, where noise caused the delta frame to remove data.

4.4.5 Sobel Edge Detection

Sobel Edge Detection is an algorithm for highlighting edges in an image. It works by applying a mask to each pixel in the image (shown in Figure 21). That pixel is then assigned the magnitude of the result of the mask (as shown in Equation 2). Most systems simplify this equations by using an approximation shown in Equation 3.

$$|G| = \sqrt{G_x^2 + G_y^2} \tag{2}$$

$$|G| \approx |G_x| + |G_y| \tag{3}$$

The result of the Sobel Edge Detection Algorithm is an image where all of the edges of objects have a high value (close to white), and everything else has a low values (close to black). The result can then be run through a threshold to reduce leftover noise in the image. A result of the Sobel edge detection can be seen in Figure 22. When run on the result of the delta frame, the highlighted object which has moved will be clearly outlined in the image.

4.4.6 Implementation

The implementation of the delta frame and thresholding can easily be done in hardware. Using AXIS-V, the delta frame implementation is relatively straight forward. The same holds for the thresholding as well. Xilinx also provides implementations of these two functions. However, because they are based on the OpenCV library, they are overly complex for the purposes the application of this project.

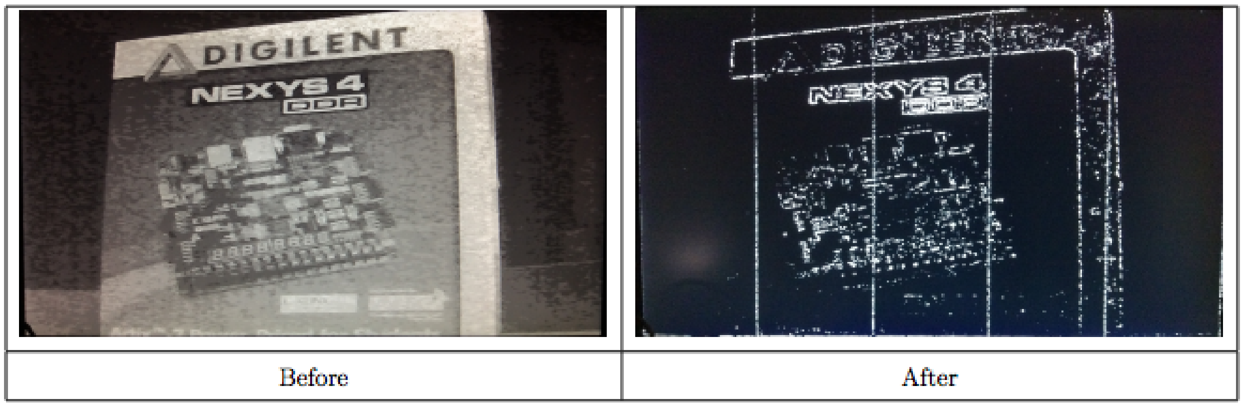


Figure 22: Before and After Sobel Edge Detection

4.4.7 Thresholding Filter Implementation

For the thresholding filter, a simple HLS function was implemented that takes in one AXI stream and outputs 255 or 0 for the data of the output stream depending on the value of the input data. This was implemented using a basic conditional assignment operator.

4.4.8 Delta Filter Implementation

For the implementation of the Delta Filter, two versions were created; one using custom AXI interface code, and one utilizing the HLS environment. This was due to various issues we discovered using both versions. The first implementation utilized HLS. The only computation that is performed is that the output data becomes the absolute difference of the two input data. The rest of the AXI interface data is pulled from one of the input streams.

For the second implementation two operations are required. The math portion takes the absolute different between the values of two pixels, which is rather simple. The more complicated part involves synchronizing two incoming video streams: one video stream for the current video and one for the delayed video. Based of the tUser (start-of-frame) signal, a state machine queues up the incoming streams so that each is lined up with the beginning of the frame. Once both inputs are lined up, then the streams are simultaneously consumed, and the delta math is applied to each pixel.

4.4.9 Median Filter Implementation

The median filter is a slightly more complex module. Due to the complicated nature of the state for this module, significant logic is devoted to controlling the operations of the median filter. The first control function is a cold-start counter. This counts out 2 rows and 3 pixels of input data, the amount of data needed to fill the buffer in preparation for the first pixel. The second control function is a pair of x-y counters. These

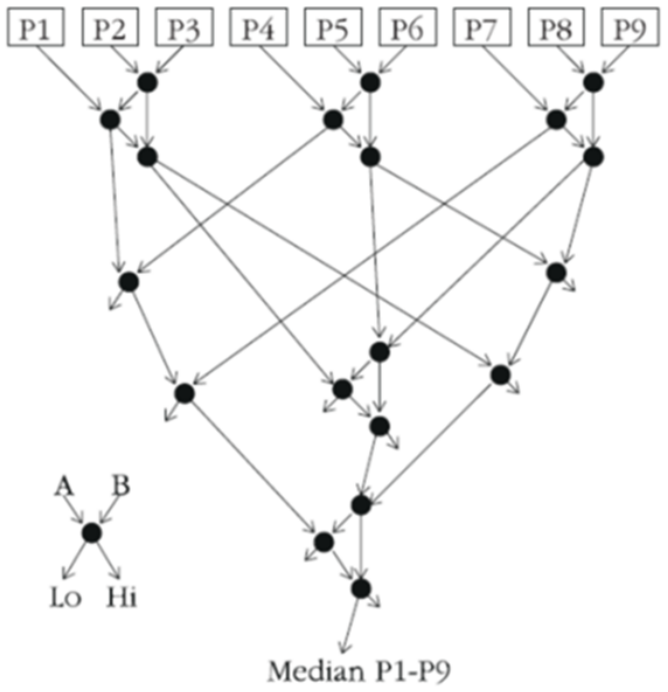


Figure 23: Median Filter Decision Tree [26]

will be used later on to detect and trap the edge cases. Lastly, control logic is threaded throughout the design to stop the pipeline if pressure is applied to this module. Should another element up or down pipe of the median filter need to stop the flow of data, the median filter will need to stop immediately and not write to any of its registers. The median filter also has to apply pressure of its own when performing a cold-start. The module will not let down pipe modules consume data until there is actually data being output by the median filter. This is done by setting tValid on the output AXI Stream.

In order to implement a median filter, a line buffer is needed. The line buffer must span two rows and three pixels, such that the top left, bottom right, and the other in between values of the median filter mask can receive data. The line buffer is a very simple shift register that shifts 8 bits at a time, loading the most significant byte (MSB) with new data and shifting old data towards the least significant byte (LSB).

In order to implement a real world functional median filter, the filter must pad data into the edges of the image as the mask encounters them. 8 trap cases are used, each corresponding to a different side (top, bottom, left, right) and corner (i.e. top-right). A special solution is used to populate the median filters 3x3 mask in each of these cases. This entire process is combinatorial.

The median calculation is performed by a combinatorial decision tree. The tree is shown in Figure 23. Each black dot is a comparator. A combinatorial solution was deemed acceptable in the target performance window (MHz). If timing is difficult to meet, registers can be inserted to add more pipeline steps to the median filter design.

4.5 Upscaling, Rotation and Combining Video

In the system, video input comes in from two cameras at 640x480 each and must be outputted as one video stream at 1080x1920. For this reason, video rotation, scaling, and combining are added to the design. The solutions to these issues are discussed below.

IP was implemented to solve these problems using Vivado High Level Synthesis (HLS). HLS is a tool that allows the programmer to write C or C++ code which is then synthesized into programmable logic. It allows for rapid development because it abstracts away low level boiler plate code that must be written for each module of hardware description language (HDL). For this case in particular, it allowed for abstraction of the logic of interfacing with AXI video streams (AXIS-V). This abstraction allowed for rapid prototyping and easy video manipulation, without requiring an interface with AXIS-V to be written.

4.5.1 Video Rotation

In order to compensate for the DK2's portrait mode screen, video must be rotated 90°to show up correctly to the user. On a typical system using the DK2, this step is performed by the graphics card in a computer. On this system, a graphics card is not used, so it would have to be done in the CPU. Such an operation is expensive since random memory accesses must be used to access one pixel from each row individually. For this reason, physical rotation of the camera is used instead. A diagram of this setup can be seen in Figure 30. This simple solution ensures that both the cameras and DK2 are scanning in the same direction. It also has the added advantage of aligning the longer resolution of the camera with the longer resolution of one eye in the DK2. Aligning them increases video quality as well as making scaling easier as discussed in section 4.5.2.

4.5.2 Video Scaler

Video is upscaled from 640x480 to 1080x960 using a variant of the nearest neighbor algorithm. This algorithm is relatively simple, each pixel in the larger image is copied from the nearest pixel in the original image [27]. A visual representation of this algorithm can be seen in Figure 25. The figure shows how the original image pixels are spaced out evenly in the new image. To complete the image, each blank pixel is selected by finding the nearest original pixel.

