7-2021

Real-Time UAV Pose Estimation and Tracking Using FPGA Accelerated April Tag

Ethan Tola
ext9285@rit.edu

Real-Time UAV Pose Estimation and Tracking Using FPGA Accelerated April Tag

Ethan Tola

Real-Time UAV Pose Estimation and Tracking Using FPGA Accelerated April Tag

Ethan Tola

July 2021

A Thesis Submitted
in Partial Fulfillment
of the Requirements for the Degree of
Master of Science
in
Computer Engineering

RIT | Kate Gleason College of Engineering

Department of Computer Engineering

Real-Time UAV Pose Estimation and Tracking Using FPGA Accelerated April Tag

Ethan Tola

Committee Approval:

Daniel Kaputa *Advisor* Date
Department of Computer Engineering Technology

Marcin Lukowiak Date
Department of Computer Engineering

Louis Beato Date
Department of Computer Engineering

Abstract

April Tags and other passive fiducial markers are widely used to determine localization using a monocular camera. It utilizes specialized algorithms that detect markers to calculate their orientation and distance in three dimensional (3-D) space. The video and image processing steps performed to use these fiducial systems dominate the computation time of the algorithms. Low latency is a key component for the real-time application of these fiducial markers. The drawbacks of performing the video and image processing in software is the difficulty in performing the same operation in parallel effectively. Specialized hardware instantiations with the same algorithms can efficiently parallelize them as well as operate on the image in a streaming fashion. Compared to graphics processing units (GPUs) that also perform well in the field, field programmable gate arrays (FPGAs) operate with less power, making them optimal with tight power constraints. This research describes such an optimization for the April Tag algorithm on an unmanned aerial vehicle with an embedded platform to perform real-time pose estimation, tracking, and localization in GPS-denied (global positioning system) environments at 30 frames per second (FPS) by converting the initial embedded C/C++ solution to a heterogeneous one through hardware acceleration. It compares the size, accuracy, and speed of the April Tag algorithm's various implementations. The initial solution operated at around 2 FPS while the final solution, a novel heterogeneous algorithm on the Fusion 2 Zynq 7020 system on chip (SoC), operated at around 43 FPS using hardware acceleration. The research proposes a pipeline that breaks the algorithm into distinct steps where portions of it can be improved by utilizing algorithms optimized to run on a FPGA. Additional steps were made to further reduce the hardware algorithm's resource utilization. Each step in the software was compared against its hardware counterpart using its utilization and timing as benchmarks.

Contents

List of Figures

List of Tables

Abbreviations

2-D	two dimensional
3-D	three dimensional
API	application program interface
arctan	arctangent
AXI	advanced extensible interface
BRAM	block random access memory
CCA	connected component analysis
CCL	connected component labeling
CORDIC	coordinate rotation digital computer
CPU	central processing unit
DMA	direct memory transfer
DRAM	dynamic random access memory
DSP	digital signal processing
DUT	device under test
FF	flip flop
FPGA	field programmable gate array
FPS	frames per second
FPU	floating point unit
GPS	global positioning system
GPU	graphics processing unit
GUI	graphical user interface
HDL	hardware description language
I/O	input output

IMU	inertial measurement unit
IP	intellectual property
LSB	least significant bit
LUT	look up table
MMCM	mixed mode clock manager
MSB	most significant bit
RAM	random access memory
RGB	red, green, blue
SoC	system on chip
SRAM	static random access memory
UAV	unmanned aerial vehicle
UI	user interface
VDMA	video direct memory transfer
VHDL	VHSIC hardware description language

Chapter 1

Introduction

Localization has been a topic of concern for many system engineers as not only unmanned aerial vehicles (UAVs), but all robotic systems are becoming more integrated into our everyday life. Generally, localization is the ability to determine and track a system's position and orientation. Robotic systems need to understand their location within an environment to properly maneuver through it and accomplish tasks. Applications can require two forms of localization, namely absolute and relative. Absolute localization gives the position of the robotic system relative to the entire environment. This can also be considered as absolute position. global positioning system (GPS) is an example of this which provides the position of an agent on Earth's surface. Relative localization gives the position of an agent relative to either a sub-section of the environment or itself. There are various methods of computing this but inertial measurement units (IMUs) are typically used. These are combinations of various sensors, most commonly an accelerometer, gyroscope, and magnetometer, that measure the agent's orientation and can estimates its position with dead reckoning.

One of the advantages of utilizing a GPS and an IMU to perform localization is the high accuracy they provide. GPS systems are well established and fairly robust with many systems in place outside the engineer's control that make it reliable [2]. An engineer can assume that GPS satellites and base stations will remain functional. IMUs are entirely in the engineer's control and can utilize many different sensors to

accomplish the task of localization. Together, both systems can perform the desired localization of robotic systems with high accuracy. The primary disadvantage of the GPS system is when it becomes unavailable because the agent can no longer connect to the system. GPS systems require direct connections to either a base station, orbiting satellites, or both which becomes difficult in remote locations. The disadvantage with IMU systems is the incurred drift. Although IMUs excel at quickly computing a system's relative orientation (roll, pitch, yaw), they naturally incur drift over time due to the constant accumulation of errors when integrating.

This work introduces the need for a non-IMU and non-GPS based localization and stabilization process and the challenges that come with it. The April Tag fiducial system was the subject for analysis due to their efficient, robust, and accurate detection process and simple pose estimation algorithm [3]. April Tags provide both absolute positioning through prior knowledge of a specific tag's placement and relative positioning through the process of homography relative to the tag. Fiducial systems were first introduced as markers for calibration in virtual reality [4]. They have been further enhanced and heavily used in augmented reality through the creation of ARTags, and soon after, ARToolKit [5]. ARToolKit has been used in resource limited mobile phones to produce interactive frames at 20 Hz, illustrating its ability to perform image processing on edge devices [6]. With the introduction of April Tag, it was possible to leverage hardware acceleration while utilizing the resource efficient and robust algorithm to achieve faster frame rates on edge devices.

The research presented in this work analyzed the prior performance of algorithms presented by [3], [7], and [8] regarding the April Tag fiducial system. Olson's implementation operated at less than 3 frames per second (FPS) on the Fusion 2 embedded system, illustrating a clear need for a faster solution. For this work, the primary target for this system was to run the April Tag algorithm and perform localization at 30 FPS on a UAV. This was considered the baseline for a real-time application where

the UAV will have an acceptable reaction time to its surroundings. The new system implemented was built and targeted for the Fusion 2 stereo camera by Craft Cameras [9]. Its accuracy was evaluated to measure any degradation in performance upon building the new system. Additionally, timing was evaluated for the proposed system to measure the overall operating speed of the system. All of the tests were performed on the embedded hardware of the Fusion 2 system attached to the UAV. There were five key implementations of the April Tag algorithm that were focused on. The first was Olson's implementation, "April Tag Baseline", acting as the baseline for the next four implementations. The second was "PyApril Tag" which was also based on Olson's implementation but with some variations by Swatbotics [10]. It operated around 11 FPS, 3 times slower than the desired 30 FPS. The third implementation, "April Tag Baseline Optimized", was an optimized version of Olson's work done in C/C++, but still running at less than 3 FPS. The fourth implementation, "Ravven Tag MagTheta", moved the first few steps of "April Tag Baseline" onto the hardware. With this improvement, ideal frames would compute quickly, around 12 FPS but real world input would cause the process to slow to less than 1 FPS. The final implementation, "Ravven Tag CCA", improving upon "Ravven Tag MagTheta" by moving more steps onto the hardware, leveraging more advanced algorithms to fix the issues of the previous implementation. This final implementation performed consistently around 43 FPS, going well above the desired FPS.

Chapter 2 introduces the motivation for the research and goes more in depth on the underlying technology. Following, Chapter 3 highlights some of the previous research done in this area, their results and findings, and ultimately addressing observed drawbacks and future work. Chapter 4 depicts the final design for this research, highlighting key design choices and optimizations for heightened performance. Chapter 5 illustrates the results of the implemented technology, comparing against two different baseline algorithms, "April Tag Baseline" and "PyApril Tag".

Chapter 2

Background

2.1 Unmanned Aerial Vehicle Applications

UAVs today hold a large market value of about 19.22 billion USD in 2019 and was expected to reach a value of 59.18 billion USD by 2027 [11]. With a growing demand, faster, more efficient and more robust UAVs will be required. UAVs are deployed in a wide range of industries from (agricultural, delivery[12], construction, etc.) where the drone's autonomous ability to localize and stabilize itself is of utmost importance. For this paper, UAVs provide both a real-time constraint, instant feeedback, and a resource constraint environment to test and develop on.