Mathematically, the pixel to copy is compared by computing the ratio Old Width/New Width. That ratio is then multiplied by the index of the new pixel, resulting in the index of the pixel to copy. In the case of this project, the algorithm was greatly simplified because the vertical upscaling ratio worked out evenly to 0.5. This meant each line could be repeated twice, rather than doing the math to figure out which line to copy from. It also simplifies the buffering for the program as it only needs to buffer one complete line to

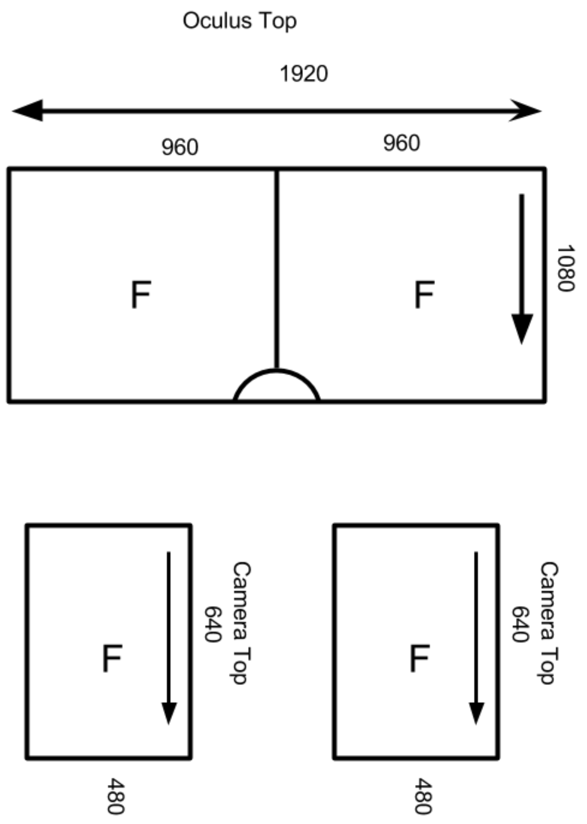


Figure 24: Diagram of DK2 and camera orientation and scan direction

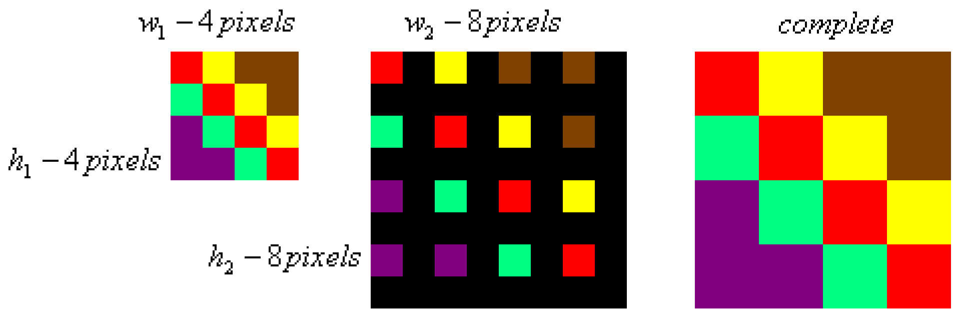


Figure 25: Visual representation of the nearest neighbor algorithm [27]

duplicate rather than a large section of the original image.

One important factor to take into account when up scaling YUV video, such as used in this project, is that the video format used shares chrominance values between neighboring pixels. This meant that rather than duplicating individual pixels, pixels had to be duplicated in side by side pairs. Another factor that had to be taken into account when using AXIS-V was each data value can only be read once. For this reason, every pixel needs to be buffered in memory until the program moves on to duplicating a different pixel.

4.5.3 Combining Video Streams

The last step in the video manipulation pipeline is to combine the two camera streams into one stream to be sent to the DK2. The module takes in two 1080x960 streams and outputs one 1080x1920 stream. Since the two streams need to be placed one on top of the other when sent to the DK2, the algorithm is rather simple. One frame of stream A will be read and sent to output, and then one frame from stream B will be read and output. A diagram of the result of this process can be seen in Figure 26. The code used to implement these modules can be found in Appendix G.

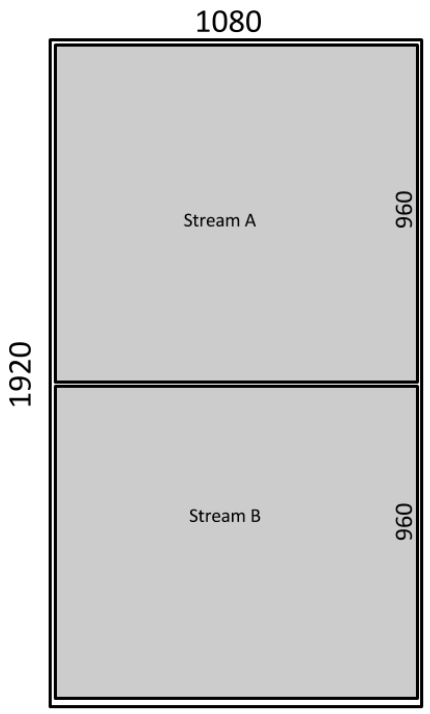


Figure 26: Diagram of how two streams are combined into 1

4.6 Barrel Projection

The barrel projection module is used to correct for the distortion applied to the image by the lenses in the DK2. The lenses create a full field of view for the user, but also have a side effect of adding a pincushion distortion effect which is counteracted by the barrel projection effect as seen in Figure 27. The barrel projection module has three main components as seen in Figure 28, the memory module (blue), the math module (orange), and CORDIC (green). The memory module is responsible for buffering incoming video, as well as retrieving pixels from memory and outputting them. The math module maps coordinates in the outgoing picture to coordinates in the incoming picture. CORDIC is a Xilinx IP which is used to convert Cartesian coordinates to polar, and polar coordinates back to Cartesian. These blocks are described in more detail below.

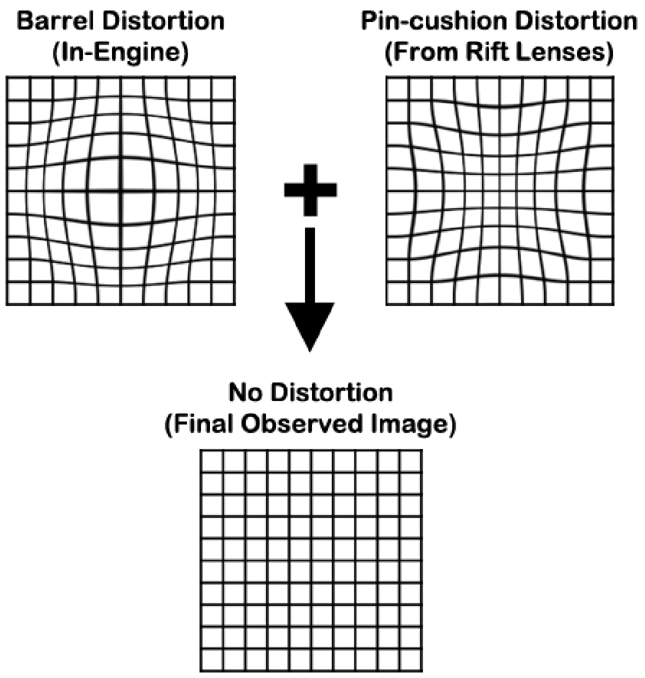


Figure 27: How barrel distortion and pincushion distortion counteract to form a final image [28]

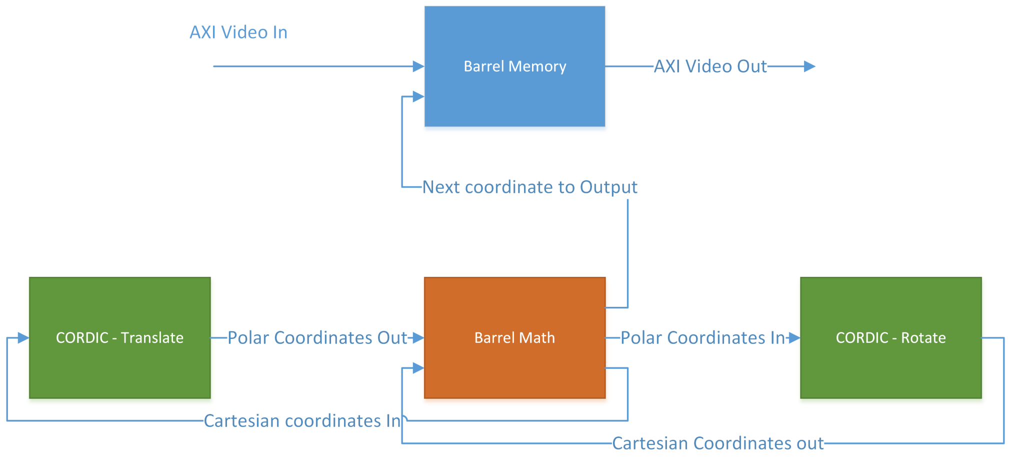


Figure 28: Overall block diagram of barrel projection module

4.6.1 Barrel Projection Memory

The barrel projection memory module handles storing, copying, and reordering pixels. In order to do this, a BRAM based 62.57 row line buffer is used (67584 pixels, exactly 132 kilobytes). 60 rows is the maximum look back distance that is required to perform barrel projection. Some slack is built in at no cost, since BRAM is only divisible in 2 kilobyte chunks. Pixels are read in through an AXIS-V interface and stored in a circular buffer. The location of the head is tracked, both in terms of address and the corresponding coordinates in terms of the 1080*960 result image. The math module passes the coordinates of source pixels in the raster order of the output image. These coordinates are converted into a look back distance based on the current head of the circular buffer. Then, the BRAM is addressed to retrieve the proper pixel. The retrieval process is performed with a two clock cycle latency: One cycle of latency to calculate the address and one cycle of latency in the BRAM itself.

New data is typically read in on every clock cycle; however, due to forward or backward pressure on the video pipeline, this will not always be the case. The module will attempt to keep the buffer filled with between 60 and 62.57 lines. If the buffer runs too low due to hold ups earlier on the pipeline, math will be halted and the output interface will signal invalid data. If the BRAM becomes completely full, the input interface will signal that the module is not ready to receive new data. If the next module on the pipe is not ready to receive data, math is halted and the output is latched but the BRAM buffer is allowed to continue filling. Through all of these interactions, the barrel projection unit is able to adapt to all coerced exceptions.

4.6.2 Barrel Projection Math

The barrel projection math module maps coordinates in the output image to the coordinate in the input image. It is done in this direction rather then mapping from input to output in order to ensure that every coordinate in the output is mapped to a value. It performs these actions in three main steps, generating a Cartesian coordinate and sending it to CORDIC to map to polar, adjusting the radius through a catmull rom spline function, and sending the new radius to another CORDIC in order to convert back to Cartesian. The whole block is pipelined in order to produce one pixel of output every clock cycle after an initial start up delay. The math and CORDIC functions use fixed point numbers in order to speed up processing. This means that the numbers have the decimal place in a fixed spot. These are typically expressed in 'Q' format where the number after Q refers to the number of fractional bits. For example, a 16 bit Q4 number has 12 integer bits and 4 fractional bits.

The math module tracks the current output position using a simple counter that is incremented on each clock cycle. In order to properly map to polar coordinate space, the origin must be in the center of the image. For this reason, x starts at -540 and increases to 539, and y starts at 480 and decreases to -479. The x and y coordinates are then converted to Q3 fixed point numbers by left shifting each by three. They are then concatenated together in the format y, x and passed to the CORDIC translate function to be converted to polar coordinates.

In the next part of the math module, the phase and radius of the current coordinate is received from CORDIC. The phase is passed through the pipeline unmodified, while the radius is adjusted based on a catmull rom spline curve. The algorithm for adjusting the radius was developed by reverse engineering the Oculus sdk as described in Section 5.5.1. First, the radius is squared and multiplied by ten and then divided by the maximum radius squared. This value is then floored, which is used to determine a precalculated coefficient. The fractional part of the scaled radius squared is also saved and used in later calculations. All of these values are then combined to calculate the final scaling factor of the radius. This value is multiplied with the radius to determine the new radius. This radius and saved phase are passed to CORDIC rotate in order to be translated back into Cartesian coordinates.

In the last part of the barrel projection, the Cartesian coordinates are retrieved from the CORDIC rotate function and then converted to be sent to the memory module. The memory module operates with (0, 0) as the first pixel of the frame, so the coordinates must be converted. This in done by adding 540 to x, and multiplying y by -1 and then adding 480. In addition the fractional part of the numbers is truncated before outputting the value to the memory module so the correct pixel can be retrieved and output.

4.6.3 CORDIC

The COrdinate Rotation DIgital Computer, or CORDIC algorithm was originally developed in 1956 by Jack Volder. It is designed as an efficient way to perform trigometric calculations [29]. This project uses an implementation of CORDIC offered by Xilinx as an IP core. As mentioned above, two instances of CORDIC are used in the barrel projection design. One is set up in translate mode where it translates coordinates from Cartesian to polar. The other is set up in rotation mode where it translates polar coordinates to Cartesian.

The CORDIC modules interface with the rest of the project using an AXI-stream protocol with the tData and tValid ports used by default (see subsubsection 4.2.3 for more information on AXI streams). This project uses the ready signal as well so that CORDIC will block if needed later in the pipeline. The translate instance takes one input, the Cartesian coordinates with y and x concatenated. The coordinates are both expressed as 16 bit fixed point numbers. The decimal point can be placed in any point in these inputs, and the module will output the radius in the same format. In this case, Q4 numbers are used. The translate CORDIC outputs the phase and radius concatenated together. Both are 16 bit fixed point numbers with the radius in the Q4 format (as passed in) and the phase in Q13 format. The CORDIC rotate function operates in a similar way, except that the phase and radius are input separately. The radius is input as a 16 bit fixed point number concatenated with a 16 bit imaginary portion (which is set to 0 in this project). The phase is input separately as a 16 bit Q13 fixed point number. The output from rotate is in the same format as the input to translate with x and y components concatenated together.

4.7 Video output

The main purpose of the video output section is to convert the video from the AXIS-V format used on the ZedBoard to the HDMI format required by the DK2. The overall design of this module can be seen in Figure 29. The module uses three IP blocks on the FPGA (green), a set of C configuration code on the processor (dark blue), and a piece of hardware on the ZedBoard (light blue). The first step of the module is the AXI stream to video out module which combines the AXI stream input with timing signals from the video timing controller (VTC). It then passes the video to the Zed HDMI out module, a piece of AvNet IP which synchronizes the video with the incoming clock signal. Last, it is passed to the ADV7511, which outputs the video to the DK2 over HDMI. The VTC and ADV7511 also receive configuration from bare metal C code running on the onboard ARM CPU.

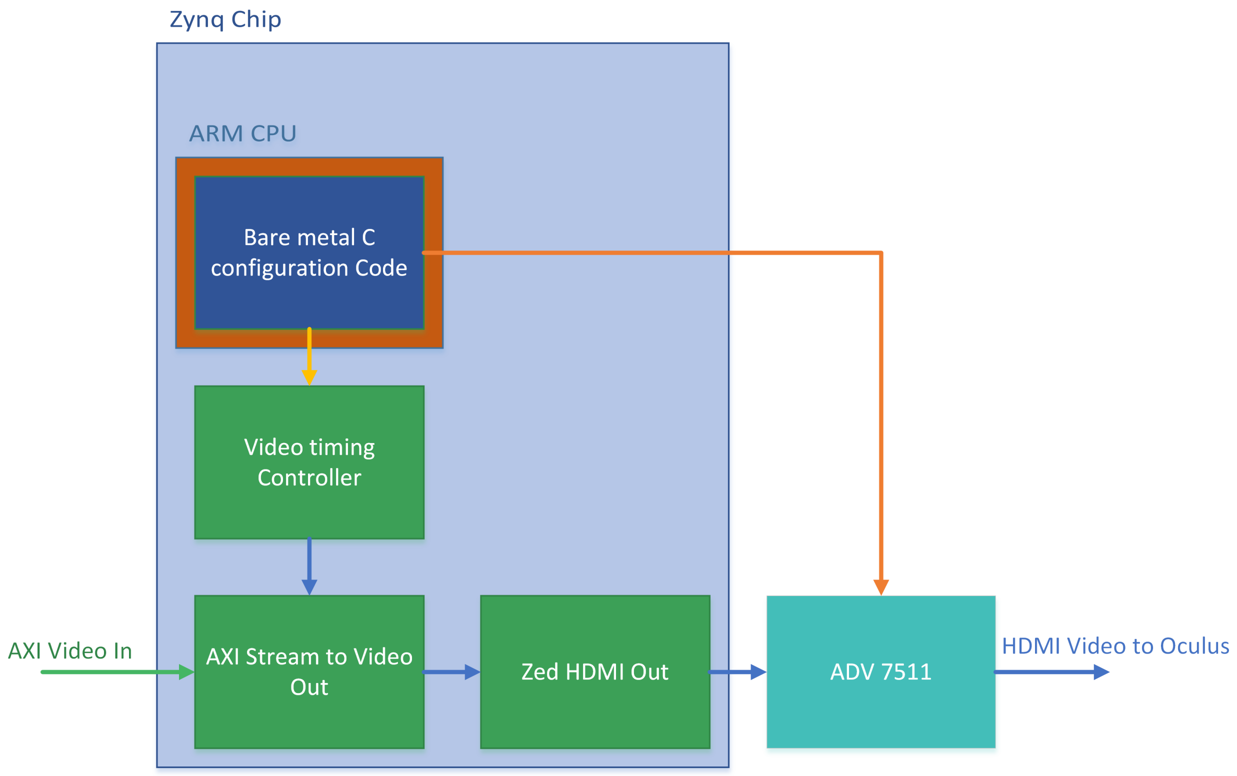


Figure 29: System diagram of the video output Module

4.7.1 Video Timing Generator

The Video timing generator is a piece of Xilinx IP used to create timing signals for the video. This IP block is necessary because AXI streams do not carry standard timing data (such as vsync and hsync), so it must be created before it can be combined with the AXI stream to form video output. In this project, the VTC is configured to be setup by the CPU at run time. This means that code running on the CPU connects to the VTC over an AXI-lite interface and programs several registers. This allows for easy reconfiguration of the VTC to different video resolutions without needing to re synthesize the code.