The UAV baselined in this paper was the Fusion 1 and was equipped with a Zynq 7020 system on chip (SoC) which contains dual ARM Cortex-A9 processors and an integrated Xilinx field programmable gate array (FPGA) [13]. Specifically, the Snickerdoodle Black board by Krtkl was used as the device under test (DUT). A summary of important specifications are summarized in Table 2.1. Additional specifications can be referenced from their website documentation [14]. Attached to the Snickerdoodle board was the Fusion 2 system by Craft Cameras which adds two cameras that provides this board's video input. The cameras can operate at a maximum of 60 FPS and provide a 752x480 resolution [9]. The goal of this system was to develop the algorithms on the Fusion 2 stereo system and then port it to the

Table 2.1: Snickerdoodle Black Specifications

Component	Specification
CPU	32-bit Dual-Core ARM Cortex-A9 @866MHz
DRAM	1GB
SRAM	256kB
FPGA	1.3M gates/53,200 LUT-6
Distributed RAM	630kB

Table 2.2: Camera Specifications

Component	Specification
Camera	MT9V034
Resolution	752x480
FPS	60
Baseline	60 cm
Maximum Data Rate	27 Mp/s
Master Clock	27 MHz

Fusion 1 UAV system that had the same camera and SoC.

2.2 Fixed-Point Versus Floating Point

Many modern central processing units (CPUs) today have integrated floating point units (FPUs) that perform the extra logic that comes with utilizing either single or double point precision from IEEE 754 standard. Performing IEEE 754 floating or double point precision arithmetic without an FPU can be slow but comes with the expanded ability of a larger range and precision. To avoid such complicated logic, fixed-point algorithms were heavily used inside the FPGA which does not contain an integrated FPU. Additionally, research indicates that even with a FPU, fixed-point arithmetic performs faster as it simply does not need to compute the extra logic that encodes the IEEE 754 standard [15].

Fixed-point arithmetic operates similarly to integer arithmetic except for the fixed point notation that denotes where the decimal point for a number resides. Without the CPU's support, overflows or underflows can be more common when not done

carefully. Overflows and underflows occur when the operation exceeds the bounds of what can be represented in a given number of bits. Additional complexity can also be drawn from using fixed-point arithmetic but instead of it being computational harder, it can be harder to design to ensure that proper interpretation of each intermediate result keeps the inter-step values from overflowing or underflowing. This becomes necessary to consider when performing algorithms such as the CORDIC algorithm which is discussed further in Section 2.3 as the precision of the integrated look up table (LUT) changes drastically with more iterations.

2.3 CORDIC

Coordinate rotation digital computer (CORDIC) can be traditionally defined as a set of micro-rotations that approximate the arctangent (arctan) of two inputs, x and y. It can be classically described as

$$\begin{cases} x_1 = x \\ y_1 = y \\ \alpha_0 = 0 \end{cases} \quad \text{and} \quad \begin{cases} x_{i+1} = x_i + 2^{-i} * s_i * y_i \\ y_{i+1} = y_i + 2^{-i} * -s_i * x_i \\ \alpha_{i+1} = \alpha_i + -s_i * \arctan(2^{-i}) \end{cases}$$

where $s_i = sgn(y_i)$, it would eventually converge to the following:

$$\begin{cases} x_i \to K * \sqrt{x^2 + y^2} \\ y_i \to 0 \\ \alpha_i \to -\arctan\frac{y}{x} \end{cases}$$

These micro-rotations are restricted to powers of 2 which makes it exceptionally easy to implement in hardware, utilizing only shifts and adds. The arctan function will also be implemented utilizing fixed look-up tables for the depth of accuracy required. This function can be commonly referred to as atan2. Unlike the arctan function which

accepts a ratio of y/x and has an output range of $[-\pi/2, +\pi/2]$, atan2 maintains the sign of the x and y inputs while having an output range of $[-\pi, +\pi]$. Traditionally, atan2 is expressed as depicted in (2.1).

$$atan2(y,x) = \begin{cases} \arctan(y/x) & \text{if } x > 0 \\ \arctan(y/x) + \pi & \text{if } x < 0 \text{ and } y \geq 0 \\ \arctan(y/x) - \pi & \text{if } x < 0 \text{ and } y < 0 \\ \pi/2 & \text{if } x = 0 \text{ and } y > 0 \\ -\pi/2 & \text{if } x = 0 \text{ and } y < 0 \\ \text{undefined} & \text{if } x = 0 \text{ and } y = 0 \end{cases} \tag{2.1}$$

There are various implementations that utilize these cases to reduce the range of the output into the first quadrant [16]. This was ignored to first achieve simplicity as the hardware will most likely perform the same regardless of these minor optimizations. The important case to note is the undefined case where $x = 0$ and $y = 0$. This case will have to be handled as separate logic. Upon observing GNU's behavior of the atan2 function which returns 0 in this case, the implemented CORDIC algorithm will also return 0.

The CORDIC algorithm has several key constants. K, as depicted in (2.3) is the product of gain that is incurred during the process of micro-rotations. The value shown in (2.3) was computed and saved prior to the hardware's execution to prevent the need for unnecessary multiplications. It was necessary to multiply this scaling factor out to compute the gradient magnitude of a pixel alongside its gradient direction.

$$K(n) = \prod_{i=0}^{n} K_i = \prod_{i=0}^{n} \frac{1}{\sqrt{1 + 2^{-2i}}} \tag{2.2}$$

$$K = \lim_{n\to\infty} K(n) \approx 0.607252935008881256169 4 \tag{2.3}$$

There can be many ways to interpret the output of atan2 such as binary or radians but for the purpose of this research, the output was kept in radians for ease of portability to the rest of the April Tag algorithm which utilizes the direction as radians. The CORDIC algorithm was implemented in the hardware for its efficient usage of resources and acceptable latency. Compared to other atan2 approximations, CORDIC can be efficiently pipelined and put into the FPGA without consuming too many resources [16]. This can be attributed to both the iterative nature of the algorithm and its usage of only shifts, adds, and LUTs.

2.4 Connected Component Analysis

Connected component analysis (CCA) is common step in image processing for grouping pixels via labels to extract specific features from them. The classical algorithm requires two raster-scan passes through the image [17]. The first pass applies a temporary label to each pixel while building a look-up table for labels that have to be merged. These mergers are typically deferred to a later step as multiple merges can occur. The second pass performs these merges. These steps are often referred to as connected component labeling (CCL). Typically, the CCA algorithm is applied to a binary image that highlights the foreground pixels and ignore the background pixels. CCA can be applied to non-binary images and simply requires a different scheme. Overall, CCA is performed in a linear process as shown in Fig. 2.1.

The pre-processing step performs filtering, conversion or grayscale or binarization of the image. CCL, as described before, applies labels to groups of pixels on the image. Feature extraction utilizes these groups to pick out important aspects of the image. For this problem set, the desired features are the minimum and maximum

X and Y points of each blob. Post-processing is a general step that utilizes the extracted features for a higher level algorithm. CCA can operate on images for varying dimensions. For 2-D images, CCA is either performed as 4-way connected or 8-way connected. Illustrated in Fig. 2.2, 8-way connected CCA considers both the neighboring pixels and those touching the primary pixel's corners. 4-way connected CCA only compares with pixels touching its edge, A and C in this case.

Performing a high speed CCA algorithm in parallel on a general purpose processor is a non-trivial task. Optimizations to transform the algorithm into a single pass operation has been performed [18]. One primary advantage is the added ability to perform the CCA algorithm in hardware for processing streamed images [19][20]. This comes with additional complexity in the algorithm at the benefit of a high throughput, low latency algorithm. Furthermore, research has been done to improve the resource efficiency of hardware implementations of CCA through various novel techniques [21]. These optimizations illustrate the potential speed up that comes with utilizing this algorithm.

2.5 April Tag

Olson [3] describes the processing steps of the April Tag algorithm and splits it into distinct steps which are mimicked in the proposed pipeline. In layman terms, the April Tag system attempts to find four-sided regions or "quads" that have a darker

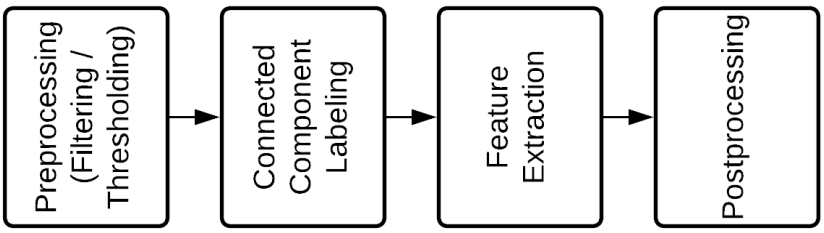


Figure 2.1: High Level Connected Component Analysis Dataflow.