4.7.2 AXI Stream to video Out

The AXI Stream to video out module is another piece of Xilinx IP, with the purpose of combining AXI stream video with video timings in order to form a video out signal. In this project, it is setup in master mode, which means that the VTC is the master. This means that the AXI stream to video out IP will wait until the timing matches the video input by putting back pressure on the video and allowing the timings to run. This is acceptable because it is after the video has been saved in the frame buffer, so at worst case it will stop video from being read out of memory until it is ready for input.

4.7.3 ADV 7511

The ADV7511 is a chip manufactured by Analog Devices that converts video signals into HDMI output. Before using the chip, on board registers must be set in order to configure various video input and output signals. It is programmed from code on the CPU over an I^2C interface. In addition, when a new device is plugged in, a signal must be sent to power on the chip, and cause it to start transmitting. This can be done by listening for the hot plug detect signal which indicates that an output source has been connected.

The ADV7511 supports a large variety of video input formats, however due to limitations of the ZedBoard, not all the required pins are connected. This means that on the ZedBoard video must be input in the 16 bit YCbCr standard. In this format, each pixel is 16 bits, with the bottom 8 being the Y or luminescence value. Each pixel will then have a 8 bit chrominance value, either Cb or Cr. The chrominance value is shared between neighboring pixels, allowing each pixel to alternate between whether it carries a Cb or Cr value. Whether the first pixel carries a Cb or Cr value is dictated by the standard, and it alternates with every subsequent pixel. After receiving video in this format, the ADV7511 will convert it to the RGB color space before outputting to the Oculus.

4.8 Oculus Rift DK2 Interface

Due to the DK2's construction using the screen from a phone, interfacing with is is differant than interfacing with a conventional HDMI monitor. This section discusses those difficulties and how they were solved.

4.8.1 Development Kit 2 Hardware

The Oculus Rift Development Kit 2 (DK2) is a virtual reality device that is a precursor to the Oculus Rift Consumer Version 1 released in quarter one of 2016. The DK2 features a high resolution OLED display as well as several sensors for tracking the device's position and movement. It has a 1080p display which is divided to 960 by 1080 per eye. It receives video over an HDMI interface. It sends data and receives power over USB. This project is only using the USB for power, however it could also use this interface to receive data from onboard sensors. Due to the fact that the DK2 is a prototype, it uses an off the shelf screen from a Samsung Galaxy Note 3 (see Figure 30) [12]. This adds complexity as the screen is designed to operate in portrait mode (1080 x 1920). While there is a simple fix for this on a PC with a graphics card, rotating video is much more difficult on an embedded system due to the expense of doing random memory accesses. For more information on the solution to this problem, see Section 4.5.1.

Figure 30: The DK2 screen removed from the device [12]

4.8.2 Video Timings

Due to the DK2's rotated display, the default video timings for 1080p cannot be used. In order to determine the correct timings for the DK2, the Enhanced Display Identification (EDID) data for the system was used. The EDID data is a packet which the DK2 sends over HDMI in order to tell the video source what timing it requires. The timing for the DK2 is shown in Figure 31.

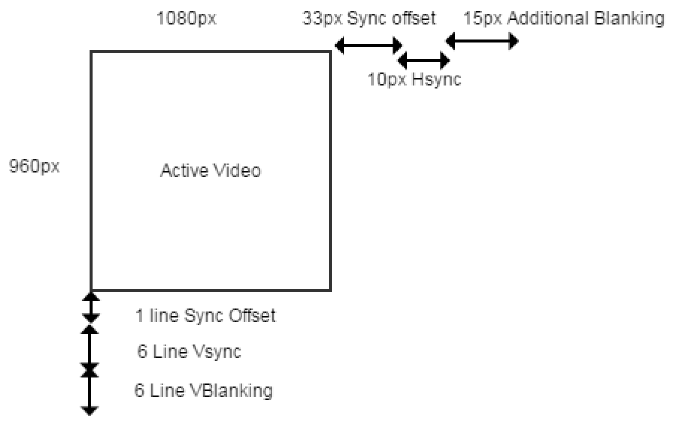


Figure 31: The Oculus Video Timings

There are four main video timing constraints in each of the horizontal and vertical directions. The first, active size, refers to the pixels/lines which video is actually being shown on. The rest of the lines are blank, and may contain syncing data. Before a sync is sent, there is an offset period. The sync will then be sent for a certain number of pixels or lines, and then there will be an additional blanking period before the end of the line or frame.

5 Testing and Results

This section describes the process the team went through to develop and test the system. Each subsection discusses the various approaches that were tried before the final approach was chosen. It also describes how each component was tested and the result of that testing.

5.1 Cameras and Interface

As discussed in Section 4.1 the initial implementation for capture from the camera input was based on [21]. However, when implemented into the final system, a custom interface for capture video data from the camera needed to be implemented due to the fact that the base code used the RGB color format, where as the rest of the system used the YCrCb format (implementation details are described in Section 4.1). However, the code for setting registers stayed the same because there was no need to change it.

One issue that was discovered, which was never solved, was capturing the color information from the camera. The output of the information coming from the camera onto a display showed odd color information, normally shades of green (see Figure 32). It was suspected that this problem was caused by color space

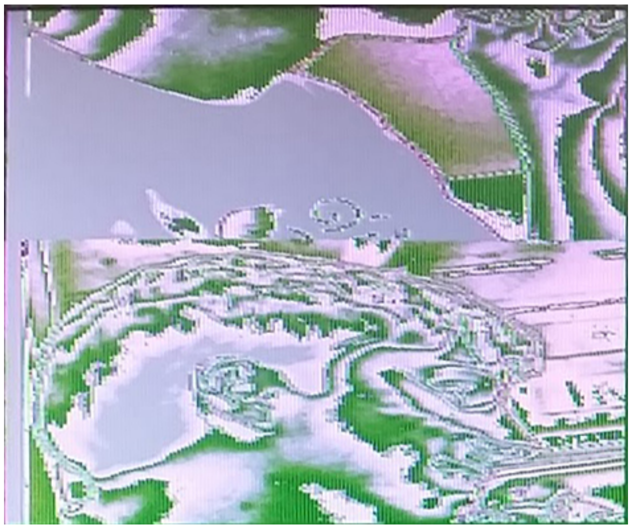


Figure 32: Result of Color Space Issues from the Camera and out Video Output

discrepancies between the camera and the video pipeline. The team confirmed that the video pipeline was using the standard YCbCr color space by using the Xilinx Test Pattern Generator, However, the camera documentation simply states the video output format is YUV [20]. The problem arises when looking into the relationship between YUV and YCbCr. YUV is a video color format that specifies how the color is transmitted. Namely, the information about the video signal is separated between the gray-scale information (the Y component) and the color chromanance values (U and V). However, it is actually a analog video format

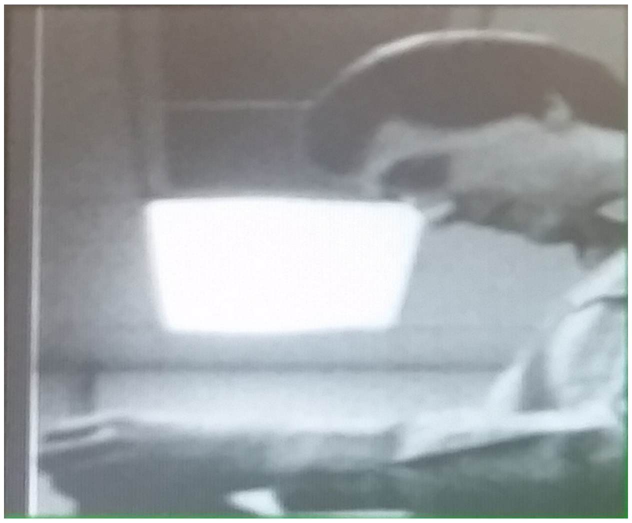


Figure 33: Gray-scale Result Image from Cameras

and therefore does not translate to digital formats as used in this project [30]. YCbCr on the other hand is a digital representation of the YUV format that maps the color space of YUV to digital bit representation. In practice, when referring to YUV format for digital signal, normally the YCbCr format is being referred to. However, this would imply that the video format coming out of the camera is compatible with the video pipeline already existing in the project, which is not what occurred in practice. In order to determine what the problem was, various experiments were done involving changing the values of the video output format registers on the cameras and adding values to the signal artificially in attempts to understand what that would do to the video output. The problem was never solved and therefore this problem was instead worked around. One option would have been to move to the RGB format from the cameras which was successfully implemented in a side project, however this was not an option because of the limitations of the ZedBoard requiring the use of the YCbCr format. As a solution, only the gray-scale information from the camera was used (see Figure 33). Although this would not work for commercial applications, it was fine for this project because none of the video processing steps implemented in this project rely on the color information of the video.

5.2 Image Processing

The image processing part of this project uses straight forward algorithms. However because they are being applied to streaming video signals they can be difficult to develop. Where as the implementation of the algorithms themselves did not require much effort, finding, developing, and testing a method of integrating them into the video pipeline is where much of the effort for the video processing pipeline went. This section describes the effort in testing each of the video filters described in Section 4.4.