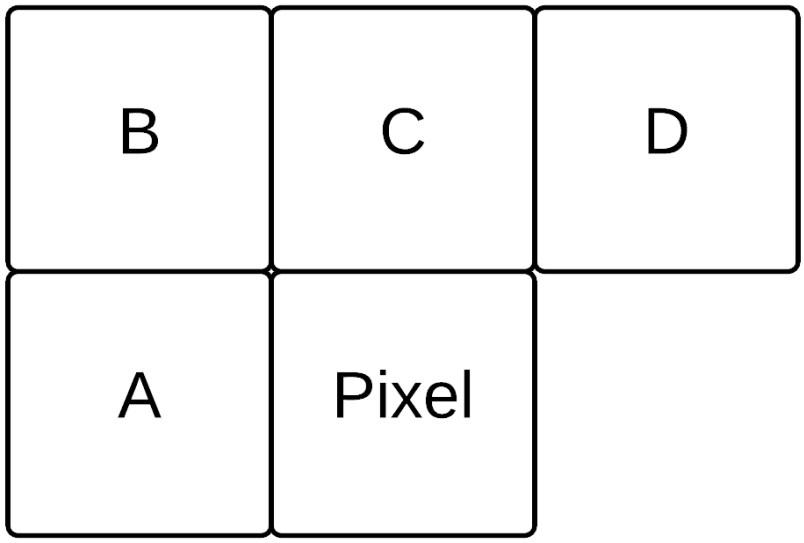


Figure 2.2: Example of 8-Way Connected Component Analysis.

interior than their exterior. As depicted in Fig. 2.3, the tags themselves were designed to have black and white borders to help accomplish this.

Olson describes four distinct phases to the April Tag process: detecting line segments, quad detection, payload decoding, and homography and extrinsics estimation. Table 2.3 breaks these sections down further upon examination of their code base. Each step of the process will be briefly covered.

The first step to most image and video processing systems is grayscaling. This is simply to reduce the computation space from 3 channels down to 1. Since the primary target is already black and white, there is little to no information loss in this process. The second step provides noise reduction with minimal losses to edge clarity due to the design of a Gaussian filter. This will be explored more in Section 4.3.2. The third step computes the gradient direction and magnitude of each pixel to cluster pixels into larger components. The fourth step deduces the weight of each edge depending on the calculated gradient magnitude and direction. The fifth step clusters these edges into larger components depending on how similar their gradient directions are.

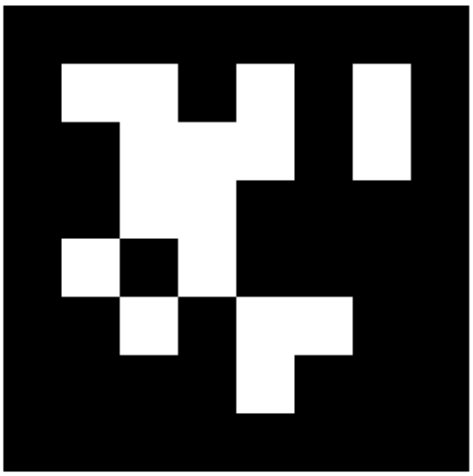


Figure 2.3: April Tag 36h11, Id = 0.

The clustering stage uses a graph-based approach in which each pixel is a node. The sixth step fits line segments to them, filtering out any edges that are too short. The seventh step connects segments that are close in proximity using a traditional least-squares procedure. The eighth step determines if the connections form quads by enforcing a winding rule and a depth-first search. The ninth step decodes these quads by computing the tag-relative coordinates and measures the bits of the black and white interior of the April Tag. The tenth step, after determining the code of the tag, removes any duplicate detection of the same quad. The eleventh computes the pose estimation of the April Tag through its homography.

Table 2.3: April Tag Algorithm: Step by Step

Phase	Step
Detecting Line Segments	Grayscale
	Gaussian Smoothing
	Gradient Magnitude & Direction
	Edge Extraction
	Clustering
	Segmentation
Quad Detection	Segment Connection
	Create Quads
Payload Decoding	Decode Quads
	Duplication Removal
Homography and Extrinsics Estimation	Pose Estimation

Chapter 3

Related Work

Localization has been done through many means and various sensors. As mentioned before, localization was traditionally done with GPS systems and IMUs. Like the systems that will follow, Mah describes the possibility of using a stereo video processing system to perform localization [22]. It differed from a fiducial localization system through the use of stereoscopic vision to perform triangulation using a red object to determine the depth, horizontal, and vertical distance of the object relative to the cameras. A major disadvantage of this approach was the need to perform the same tasks for both cameras which resulted in high memory usage and bandwidth. This created poor performance on resource constrained devices, thus not being able to operate at a real-time rate. A second disadvantage was the robustness of the algorithm to occlusions and large distances. Minor covering to the object would cause the algorithm to fail, making it not ideal for real-world environments where the object may not always be in direct line of sight. It's inaccuracies at large distances reduces the effective range of the algorithm, limiting it to smaller and tighter spaces.

To handle the high memory usage and bandwidth of performing the same task twice, April Tags were used to perform the same task as Mah's red object with a single camera instead. Olson [3] proposes an April Tag fiducial system similar to ARToolKit [4]. One of the main advantages of shifting to a fiducial system like April Tag was the speed and efficiency at which the tags can be detected. Once found, a quick process

can be done to compute the homography of the tag relative to the camera, thus producing the desired localization. The second advantage of this system was that it has been proven to be a robust system, even to variations in lighting, occlusions and noise. Olson illustrates the accuracy of April Tag as being with 2 degrees of error for off-axis angles from 0 to 90 degrees. Similarly, April Tag's are robust enough to have high accuracy for distance calculations up to 20 meters. Wang [7] worked with Olson to improve upon the existing April Tag algorithm, implementing a new threshold operation as well as a new segmentation algorithm. The threshold operation was done to make the system more robust to variations in lighting by implementing an adaptive thresholding algorithm. This was done by computing the local extrema within a 4x4 tile that merged with neighboring tiles in a 3x3 fashion. Although this algorithm was difficult to implement in hardware, it illustrates the need for adaptive thresholding. The new segmentation algorithm introduced the concept of connected component analysis to quickly join together edges that form quads.

To enhance this performance, Zhang [8] proposed the usage of an FPGA SoC fiducial system. Depending on the CPU and FPGA system, there can be advantages to using one over the other. Explored by Asano and others [23], the performance between a CPU, graphics processing unit (GPU), and an FPGA was explored specifically for image processing. The FPGA could out perform a multi-core CPU given large enough two dimensional (2-D) kernels but could never surpass a GPU. The main advantage the FPGA has over the other two systems is lower power consumption [24]. For both a resource and energy constrained environment such as a UAV, power efficiencies becomes an important factor. Zhang's work introduced the usage of an FPGA which illustrated clear advantages when applying 2-D kernels compared to the CPU but showed a slowdown when calculating gradient directions. One of the primary advantages of Zhang's approach was the clear speed-up that was acquired given hardware friendly computations such as applying 2-D kernels. This inspired the

usage of the FPGA to perform more tasks and redesigning the system to fully utilize the FPGA. Zhang's approach differs in that the hardware was working synchronously with the CPU. This can be seen as a disadvantage as it did not fully utilize the capabilities of FPGA hardware acceleration when performing image processing tasks. The following research illustrates a streaming framework that was done to perform image processing tasks as each pixel was read in. The second disadvantage was the atan2 approximation that was not hardware friendly but rather CPU friendly. There are many hardware optimized approximations for atan2 such as the CORDIC algorithm proposed in Section 2.3 that can perform the operation in a streaming fashion with a very small latency.

Chapter 4

Methodology

The entire design was split into two optimizations. The first optimization introduced moving "Grayscale", "Normalize", "Gaussian Smoothing", and "Gradient Magnitude and Direction" into hardware and in a streaming fashion. This optimization required "Gradient Magnitude and Direction" to be redesigned compared to [8] so the computation can be effectively done within a clock cycle. The second optimization moved "Edge Extraction", "Clustering", and "Segmentation" into the hardware with the same streaming theology. Fig. 4.1 illustrates the final block diagram that was implemented. The primary difference between the first and second optimization was the input output (I/O) out to the software. Otherwise, the second optimization was a superset of the first.

With many moving parts, it was necessary to derive a framework that would allow for easy testing and feedback of the system. Section 4.1 will examine the the framework topology of the system. Section 4.3 will explore the high level design of the various algorithms and their components, detailing how the pipeline was broken down and modified for the various optimizations. Section 4.4 will introduce the Matlab implementation and verification workflow. Lastly, Section 4.5 will discuss the Simulink implementation.

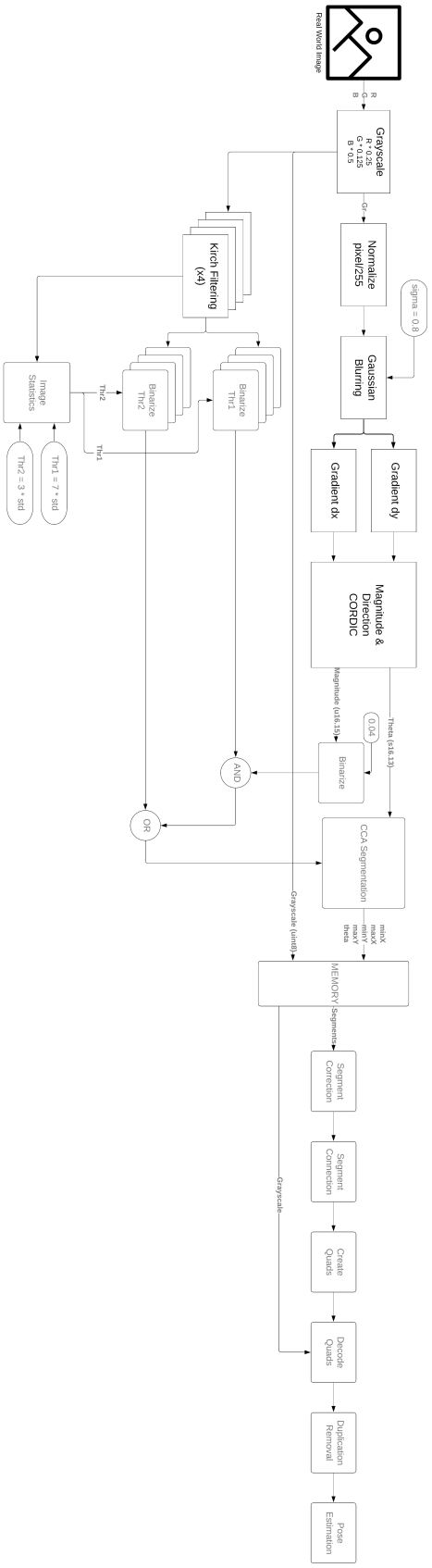


Figure 4.1: Second Optimization Block Diagram.

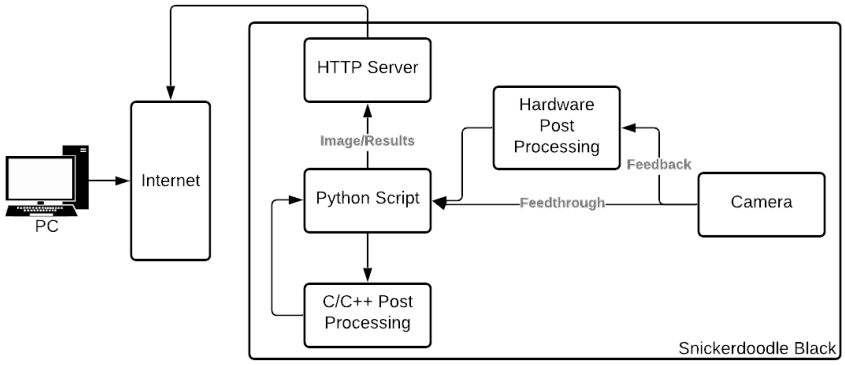


Figure 4.2: Python Framework Topology.

4.1 Framework Topology

The hardware testing and feedback systems leveraged Python's high level capabilities to coordinate between the C/C++ code implementation and the hardware optimizations. For testing, Python scripts were used to perform timing and compare inaccuracies between each algorithm implemented. For feedback, REMI [25] was used to create a local http server on the Snickerdoodle Black that would display the original image and the image outputs of the steps under test. A computer can then connect to the server and view the image outputs in real time.

To connect Python to the hardware, special kernel drivers were written to capture the output from memory and pass back references to the data output. As illustrated in Fig. 4.2, the script needed the feedthrough and feedback components. Feedthrough was leveraged to pass the original input from the camera so it can be verified with the original and "PyApril Tag" algorithm. The feedback component was the original image fed through the hardware post-processing block which had a varying output.

The Python script could configure the hardware to select whether the Simulink output would be fed back into the model or if it would take real images. As illustrated in Fig. 4.3, both images would be sent to memory through video direct memory transfer (VDMA) and leveraging advanced extensible interface (AXI) video stream

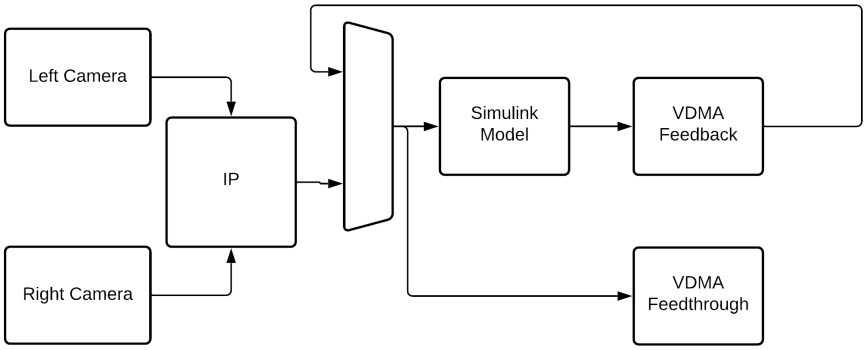


Figure 4.3: Fusion 2 Hardware Framework Topology.

protocol. Similarly, the C/C++ code was often configured and reconfigured to accept varying inputs. It was also configured to skip certain steps given they have already occurred in the hardware prior to receiving the data. These were referred to as on ramps for the code. The separation in these stages will be explained more in Section 4.3. Overall, this framework was primarily geared towards providing as much debugging and image output for each step as possible. It was flexible by maintaining a simple interface that was independent of the actual output data as well as pipelined for interoperability between code bases.

4.2 April Tag Implementations

As mentioned in the previous sections, there were various implementations of the April Tag algorithm. This section will formally introduce each implementation, their usage in this thesis, and how the modified implementations differ. They were split up into two categories: base library, whose code was left untouched by the publisher, and custom, whose code was modified from the base libraries.

4.2.1 Base Library April Tag Implementations

The "April Tag Baseline" implementation was a C/C++ library that was provided by Olson [3] from University of Michigan. The original code base was compiled and

used on the Fusion 2 system for timing and accuracy. Additionally, the algorithm was implemented in Matlab for ease of testing in that environment as well. Table 2.3 depicts the various steps this implementation used. Overall, this library was used as the baseline implementation to compare against for the other April Tag implementations.

The "PyApril Tag" implementation was a Python library provided by Swatbotics [10] that was built against Olson's [3] existing code base. The library leveraged a C/C++ backend that differed from Olson's implementation and pipeline, using other techniques such as using a contour-based quad detection algorithm. This implementation was used to illustrate an up-to-date April Tag algorithm that was published to Python which may be commonly used by others.

4.2.2 Custom April Tag Implementations

The "April Tag Baseline Optimized" implementation was the "April Tag Baseline" library with slight optimizations that leveraged the OpenCV and Eigen libraries. These were very simple optimizations that increased the algorithm's overall performance. This was done to have a cleaner and more organized code base that would later be broken down.

The "Ravven Tag MagTheta" implementation was the first to exhibit a hardware acceleration of the algorithm, moving "Grayscale", "Normalize", "Gaussian Smoothing", and "Gradient Magnitude and Direction" portions of Olson's April Tag algorithm onto the hardware. As it will be explored later, this would effectively remove those steps from the pipeline, ideally allowing the processing time to dramatically increase. This will be explored in-depth in Section 5.5, but the final results failed to meet the desired goal of 30 FPS, prompting for additional optimizations.

The "Ravven Tag CCA" implementation takes "Ravven Tag MagTheta" a step further, moving "Edge Extraction", "Clustering", and "Segmentation" onto hardware to improve upon the drawbacks of the previous algorithm. These steps required

Table 4.1: April Tag Algorithm Implementations.

	Implementation	Description
Base Libraries	April Tag Baseline	A C/C++ April Tag library provided by Olson [3] from University of Michigan.
	PyApril Tag	A Python April Tag library provided by Swatbotics that was also built against [3] but with optimizations [10].
Modified	April Tag Baseline Optimized	A C/C++ April Tag library that provides slight optimization on Olson's [3] existing code base.
	Ravven Tag MagTheta	An April Tag algorithm that moves "Grayscale", "Normalize", "Gaussian Smoothing", and "Gradient Magnitude and Direction" portions of Olson's April Tag algorithm onto hardware, leveraging the Fusion 2 SoC to provide optimizations.
	Ravven Tag CCA	An April Tag algorithm that builds off "Ravven Tag MagTheta" by moving "Edge Extraction", "Clustering", and "Segmentation" onto hardware to provide even greater optimizations.

additional algorithms to make the FPGA implementation efficient, robust, and fast. As mentioned previously, this final implementation met the desired goal of 30 FPS, operating around 43 FPS consistently. Table 4.1 tabularizes the various implementations.