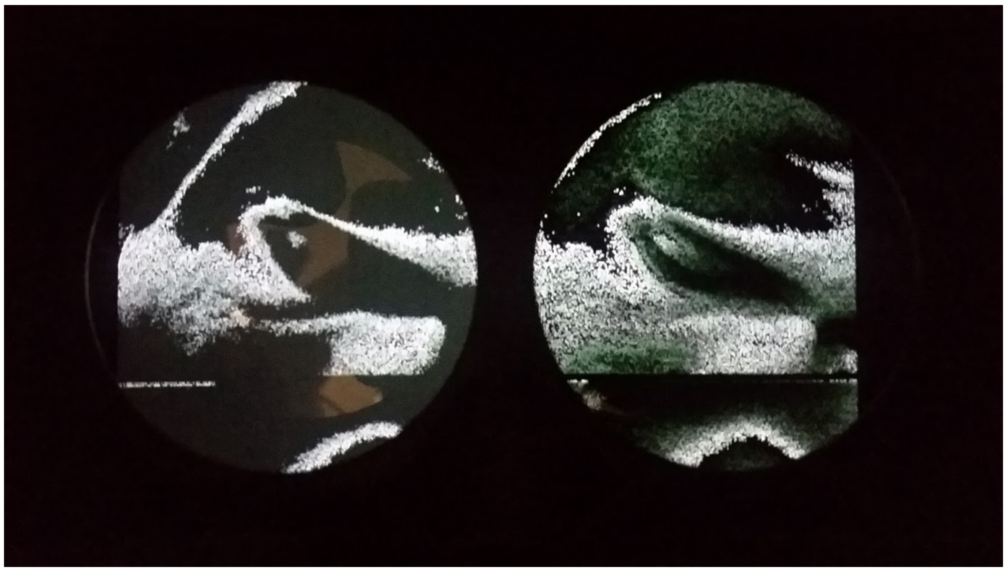


Figure 34: Thresholding Result, Original on the Right, Threshold on the Left

5.2.1 Delta Filter

The initial implementation of the delta frame filter was done in HLS in order to utilize the abstraction of the AXIS-V interface it provides as seen in Appendix A. However when added to the video pipeline, the delta frame failed to output video and therefore broke the video pipeline. The problems were most likely due to the synchronization between the two input frames to the filter. However, due to the abstraction of HLS, it can be very difficult to debug the interfaces created by HLS. Therefore, various alterations to the HLS code was created to try and fix the problem. After many failed attempts, other implementations were investigated.

The implementation the team decided on was a custom implementation built from the ground up in Verilog. As the team progressed through the project, the AXIS-V interface became much more understood and implementing them from scratch seemed feasible. The custom designed filters were quickly constructed to bypass the limitations of the HLS implementation, which in the case of the Delta filter, prevented two streams from being synchronized. This was an easily implemented state machine in Verilog. Unfortunately, when it came to testing on the hardware, this version failed to output video. Therefore in the end implementation the original version created in HLS was used.

5.2.2 Thresholding

The thresholding filter was successfully implemented in HLS and then tested on the video pipeline (a code listing is located in Appendix B). The results from a test done using a Xilinx test pattern generator can be seen in Figure 34.

One important note about the implementation of the Thresholding filter in HLS is that the data in the AXI Stream contains the Y and Chromanance component in one integer. In this implementation of this project the data is a 16 bit number, with the high 8 bits being the Chromanance component, which needs to be ignored, and the lower 8 bits are the Y component which is the actual data that needs to be thresholded.

5.2.3 Median Filter

Initially, the team planned to implement the median filter on the PS side of the Zynq chip. This would have had the advantage of being able to directly access the video stored from the VDMA in memory. However, in order to implement this into the video pipeline, VDMA had to be controlled from the CPU. In theory this is possible using the external genlocks, however, when it came to implementation this was never accomplished for unknown reasons (see Section 4.3.3 for more details)[1].

The median filter was also redesigned from the ground up. After some closer analysis, the data buffering requirements for a median filter are not as severe as anticipated. Only two rows and three pixels need to be buffered in order to look back to all data needed on a calculation. This a much smaller requirement compared to buffering the whole frame and performing random memory access as initially speculated, and thus this module was rewritten in Verilog. Unfortunately, as with the delta filter, this filter failed to output video when tested on the hardware. Most likely the failure of this module and the delta filter module were due to the implementations of the AXIS-V, as the interface can be difficult to code manually.

5.2.4 Overlay

The overlay step in the video pipeline was also implemented in HLS (see Appendix C for a listing of the code). The result of testing the overlay step can be seen in Figure 35, Figure 36, and Figure 37. The original video (which in practice would be the video from the camera) is shown in Figure 35. The video shown in Figure 36 (which would be the video from the processing pipeline) is overlaid on top of the original video. Meaning that any white pixels will be shown in the output video and any black pixel will instead come from the original video. This results in a Figure 37 (which would be the output of the video processing pipeline) where there is pixel data from both of the original and the overlay video.

5.3 VDMA

As discussed in Section 4.3, the VDMA implementation in the project uses the Xilinx Video DMA controller IP. Bringing this module into the video pipeline was a difficult process due to the amount of content that needed to be brought into the project to make VDMA operational. Both proper configuration of the hardware at synthesis time and the configuration software that runs on the ARM cores at run time

[1]https://forums.xilinx.com/t5/DSP-and-Video/Controlling-VDMA-from-CPU-GPIO/td-p/684573

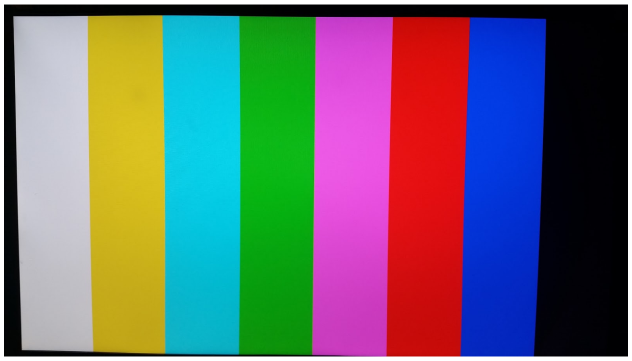


Figure 35: Original Video for Overlay Process

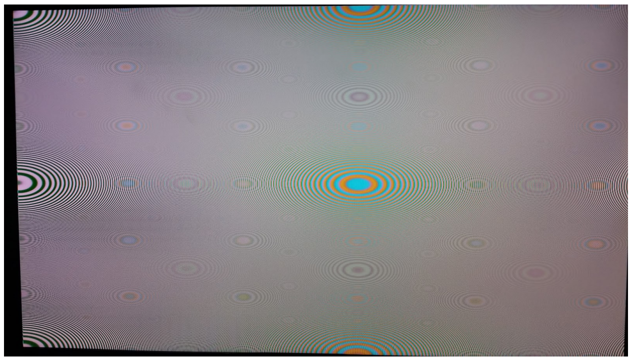


Figure 36: Video to be Overlayed

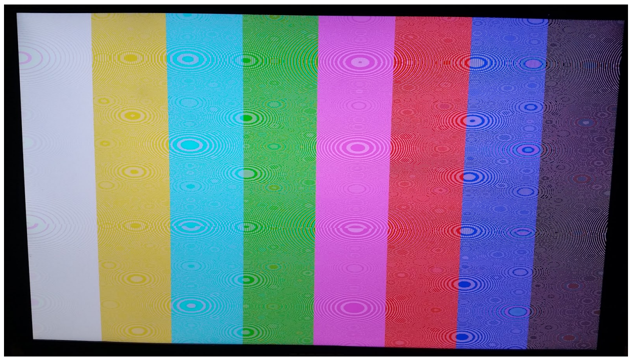


Figure 37: Result of Overlay Process

is critical to making VDMA system operational. Setting up the DMA system was especially challenging as there was not a good method for debugging the system as it is usually completely unresponsive during errors. Additionally, the DMA module is not easily simulated since hardware simulators do not have models for the Zynq PS and are unable to simulate configuration code execution. To get a minimum viable product, almost all of the debugging was done with hardware. Therefore, debugging took a long time and there was not much feedback about what part of DMA was failing and which setting was configured incorrectly. When there are a hundred settings, all of them have to be right, and there is little guidance about which one is wrong, implementation and testing can be difficult. Building the minimum viable DMA implementation took about two months. After reaching a point where output video was present, the configuration became significantly easier as better feedback was available and the remaining settings were much easier to understand.

Debugging the software portion of the configuration was significantly easier than the hardware portion. The BSP generates documentation alongside the C API that indicates all the operations that must be performed to configure DMA. The documentation also explains what all the settings are and how to use API functions to set them. In addition, there is a lot more information available when performing software configuration such as a register dump. This can identify what parts of the DMA unit are not working and can help spot some misconfiguration issues. For example, the Avnet example project incorrectly calculated the size of each pixel in bytes. This led to a half frame issue where only the left half of each frame was copied. This could be seen not only in the output video but also because a register dump indicated pixels were four bytes wide but they are actually two bytes wide.