4.3 High Level Design

Breaking apart the April Tag algorithm provided the capability to analysis and optimize portions of the algorithm independent of each other. The primary separation goals were to determine which parts of the algorithm could be optimized in the hardware and how that output would look like when interfacing with the C/C++ code that would complete the operation. Initially, a lot of this testing was done in Visual

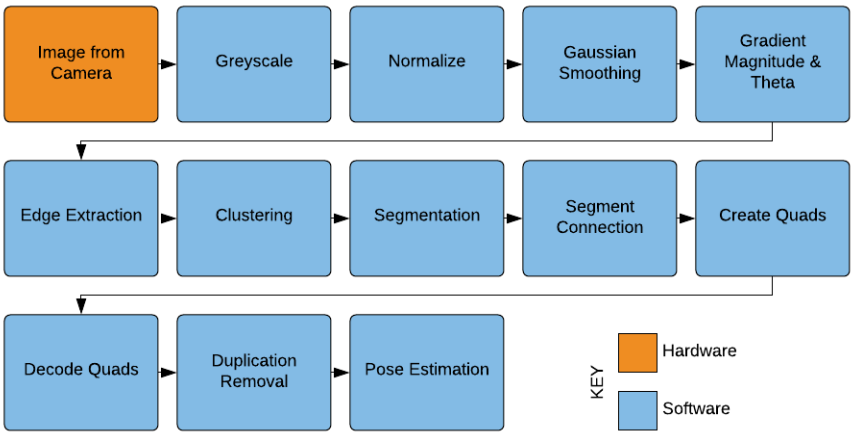


Figure 4.4: April Tag Software Pipeline.

Studio on a PC for quick prototyping and visual outputs of each step. Later, the code was transferred and compiled to the Fusion 2 Zynq 7020 processor. Fig. 4.4 illustrates the initial break down of the software code.

The first step, "Image from Camera", was included in this pipeline for completion. This process operates entirely in the hardware but can be configured by the software. The "Image from Camera" step produces an RGB888 output which was the starting data for the rest of the pipeline. Each of the following blocks and their observed outputs were segments of the code from [3] and [8]. Furthermore, Table 4.2 depicts the specific outputs of each step (and the inputs to the next step) as well as their data types and ranges, if applicable.

Upon initial inspection, it seemed that separating the first 4 steps, excluding the "Image from Camera", would be the most efficient way to gain a large speed up. Each of those blocks can be done quickly within a pixel clock cycle or pipelined to do so. Initial timings on a PC supported these findings as a majority of the time was spent applying the Gaussian blurring kernel and calculating the gradient magnitudes and directions for each pixel [8]. The "Edge Extraction" step began to perform more complex operations that would take more time to translate to a hardware efficient

Table 4.2: April Tag Software Step Outputs

Steps	Output/Data Type
Image from Camera	red, green, blue (RGB)/uint8
Grayscale	Gray/uint8
Normalize	float [0 1]
Gaussian Smoothing	float [0 1]
Gradient Magnitude and Direction	float [0 1] & float $[-\pi\ \pi]$
Edge Extraction	List<Edge>
Clustering	Map<int, XYWeight>
Segmentation	List
Segment Connection	List
Create Quads	List<Quad>
Decode Quads	List<Detection>
Duplication Removal	List<Detection>
Pose Estimation	Stats

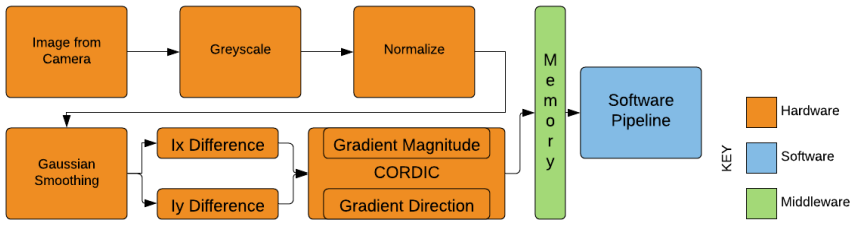


Figure 4.5: AprilTag Hardware Pipeline.

algorithm. Fig. 4.5 illustrates the proposed pipeline, providing greater insight on how the hardware components have changed.

As mentioned before, one of the primary advantages of utilizing an FPGA is the parallelism that it introduces. For this application, it was possible to compute both the gradient magnitude and gradient direction simultaneously with a CORDIC algorithm as illustrated in Section 2.3. The primary difference between the hardware components and the software pipeline was that the hardware operated in a stream fashion. Instead of the receiving the entire image after being read in from the camera, each pixel will be operated on. This effectively reduces the computation time of each image down to the clock cycles it takes for the first pixel to make it to memory.

The second phase of this thesis introduced a further reduction in the software

pipeline by moving "Edge Extraction", "Clustering", and "Segmentation" to the hardware and creating the additional step, "Segment Orientation Correction". The purpose of this modification and additional step will be discussed later in Section 4.3.4. The following subsections will provide a more in-depth discussion on the proposed methodology of implementing an optimized hardware version.

4.3.1 Grayscale

Grayscaling is a common and often necessary step in the image processing flow to limit the space in which computations are done. An example is illustrated in Fig. 4.6. Dedicating this process to the hardware allows for the image to be completely grayscaled by the time the entire image is read through as it can be performed on each pixel as its streamed in. This was done prior to "Normalize" to avoid performing the same division on three pixels. The traditional grayscaling formula used to convert RGB color space to grayscale is shown in 4.1.

$$Gray = 0.299 * Red + 0.587 * Green + 0.114 * Blue \tag{4.1}$$

Instead of computing three multiplications, a simple approximation that utilized shifts instead was used. The resulting equation is shown in 4.2 where each multiplication can be interpreted as a shift to the right, dividing the RGB components by 4, 2, and 8 respectively.

$$Gray = 0.25 * Red + 0.5 * Green + 0.125 * Blue \tag{4.2}$$

Although this saves time and space in regards to multiplications, it creates a slight luminescence change. The sum of the coefficients is no longer 1 which may cause the April Tag fiducial system's original robustness to light variations to diminish. It is assumed that by the nature of the April Tag being black and white, a small change

(a) Original Image

(b) Grayscale Image

Figure 4.6: Illustration of grayscale image.

0.06292	0.1250	0.06292
0.1250	0.2483	0.1250
0.06292	0.1250	0.06292

Figure 4.7: Gaussian 3x3 Kernel with $\sigma = 0.8$

in luminescence will have a minimal impact on accuracy.

4.3.2 Gaussian Smoothing

Common problems when dealing with images taken by imperfect cameras is noise. Noise can come from a wide variety of sources which can cause aliasing, hot/cold pixels, and many other undesired artifacts to present in the image. For this application, the primary source of noise was aliasing. Aliasing is formally defined as an effect that causes different signals to become indistinguishable when sampled. In digital image processing, aliasing is spatial, where moiré patterns that are present will distort the image. Low pass filters are typically used to reduce or remove this form of noise as spatially, it is rapid changes in pixel values. For this application, a Gaussian smoothing filter was used as the low pass filter, as depicted in Fig. 4.9. This filter was used for its symmetry, high frequency attenuation, and low edge distortions as depicted in Fig. 4.7.

0.2256	0.5488	0.2256

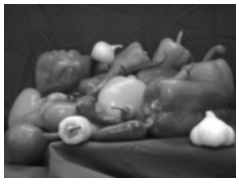
Figure 4.8: Gaussian Separable Kernel with $\sigma = 0.8$

The symmetrical properties of the Gaussian filter allows for it to be separable, where the 3x3 kernel that would traditionally slide across the image can be split into two one dimensional filters as depicted in Fig. 4.8. Separable filters are determined by the rank of the matrix formed. All separable filters must have a rank of 1 to indicate that the rows and columns are related linearly and can thus be broken into their horizontal and verticle components.

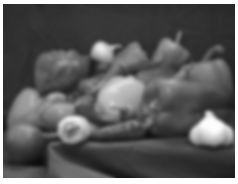
In hardware, this saves resources where the time complexity shifts from O(M * N * m * n) to O(M * N * (m + n)) where the image is N x M and the filter is m x n [26]. For a 3x3 kernel, this reduces the number of multipliers from 9 down to 4 when the same symmetry that can observed in Fig. 4.8 is utilized. Edge preservation was also a key aspect of the Gaussian smoothing kernel as the primary feature that needs to be extracted from the image were the edges surrounding the April Tag. This feature comes from the unequal weighting and averaging of the Gaussian coefficients, giving more weight to the center pixel than the surrounding ones.

(a) Grayscale of peppers.png

(b) Gaussian smoothing with $\sigma = 0.5$



(c) Gaussian smoothing with $\sigma = 2$



(d) Gaussian smoothing with $\sigma = 4$

Figure 4.9: Gaussian Smoothing with $\sigma = \{0.5, 2, 4\}$.

0	0	0
1	0	-1
0	0	0

0	1	0
0	0	0
0	-1	0

Figure 4.10: Ix (left) & Iy (right) 3x3 Kernel.

The Gaussian filter was generated using a $\sigma = 0.8$ as per [3] recommendation. When converting to fixed-point, this σ value proved to have a slight error in the corresponding coefficients as the sum of values were greater than 1. Similarly to the grayscale gain, this error will also add a small gain to the pixels. In this case, the error was minute enough to ignore.

4.3.3 Gradient Magnitude & Direction

Calculating the gradient magnitude and direction of each pixel was a necessary step to determining and connecting edges. "Gradient" in this process indicates the computation used to create the pixel values where their magnitude and direction would be calculated from. In both software and hardware, the gradient was computed using a simple difference to bring the edges to the foreground while ignoring the other pixels.

The Ix and Iy kernels produces an image that highlights edges horizontally and vertically respectively. These difference images are the values that are fed into the CORDIC algorithm as illustrated in 2.1. The CORDIC algorithm was leveraged at this point to perform simple calculations on the stream of pixels within a clock cycle. The original magnitude calculation, illustrated in (4.3), requires a square root computation which can be very complex when implemented in hardware. The

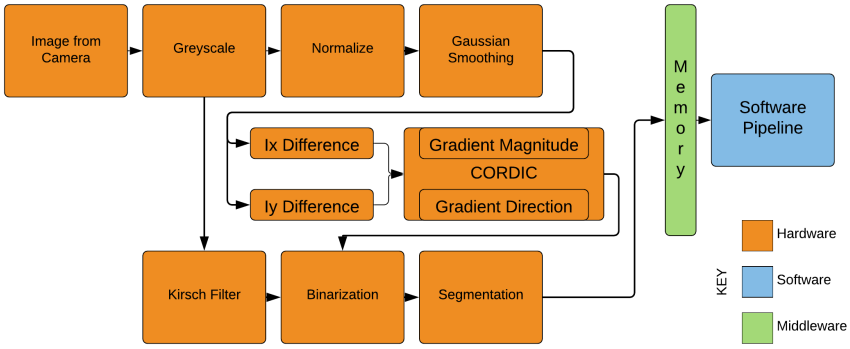


Figure 4.11: April Tag Hardware Pipeline with Segmentation.

CORDIC algorithm was able to approximate both the direction and magnitude of the pixel without significant additional software to compute both a square and a square root.

$$\sqrt{g_y^2 + g_x^2} \tag{4.3}$$

4.3.4 Edge Detection & Clustering

The second phase of the thesis introduced the usage of connected component analysis to replace "Edge Extraction", "Clustering", and "Segmentation" in the April Tag pipeline. The primary purpose of introducing this step was to provide an even greater increase over the baseline April Tag algorithm by [3] and the Python April Tag implementation. As depicted in Fig. 2.1, there were three primary components of the CCA algorithm that needed to be translated before being implemented in the April Tag algorithm. The first component, described as the pre-processing step, would need to binarize the image to bring forth the desired pixels. The second component will need to label the binarized image, creating groups of pixels where the third component can extract the desired features. The final step would be formatting the necessary data to be sent to the software pipeline through memory. Fig. 4.11 summarizes this.

As illustrated in Fig. 4.11, moving the "Segmentation" into hardware introduces

additional complexities such as a data dependency between the binarized Kirsch filters and the binarized "Gaussian Smoothing". Section 4.3.4.1 will discuss the motivation behind utilizing this first order filter. Section 4.3.4.2 introduces the concept of binarization and the complexities drawn from it. Section 4.3.4.3 proposes the implementation of the connected component analysis for the April Tag algorithm and the specific features that will be extracted.

4.3.4.1 Kirsch Filter

Kirsch filters, also known as Kirsch operators, is a first order derivative kernel, similar to the Ix and Iy differences, in the primary and secondary cardinal directions [27]. It finds the maximum edge strength in the 8 compass directions with weights depicted in Fig. 4.12.

Only these four kernels were leveraged in the design to maintain simplicity but also because the other directions would not provide additional information on the image. Considering the filters become symmetric afterward (i.e North and South kernels would produce two images with positive and negative values flipped), hardware resources can be saved by not implementing the other four kernels. This symmetry was achieved by computing the absolute value of the Kirsch filter. Kirsch filters were used here instead of other edge detection algorithms such as Sobel filters as the Kirsch filter does not blur the image like the Sobel filter. Discussed in greater depth in Section 4.3.4.3, it was desired that the edges be as thin and selective as possible since the features extracted were the minimum and maximum XY pairs for a total of four points.

The Kirsch filter was implemented as resource efficiently as possible by following a similar methodology as the separable Gaussian filter. In hardware, the values of pixels with the same coefficients were summed prior to multiplication. Additionally, to save a bit, the "3' components were multiplied with a positive three and then subtracted

5	5	5
-3	0	-3
-3	-3	-3

5	5	-3
5	0	-3
-3	-3	-3

5	-3	-3
5	0	-3
5	-3	-3

-3	-3	-3
5	0	-3
5	5	-3

Figure 4.12: North (top left), Northwest (top right), West (bottom left), and Southwest (bottom right) Kirsch Filter Kernels.

from the '5' components so the sign bit was left till the last possible moment.

4.3.4.2 Binarization

The binarization step simplifies the labeling portion of the CCA algorithm by clearly separating the background and foreground pixels. This process was applied to each Kirsch filter compass direction and the gradient magnitude. There were two thresholds applied to the Kirsch filter while a single, constant threshold was applied to the gradient magnitude. The primary goal of this was to overlap the pixels with high magnitudes and strong directional values with the moderately strong directional values. To maintain the robustness of the April Tag algorithm in varying lightning conditions, an adaptive threshold mechanism was chosen for the Kirsch filters. This adaptive mechanism computed the average and standard deviation of pixel values for the frame, taking only the top few pixel value percentages. This was done with simple binary operators between the three binary images to produce a single binary image to be fed into the "Segmentation" block. The final binary image operations can found in (4.4) where BW1 was the strong directional Kirsch filter, BW2 was the binarized gradient magnitude, and BW3 was the moderately strong directional Kirsch filter.

$$\mathrm{BW} = (\mathrm{BW1} \wedge \mathrm{BW2}) \vee \mathrm{BW3} \tag{4.4}$$

Computing and applying the statistics for an image in a streaming fashion is impossible. To avoid the need of storing the entire image in hardware and creating a frame phase delay, the mean and standard deviations of the previous frame was used. The thought process behind this was that for high frames rates, the statistics of the previous frame would not be drastically different. A dropout would occur during drastic changes in the environment as the previous frame statistics would not accurately reflect the current frame.

(a) Straight Line Segment. **(b)** Bent Line Segment.

Figure 4.13: Line Segment Primitives with Highlighted $minX$, $maxX$, $minY$, $maxY$ Coordinate Pairs

4.3.4.3 Segmentation

The "Segmentation" portion of this enhancement was the most complicated as it implemented the CCA algorithm in a single pass. A similar design to [20] was followed for the FPGA implementation of this algorithm with a adaptation to 4-way connected components. This modification was to maintain the focus on complete blobs and preventing the chance of the connected components no longer being a single, straight line. Additionally, to fit into the Fusion 2 topology explained in Section 4.1, the CCA algorithm will not leverage block random access memorys (BRAMs) for direct memory transfer (DMA) transfers to main memory. Instead, the extracted features will be dumped into a secondary BRAM that will stream out the segment data on the VDMA line, alternating between the five different feature data.

The two end points and the approximated segment direction were the features being extracted from the various Kirsch filters. This was done by initially extracting four coordinate pairs, minimum X ($minX$), maximum X ($maxX$), minimum Y ($minY$, and maximum Y ($maxY$) as depicted in Fig. 4.13. From these four points, six line segments can be drawn where the longest line segment can be assumed to be the actual line segment given the blob was a single edge. This process failed when there was a bend in the line, thus the line segment was finding the hypotenuse of the implied triangle rather than an edge. To limit this, the Kirsch filters were used to grab edges that were mostly flat through the North and West filters while diagonal lines were

Figure 4.14: Line primitives

captured with Northwest and Southwest filters. This portion of the algorithm was also put into hardware as the FPGA can find the four pairs of coordinates, approximate their lengths, and select the largest length all in parallel as this process has to be done for all four Kirsch filters. (4.5) mathematically describes the length approximation used. This allows for the computation to be done within a single clock cycle and avoids an unnecessary square root operation.

$$L = |dx| + |dy| \tag{4.5}$$

One of the primary assumptions for the Kirsch filters, binarization, and segmentation to work was that they were mutually exclusive, matching the line primitives depicted in Fig. 4.14.

If one of the line primitives were missing during the binarization process, the segments formed during segmentation would produce min/max coordinate points depicted in Fig. 4.13b. The quad would not have four segments but instead form a triangle, thus producing a false negative.