5.4 Output video processing

The output video processing blocks were developed in HLS and tested using test pattern generators. Overall, this verified that blocks were working as intended. One issue with testing this way was that the system could only be tested on the DK2 when outputting two upscaled streams side by side. This was because due to the DK2's rotated screen, the resolution was not supported by a desktop monitor. This meant that only one component that depended on the DK2 could be tested at a time. In order to help circumvent this problem, an IP block was written in order to allow testing of one upscaled stream on a normal monitor. This block simply took the upscaled video for one half of the DK2 and surrounded the image with 0's in order to fit in a normal resolution. In addition a test bench was written for the upscaler in order to allow for testing without the hardware. This used HLS's C test bench feature which runs the test bench against the unsynthesized C code using a standard compiler. The test bench simply checked that the output for a row met the expected output.

One initial approach taken to upscale the video which didn't work was to use the HLS video library. This libray is an implementation of the OpenCV library which can be synthesized in HLS. This library had a resize function which the team attempted to use, however this function used excessive FPGA resources and didn't output correct video. After further research the team determined the reason was that the resize function was only intended for downscaling, which was not documented in the wiki about the library. At this point the team decided to use the nearest neighbor algorithm.

Based on the results from HLS, the upscaler was found to meet timing constraints with an additional latency of 20 clock cycles. The combine streams module also met timing requirements and had an additional latency of 20 clock cycles.

5.5 Barrel Projection

There were many challenges in developing barrel projection on the FPGA because to the team's knowledge it was a task that had never been attempted before. There were initial challenges that the algorithm for the DK2 was not published, as well as whether the FPGA had sufficient block RAM to buffer enough lines of video. The team also experienced issues with determining the right fixed point precision to use in the barrel math module. In order to help solve these, test fixtures were developed in order to allow the module to be simulated and verified without depending on the hardware.

5.5.1 Initial Algorithm Investigation

The first problem encountered when implementing barrel projection was figuring out the algorithm for the DK2. Detailed information could be found about the algorithm on the DK1 which uses a simple polynomial function to map the radius, however there was little information about the DK2 which uses different lenses. After reaching out to other Oculus Rift developers, it was discovered that the barrel projection code was included in an older version (0.4.4) of the Oculus Rift PC SDK. After reviewing this code, it was mostly apparent how to implement the algorithm, however there was an array of scaling factors that was difficult to locate in the code. In order to solve this problem, the team decided to run the SDK locally on a PC. The SDK was run in a debugger using Microsoft Visual Studio, with breakpoints at the points the parameters were used. This allowed the team to retrieve the value of the array from memory as well as verify their understanding of the algorithm.

After discovering the algorithm, the next step taken was to implement the algorithm in software on a PC in order to verify both that it was the correct algorithm and that the team completely understood how to implement it. In order to speed up the process, a pre-existing implementation of the DK1 algorithm, which was written in Javascript, was used. Since the algorithms are identical other than the function used to adjust the radius in polar space, the team was able to simply substitute the function without needing to write the other scaffolding around it.

The team desired to implement the barrel projection completely in programmable logic, which would require buffering video in a location that allowed for efficient random access of data. It was determined the the most efficient way to do this would be to use the block RAM (BRAM) in the FPGA as it has less latency than VDMA. There were concerns that there would not be enough space in the FPGA's BRAM to buffer all the needed lines of video. In order to investigate whether this would be a problem, the team decided to implement the algorithm in MATLAB. In addition to verifying if implementing the algorithm in hardware was feasible, this would allow for quick generation of a look-up table in case an alternate approach was needed. After this investigation it was determined that up to 60 lines of input needed to be buffered at a time, which was possible with the hardware.

5.5.2 Implementation

Initially, the math module of the barrel projection was developed using HLS, in order to abstract away details about the AXIS-V interface and fixed point math. This led to issues however where the AXIS-V interfaces defined by HLS did not work as the team expected. It was decided that it would be better to implement the block in Verilog instead, as that would make it easier to debug in the simulator, and allow full control over all interfaces.

In order to test and debug the barrel projection block, a simulation test bench was developed. The test bench passes the barrel memory video data over the AXIS-V interface until enough of a buffer had been built to start the math block. The math block then starts generating coordinates which can be observed and verified against the results in MATLAB. In addition, the test bench was configured to save a look-up table generated from the output of the barrel math module in a csv format. Using an additional MATLAB script, this look-up table could be applied to an image, allowing the team to verify whether the module was working without integrating it into the video pipeline.

In addition to the test bench developed for the entire barrel projection module described above, an additional test bench was developed to test the barrel math portion of the module on its own. This was done because the test bench described above could only be run on synthesized code due to faulty simulation models on some of the other IP. The test bench for the barrel projection can be run in behavioral simulation, which allows for faster testing and debugging. In addition, it made it easier to view specific internal registers which were optimized out of the synthesized code. A listing of the barrel projection testing code is located in Appendix E.

5.5.3 Results

After testing, it was determined that the barrel projection math module was working properly. Images generated using the look-up table were compared to the images generated from MATLAB and found to be nearly identical. An example of the input and output of the barrel projection module can be seen in

Figure 38. In addition, the new positions of the pixels in several rows were graphed as seen in Figure 39. As can be seen, in row 1 the y position increases to a maximum and then bows upwards (towards smaller values) as it is in the top of the frame. In comparison, in row 896, the y coordinates bow downwards instead as the row is in the bottom half of the image. In both images, the x position seems to follow a slight S curve, which makes sense as the x coordinates should not be changing dramatically.

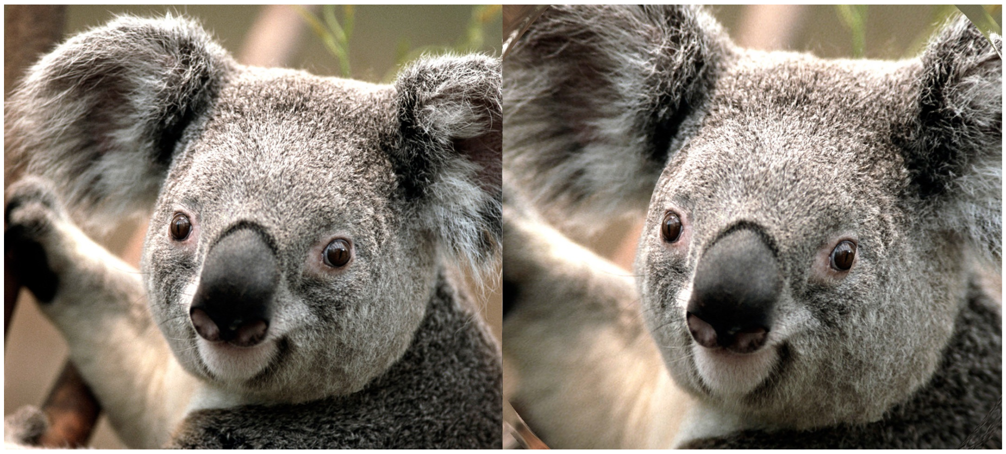
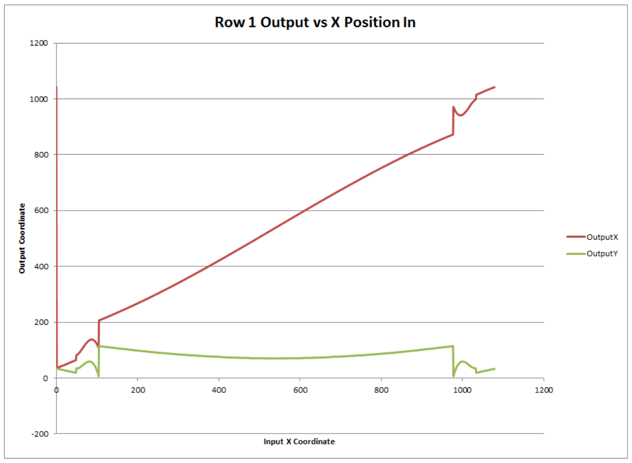
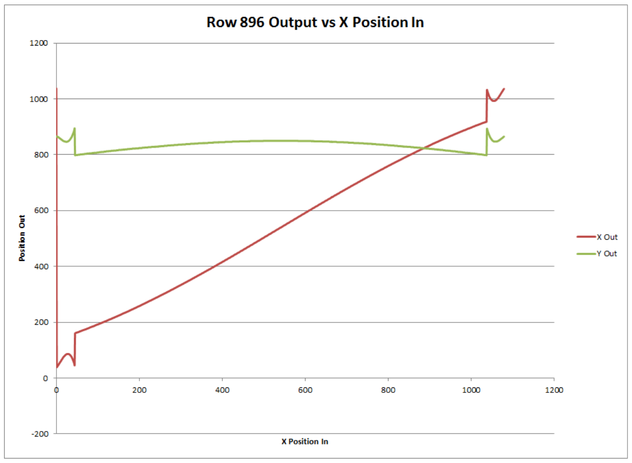


Figure 38: Barrel projection input (left) and output (right)



(a) Row 1



(b) Row 896

Figure 39: Output X and Y position vs input position

While the math module worked correctly in simulation, the team ran out of time to successfully test it on the hardware due to the barrel projection memory module. The memory module did not output correctly formed AXIS-V, causing the output of the system to be blank. Due to the fact that both modules were successfully tested in simulation, the team believes the issue was due to an error in the AXIS-V wrapper. Based on the simulation results, the team believes that if the output interface blocks were fixed, the module would work on the hardware.