4.4 Matlab Implementation

A proper framework can increase the speed at which research, testing, and prototyping can be done. A large portion of using Matlab was to create and test a workflow that would increase a user's productivity. One of the primary advantages of utilizing Matlab in this workflow was its pre-existing integration with Simulink. Each step

of the processes, "Grayscale", "Gaussian Smoothing", and "Gradient Magnitude and Direction" were first written in Matlab to ensure that they function correctly. The images produced by each step were inspected and compared against any variations from the baseline code base. Although the hardware itself would be utilizing fixed point arithmetic versus Matlab's native floating point, simulating these impacts was done through casting to quantize the end results. Although this was not a perfect solution, it was necessary as there was a significant performance degradation when working with fi objects in Matlab for operations such as atan2.

The verification process implemented in Matlab to ensure the Simulink code operated correctly was a combination of "unit" and system level testing. A "unit" in this case would be a single portion of the algorithm such as "Gaussian Smoothing". It would be directly compared to the Matlab implementation, typically through subtracting images. This will be illustrated for any steps where this becomes applicable. Additionally, for the second optimization, applying overlays of the binary images over the grayscale version allowed for visual inspection of the algorithms to determine any phase shifts. System level testing was done to inspect how the modified versions of the algorithm compared against the baseline. Generally, the baseline algorithm was taken as ground truth, thus if the modified either missed an April Tag while the baseline found it (false negative) or found an April Tag while the baseline did not (false positive) could be pinpointed. Static images were fed through each algorithm so they can be inspected thoroughly, saving any end to end testing until the very end.

4.5 Simulink Implementation

Hardware programming can be a tedious and often long process. Using any hardware description language (HDL) language, large projects often become incredibly confusing and complex to navigate when tracking down bugs. Simulink provides a block editor that not only makes it easier to navigate and manage large projects, but

also reduces the need to re-write boilerplate code. Simulink itself is a model-based design tool that provides simulations for quick prototyping prior to moving a design into hardware [28]. Following this application, Simulink was leveraged as the second tool in the design flow after Matlab to observe the various optimizations and approximations that would be implemented in hardware. After successful prototyping of the various algorithms in Matlab, the Simulink models were constructed first in isolation and then combined together. The complete Simulink model was also built in stages with each stage providing an output that can be fed back into the Matlab code to complete the April Tag detection. Once the simulations demonstrated a working system, HDL Coder was used to generate the intellectual property (IP) cores that would be loaded onto the Snickerdoodle board. HDL generation is a tool provided by Simulink to automatically generate the VHSIC hardware description language (VHDL) or Verilog code that becomes synthesized. With any generation tool, there can be some limitations and inefficiencies that has to be taken into consideration when designing. As a result, a majority of the blocks used were basic blocks for finer control and transparency on the end generated code and ultimately the end performance.

Chapter 5

Results

Simulation and testing was performed on the embedded Fusion 2 system as described in Section 4.1. To reiterate, all of the software algorithms ran on Zynq 7020 processor as embedded C/C++ code, leveraging Python to be the user interface (UI). The hardware optimizations ran on the Zynq 7020 embedded FPGA, leveraging Xilinx's toolchain to communicate between the FPGA and the processor. Python was also used to help configure, provide a graphical user interface (GUI), and collect results as depicted in Fig. 4.2. During simulation and testing, Fig. 5.1 was used as it provided a clean reference image that would easily highlight any bugs that may arise during the hardware algorithm development. All of the following sections will refer to Fig. 5.1 as each block was developed. This image was used to make it easy to find errors and bugs in the algorithm. Final results and testing was done on statically captured, real world images such as the one depicted in Fig. 5.53. Fig. 5.2 depicts the final Simulink block that was created after each sub component was connected. It was this block was put through the HDL Coder to generate the final binary file that would be loaded onto the FPGA and used in Section 5.5 to conduct benchmarks. The following sections will discuss the implementation results and Simulink image outputs. Section 5.1 will discuss the results for "Grayscale". Section 5.2 will discuss the results for "Normalize" and "Gaussian Smoothing". Section 5.3 will provide an in-depth discussion for the CORDIC atan2 results. Section 5.4 will show the final

Figure 5.1: Starting Image used for Testing.

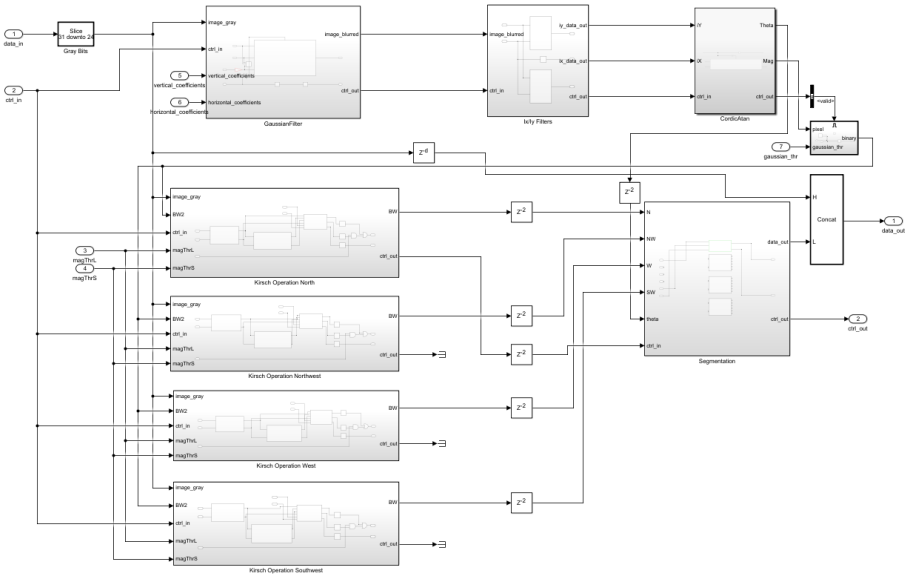


Figure 5.2: Final Simulink Block.

implementation of the CCA segmentation component.

As depicted in Fig. 5.2, the final block diagram has to 64-bit wide data output and a pixel control output. The 64-bit wide data output was summarized and can be found in Table 7.1 located in the Appendix.

5.1 Grayscale

As described before, the grayscale block was implemented by utilizing only shifts and adds to conserve on resources. Fig. 5.3 depicts the final block implementation in Simulink.

Upon further inspection, this algorithm reduced the total luminance of the image by a factor of 0.125. Considering the total luminance as 1, the sum of the factors

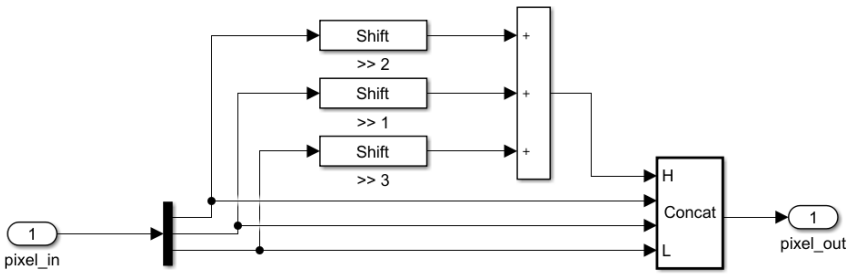
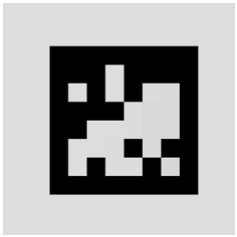
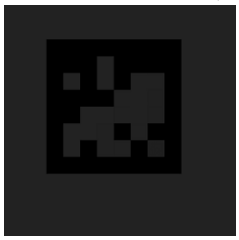


Figure 5.3: Grayscale Simulink Implementation.

(a) Grayscale Baseline.



(b) Grayscale Optimized Output.



(c) Grayscale Subtraction Output.

Figure 5.4: Grayscale Outputs.

applied to the RGB signal as depicted in Fig. 4.1 was 0.875 as depicted in (5.1).

$$L = \frac{1}{2^2}_R + \frac{1}{2^1}_G + \frac{1}{2^3}_B = \frac{1}{4} + \frac{1}{2} + \frac{1}{8} = 0.25 + 0.5 + 0.125 = 0.875 \tag{5.1}$$

As depicted in Fig. 5.4c, the approximate grayscale operation creates some minor error. The maximum error reported from the image was 34. Although the theoretical max was 32 $(255 - 255 * 0.875)$, 34 was observed because of rounding errors accumulated from each shift as the data remained a uint8 during the shifting process. Table 5.1 shows the generation report from the block depicted in Fig. 5.3.

This block was expected to take a very small amount of resources but additionally, Table 7.2 depicts that no multipliers were used but rather three static shifts and two

Table 5.1: Grayscale HDL Hardware Utilization Report Summary.