5.6 Video Output

While developing the video output section several issues were encountered. This section describes the development procedures used to debug these issues, and the results.

5.6.1 Development Process

In the beginning of the project, the video output block needed to be tested before the camera capture and input interfaces were completed. In order to simultaneously develop both blocks, the video input block was replaced with a Xilinx test pattern generator. This generator always output a valid AXI signal displaying color bars. This allowed both simultaneous development and served as a fallback if the video stopped working to determine whether the output was broken, or another component was not working.

One of the major issues encountered throughout the project was that the video would not lock. What this means is that the AXI stream to the video out block could not synchronize the timing with the video and therefore would not produce any output. In order to detect if this was a problem, the port indicating whether the signal was locked was connected to one of the onboard LEDs. The AXI to video out block also produces signals indicating if its on board FIFO is empty or full, which were also connected to LEDs. Common causes of these problems were a malformed AXIS-V signal, or timings that were incorrect for the resolution of video being used.

Another issue encountered with this block was that the timings were incorrect for the DK2, indicated by the status light on the DK2 blinking between orange and blue. This was due to the fact that the DK2 uses non-standard video timings. The issue was solved by getting the EDID data for the DK2 as described in Section 4.8.2.

5.6.2 Results

Overall, the video output block worked as intended. Side by side video was output to the Oculus without tearing or timing errors. One issue that still remains however, is that the video wraps around on the DK2 screen. This issue was debugged by substituting a test pattern generator before each of the modules. The bar did not disappear even when the test pattern generator was plugged directly into the AXI stream to video out block. This means that the problem was introduced by the AXI stream to video out block. The team decided not to continue debugging the issue and instead advance the video processing filters, since it did not break the implementation and would require debugging Xilinx IP that the team does not have the source code for. This bug would be an issue that future iterations of the project could investigate.

5.7 Additional Tools & Techniques

This section will describe some techniques the team used to make working on the project easier that are not components of the final product. The primary methods are out of context runs, source control, and PCB design.

5.7.1 Out of Context Runs

Vivado contains a feature that allows synthesis to be performed independently for different parts of a design called "out-of-context for synthesis". This is similar to the unix make command: only sources that have been changed will be rebuilt. Since many parts of the project can take more than 10 minutes each to build (i.e. VDMA), placing these in their own contexts can be a major time saver. A full rebuild of this project takes over 40 minutes. A 40 minute synthesis and implementation time can make iterative testing nearly impossible.

5.7.2 Source Control

Source control is an essential feature for any code project, regardless of target technology. It allows for easy synchronization of source files between team members, and provides a listing of known working copies of the source code. To this end, the team decided to use Git as the source control program, and hosted the project on Github. Vivado does not have a built in function for source control such as interfacing with a Git repository. Adding a source control function is not trivial, but can be done through TCL scripting. The source control process involves not storing the Vivado project directory in the repository because it includes metadata which is not portable between different computers. Vivado frequently uses absolute directory paths to reference the working directory of the project which leads to issues as these paths will not be the same between systems. The project directory also contains automatically generated files that are frequently very large, extraneous, and conflict with design sources. To avoid storing these and the project metadata, only the design sources are stored in Git. A TCL script is used to rebuild the project each time it is updated from the repository. The design sources come in three varieties: HDL, block diagrams, and additional IP. A common way of organizing these is to put each in its own directory. This is referred to as the NOA directory tree standard and although the name is not well known, the format is a very common hierarchy for storing sources in FPGA projects. To rebuild the project, a generic TCL script is used to import each source type. Sources in the HDL folder are simply added to the project. The block designs are exported as TCL scripts that list the IPs that are inside them, the settings configured on these IPs, and how they are wired together. These TCL scripts are called from within the main TCL script in order to rebuild the block diagrams from scratch. The IP folder is added as an IP repository, which is a search path Vivado uses for finding referenced

IP from other parts of the design. The TCL script also needs to be manually coded with certain special cases: top level IP not inside a block design, and out of context runs. A copy of a TCL script that performs this is included in the project archive.

5.7.3 Camera PCBs

Due to many issues the team was having with using the same pin out every time the cameras were connected to the ZedBoard, custom cables and PCBs were designed to plug the cameras into the ZedBoard. These proved to be helpful in reducing the issues with poor camera connections and constant changes to the constraints file to accommodate different pin outs. Two versions were designed and built. The first version used two ribbon cables terminated in a ribbon to header pin adapter on the PMOD port end and manually soldered into the camera's pin out on the camera end (Figure 40). Even though the soldered end was also heatshrinked and stress relieved, the connection was unreliable and wires frequently broke off or became internally disconnected. A second cable was developed that had a single ribbon cable with ribbon to header pin adapters on both sides with a straight through pin out. A PCB was designed that interfaced this cable with the PMOD connectors on the ZedBoard. The final product is pictured in Figure 41.

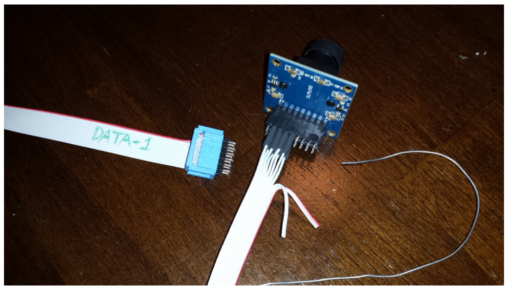


Figure 40: Assembly of first camera cables

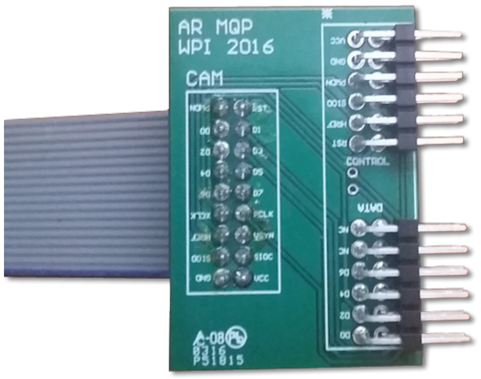


Figure 41: Camera PCB and Ribbon Cable

6 Conclusions

This project demonstrates that FPGAs can be utilized as a platform to do augmented reality processing. Each part of an augmented reality process has been implemented: capturing real time video, processing the video to display information to the user, and outputting video to a head mounted display. Video is captured with two stereo cameras and converted to an internal bus format. A four step filter process finds and highlights moving objects in front of the user: delta, threshold, median, and Sobel filters. Although highlighting moving objects is a simple application, the processing required is a foundation for many other applications of augmented reality. An additional video filter, barrel projection, adjusts for pin cushion distortion on an Oculus Rift DK2 display. An HDMI interface communicates with the Oculus Rift. In order to be a viable platform for augmented reality, the system also operates in real-time and with low latency. All video processing occurs in the programmable logic and runs with a latency of only hundreds of microseconds, which is much faster than frame buffer encumbered desktop solutions operating in the 10-30 millisecond range. These accomplishments show that FPGAs are not only capable of augmented reality but add improvements, and can be worth the additional effort to develop on.

At the time of submission, the project is nearly complete. All portions of the project have been developed and tested, but some have not been successfully implemented in hardware. The camera interface, Video Direct Memory Access, and HDMI output systems are fully operational in hardware. Due to running out of time, the video filters were not successfully integrated into and tested in hardware, however they do work in simulation. Video processing was not a novel part of the project as it was already known that video processing can be done on FPGAs, thus the conclusions of this project are not affected. However, it is a good example of the trade off of implementation difficulty between FPGAs and desktop computers. The Software Development Kits (SDKs) available on desktops significantly streamline the process of designing augmented reality systems.

6.1 Future Work

While this project showed that FPGAs are a more compact and less expensive platform for Augmented Reality than current PC based solutions, there is much that can be done to further the development on FPGAs. Initial implementation of object tracking was completed, however additional features can be built on top of this including speed of the object, or implementing other object tracking algorithm pipelines.[31] A easy extension of the work already done would be other simple image processing such as surface detection.

Other simpler improvements can be made to the system in order to improve the overall performance of the system. The color issues experienced (explained in Section 5.1) throughout the system are an example. They would be most easily fixed by adding more connections to the HDMI output chip than what the ZedBoard provides. The problem with the chip on the ZedBoard is it is fixed to using YUV due to the the pins that are connected to the Zync. YUV is a more difficult standard to work with in terms of color than RGB. An alternative solution would be to invest more time into understanding the YUV standard and looking deeper into each stage of the video processing pipeline to determine what is causing the mismatch in color.

Another simple improvement would be replacing the cameras with higher quality cameras that feature external frame syncs (fsyncs). Having an external fsync would reduce the need for the system to maintain frame synchronization and reduce the overhead on the system. They would also fix a complicated issue with stereo synchronization: objects moving in front of the user may appear in different places due to the difference in time when each camera captured the image. In addition, increasing the quality of the cameras would lead to more dependable performance in addition to higher resolution which would eliminate the need for an upscaling stage, decreasing the amount of logic that must be implemented. Lastly, higher frame rate would improve user experience.

Related to frame rate, another improvement to the system would be to increase the clock speed of the system (or upgrade to a higher performance Zynq chip that supports higher clock speeds) in order to increase the frame rate of the system overall. The current consumer version of the Oculus Rift uses 2 1080x1200 screens (an input resolution of 2160x1200) at 90 FPS. This is a small increase in resolution as well as a small increase in frame rate, which would require a faster clock to achieve.

A different type of improvement that could be made to the system would be to design a custom PCB and harness for the system. This PCB would include the Zynq chip and only the necessary hardware from the ZedBoard. This could drastically reduce the size of the system (as the project is currently not using many of the features available on the ZedBoard) and potentially be used as an attachment for the Oculus Rift rather than a separate piece. It would also allow the system to be used as a mobile system if combined with a battery.

A potentially major upgrade for the project could be to convert the system to use the consumer version (CV1) of the Oculus Rift rather than the Development Kit 2 (DK2) version. This would allow for a more advanced system as well as a better experience for the user. The CV1 features many improvements over the DK2 in terms of the user's experience of the video, such as increased resolution and frame rate. Alternatively, using a see through screen design as seen in the Microsoft Hololens or the Google Glass could be investigated. This would reduce the amount of computation on the system and allow for more data processing rather than video output processing.

References

[1] R. Metz, *Four important things to expect in virtual reality in 2016*, 2015. [Online]. Available: https://www.technologyreview.com/s/545011/four-important-things-to-expect-in-virtual-reality-in-2016/.

[2] M. Mihelj, D. Novak, and S. Beguš, *Virtual Reality Technology and Applications.* Springer Netherlands, 2014, ISBN: 978-94-007-6910-6. DOI: 10.1007/978-94-007-6910-6.

[3] D. A. Guttentag, "Virtual reality: applications and implications for tourism", *Tourism Management*, vol. 31, no. 5, pp. 637–651, 2010, ISSN: 02615177. DOI: 10.1016/j.tourman.2009.07.003.

[4] M. Pesce, *Htc executive director of marketing, jeff gattis wears the htc re vive*, 2015. [Online]. Available: https://www.flickr.com/photos/pestoverde/16513847554.

[5] T. J. Buker, D. A. Vincenzi, and J. E. Deaton, "The effect of apparent latency on simulator sickness while using a see-through helmet-mounted display: reducing apparent latency with predictive compensation", *Human Factors: The Journal of the Human Factors and Ergonomics Society*, vol. 54, no. 2, pp. 235–249, Apr. 2012, ISSN: 0018-7208. DOI: 10.1177/0018720811428734. [Online]. Available: http://hfs.sagepub.com/cgi/doi/10.1177/0018720811428734.

[6] Avnet, *Zedboard.* [Online]. Available: http://zedboard.org/.

[7] Oculus, *Time machine vr.* [Online]. Available: https://www2.oculus.com/experiences/app/1092474780792679/.

[8] ——, *Subnautica.* [Online]. Available: https://www2.oculus.com/experiences/app/993520564054974/.

[9] Virtual Reality Society, *Applications of virtual reality*, 2016. [Online]. Available: http://www.vrs.org.uk/virtual-reality-applications/.

[10] Edmunds, *Nissan 240sx.* [Online]. Available: http://www.edmunds.com/nissan/240sx/.

[11] G. Brindusescu, "Continental shows its augmented reality head-up display for 2017", *Autoevolution*,

[12] IFIXIT, *Oculus rift development kit 2 teardown*, 2014. [Online]. Available: https://www.ifixit.com/Teardown/Oculus+Rift+Development+Kit+2+Teardown/27613.

[13] Oculus, *Oculus rift — oculus.* [Online]. Available: https://www.oculus.com/en-us/rift/.

[14] Atman Binstock, *Powering the rift*, 2015. [Online]. Available: https://www.oculus.com/en-us/blog/powering-the-rift/.

[15] R. Madi, R. Lahoud, and B. Sawan, "Face recognition on fpga", *Spring Term Report,*, 2006. [Online]. Available: http://wwwlb.aub.edu.lb/fea/ece/research/Documents/Report/fyp%7B%5C_%7D0506/4%7B%5C_%7DReport.pdf.

[16] M. Ye, W. ZHOU, and W. GU, "Fpga based real-time image filtering and edge detection", *Chinese Journal of Sensors and Actuators*, 2007. [Online]. Available: `http://en.cnki.com.cn/Article%7B%5C_%7Den/CJFDTOTAL-CGJS200703034.htm`.

[17] Microsoft, *Microsoft hololens*. [Online]. Available: `https://www.microsoft.com/microsoft-hololens/en-us/hardware`.

[18] Leap Motion, *Leap motion desktop*. [Online]. Available: `https://www.leapmotion.com/product/desktop`.

[19] ——, *Hmd/vr/ar*. [Online]. Available: `https://www.leapmotion.com/solutions/vr`.

[20] Omnivision, "Ov7670/ov7171 cmos vga (640x480) camerachip tm with omnipixel ® technology o mni ision®", Tech. Rep., 2005. [Online]. Available: `http://www.voti.nl/docs/OV7670.pdf`.

[21] M. Field, *Zedboard ov7670*, 2013. [Online]. Available: `http://hamsterworks.co.nz/mediawiki/index.php/Zedboard%7B%5C%7D%7B%5C_%7DOV7670`.

[22] Xilinx, *Axi4-stream video ip and system design guide*, 2016. [Online]. Available: `http://www.xilinx.com/support/documentation/ip%7B%5C_%7Ddocumentation/axi%7B%5C_%7Dvideoip/v1%7B%5C_%7D0/ug934%7B%5C_%7Daxi%7B%5C_%7DvideoIP.pdf`.

[23] Vanessaezekowitz, *File:tearing (simulated).jpg*, 2009. [Online]. Available: `https://commons.wikimedia.org/wiki/File:Tearing%7B%5C_%7D(simulated).jpg`.

[24] G. Shrikanth and K. Subramanian, "Implementation of fpga-based object tracking algorithm", *Postgraduate Thesis,(SVCE, India)*, 2008. [Online]. Available: `http://www.cc.gatech.edu/grads/k/ksubrama/files/FPGA%7B%5C_%7DReport.pdf`.

[25] *Edge detection (image processing) part 1*. [Online]. Available: `http://what-when-how.com/embedded-image-processing-on-the-tms320c6000-dsp/edge-detection-image-processing-part-1/`.

[26] J. L. Smith, "Implementing median filters in xc4000e fpgas", *XCELL*, no. 23, p. 16, 1996. [Online]. Available: `http://www.xilinx.com/publications/archives/xcell/Xcell23.pdf`.

[27] *Nearest neighbor image scaling*, 2007. [Online]. Available: `http://tech-algorithm.com/articles/nearest-neighbor-image-scaling`.

[28] N. Whiting, *Integrating the oculus rift into unreal engine 4*, 2013. [Online]. Available: `http://gamasutra.com/blogs/nickwhiting/20130611/194007/integrating/%7B%5C_%7Dthe/%7B%5C_%7Doculus/%7B%5C_%7Drift/%7B%5C_%7Dinto/%7B%5C_%7Dunreal/%7B%5C_%7Dengine/%7B%5C_%7D4.php`.

[29] Xilinx Inc, "Cordic v6.0 logicore ip product guide vivado design suite", 2015. [Online]. Available: `http://www.xilinx.com/support/documentation/ip%7B%5C_%7Ddocumentation/cordic/v6%7B%5C_%7D0/pg105-cordic.pdf`.

[30] S. Sudhakaran, *What's the difference between yuv, yiq, ypbpr and ycbcr?*, 2013. [Online]. Available: `http://wolfcrow.com/blog/whats-the-difference-between-yuv-yiq-ypbpr-and-ycbcr/`.

[31] J. Cho, S. Jin, X. Pham, J. Jeon, J. Byun, and H. Kang, "A real-time object tracking system using a particle filter", in *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*, IEEE, Oct. 2006, pp. 2822–2827, ISBN: 1-4244-0258-1. DOI: `10.1109/IROS.2006.282066`. [Online]. Available: `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4058820`.

Appendix A Delta Filter Implementation (HLS)

The delta filter hls code is located in src/hls/threshold.cpp

Appendix B Thresholding Filter Implementation (HLS)

The thresholding code is located in src/hls/threshold.cpp

Appendix C Video Overlay (HLS)

The Video Overlay code is located in src/hls/overlay.cpp

Appendix D Camera Register Programming

The camera register code is located in src/hdl/camera_interface.vdl

Appendix E Barrel Projection Testing Code

The matlab code is located in src/matlab_scripts

The Verilog test benches are located in src/hdl/mathTester.v and src/hdl/mem_interface_testbench.v

Appendix F Barrel Projection

The barrel math block is located in src/hdl/barrelMath.v

The barrel memory block is located in src/hdl/barrel_mem_interface.v

Appendix G Upscaling, Rotation and Combining code

The code for the upscaler code is located in src/hls/upscale.cpp

The code for the stream combiner is located in src/hls/combineStreams.cpp