Resource	Usage	Percentage
LUT	7	0.01

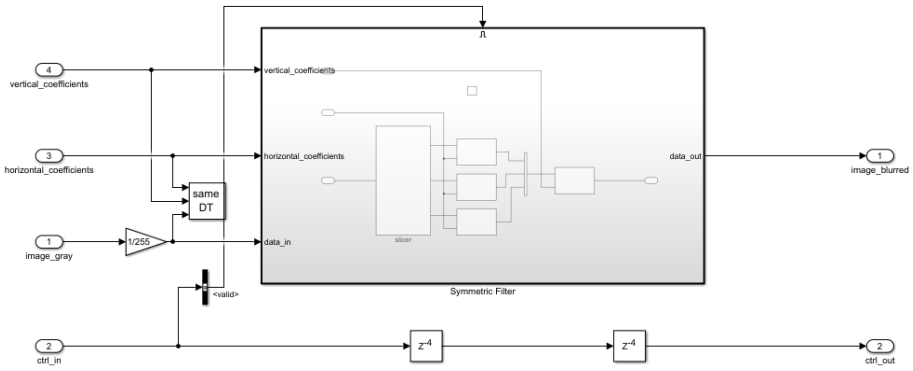


Figure 5.5: Gaussian Smoothing Simulink Implementation.

adders. This was translated to the seven LUTs seen in Table 5.1 that were used to implement the shifting logic. There were no timing reports associated with this block as there were no delays placed in-line with this block. Considering the simplicity of this block, it was not needed and would be added later in another block.

5.2 Gaussian Smoothing

The Gaussian smoothing kernel was the first block that introduced complexity regarding the control signal. Fig. 5.5 depicts the Simulink implementation of the Gaussian smoothing at a high level.

Although briefly mentioned, "Normalize" was also done inside the "Gaussian Smoothing" block as well. This will be discussed further in Section 5.4.1. During the Simulink implementation, the control signal had to be delayed to maintain synchronization with the input pixels and the output pixels. Misaligned control signals would cause undesirable phase shifts in the end image which may result in an increase in false negatives. Additionally, an enable signal was required as not every pixel passing through the system will be a valid pixel. As depicted in Fig. 7.1, the camera will

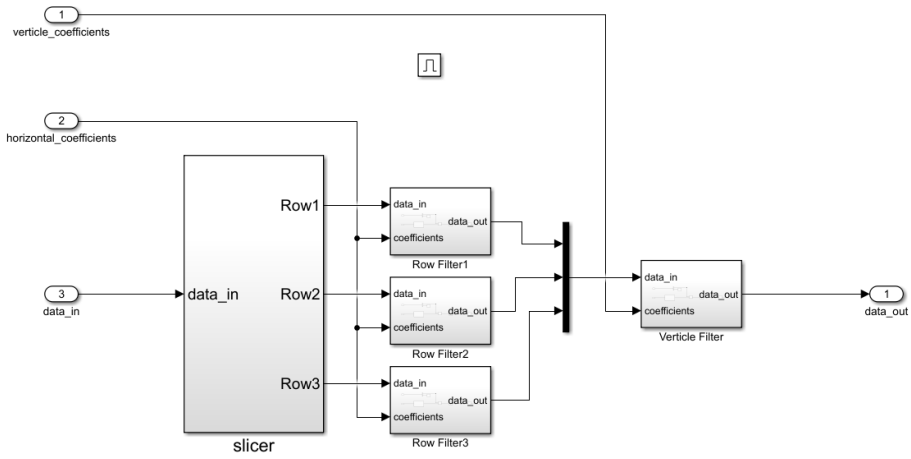


Figure 5.6: Gaussian Smoothing Symmetric Simulink Block Implementation.

also be reading in additional rows and columns it internally uses to calibrate itself which may or may not be read out on the line as well. For this application, they will not be enabled but future proofing the system was still done. As a result, this enable signal was peeled off the control bus to control when the Gaussian smoothing kernel would advance.

The design of this block was for overall portability and customizability given it doesn't impede on performance. The Gaussian constants were passed in from constant blocks external to the system but hold the values depicted in Fig. 4.8. For compatibility, a same data type block from Simulink was used to ensure the constants will be interpreted in the same manner as the input pixels. For this application, a fixed-point format of u,32.31 was used. This parameter was configurable by the Simulink model's initialization script.

Depicted in Fig. 5.6 was the implemented separable filter which leveraged the Gaussian kernel's symmetry. The slicer block contains the logic of creating a 3x3 kernel. It outputs the data by rows for simplicity as the kernel performs the arithmetic on each row rather than on each cell. The first filter to be applied were the row filters that took in the same horizontal filter constant. From there, the single value outputs were combined and then fed through the vertical filter and its corresponding constant.

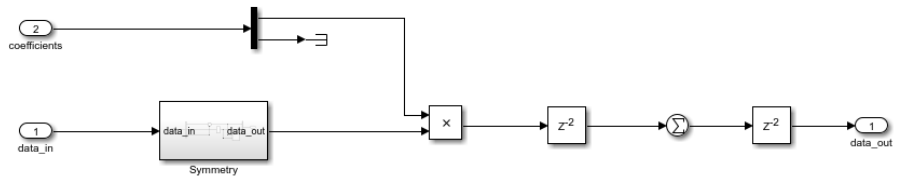


Figure 5.7: Gaussian Smoothing Row and Vertical Filter Simulink Block Implementation.

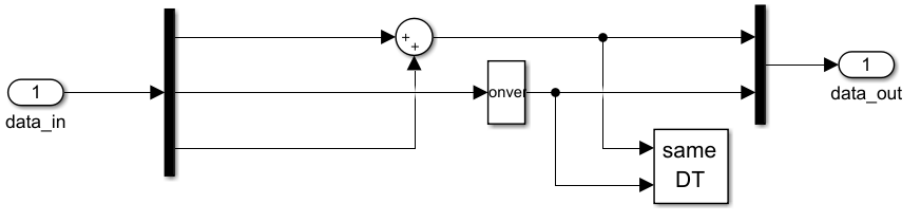


Figure 5.8: Gaussian Smoothing Symmetry Simulink Block Implementation.

For a Gaussian smoothing filter that was balanced, the constants were the same but were left separated here for re-usability.

As shown in Fig. 5.7, the row and vertical filters contain a symmetry block, a multiplier and a summation. One of the coefficients were ignored because it would be handled inside the symmetry block through factorization. The multiplier was a element-wise multiplier, combining the coefficient with the respective pixel value. Delays were introduced here to meet timing and provide signal stability.

As mentioned before, the factorization was handled by summing the pixels that would have the same coefficients multiplied to them. This optimization saved several multiplications as defined in Section 4.3.2. Additionally, a data type conversion block with the same data type block was leveraged here to keep the precision between the two signals the same. This adds an unnecessary bit to the unmodified signal but was necessary as Simulink cannot handle mixed types when recombining signals onto a single wire. Table 5.2 shows the resources used to implement this block.

Unlike what has been defined in Section 4.3.2, Table 7.3 reports that only a single multiplier was saved. This came from the simple buffering implementation that gave the output as 3x3 kernel instead of a single column at a time which required

Table 5.2: Gaussian Smoothing HDL Hardware Utilization Report Summary.

Resource	Usage	Percentage
LUT	779	1.46
LUTrandom access memory (RAM)	69	0.4
flip flop (FF)	1867	1.75
BRAM	2	1.43
digital signal processing (DSP)	32	14.55

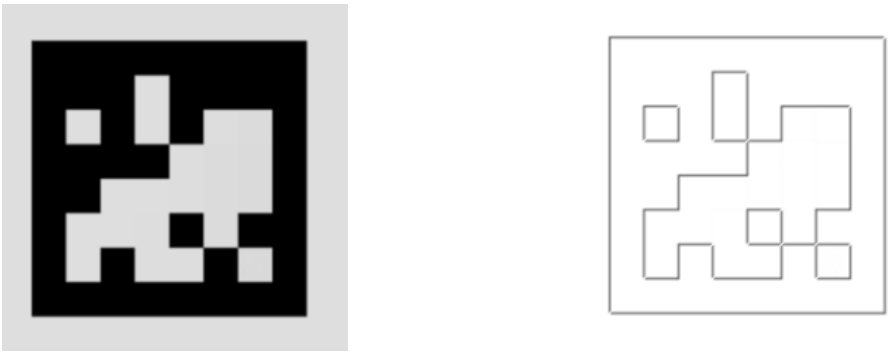
Table 5.3: Gaussian Smoothing HDL Hardware Timing Report Summary.

Target Frequency: 27 MHz	Setup (ns)	Hold (ns)	Pulse Width (ns)
Worst Negative Slack	29.153	0.097	17.538
Total Negative Slack	0.000	0.000	0.000

the redundancy of resources that could be saved with a complex buffering system. Although seemingly insignificant, this type of savings becomes incredibly helpful in larger designs and with larger filters. Overall, the resources consumed by this filter was minimal. Most of the advanced computations were put into DSPs, taking up a significant portion of the available space. For the buffering, a BRAM was used along with many FFs.

As depicted in Table 5.3, the Gaussian smoothing kernel met the timing requirement given by the camera. Overall, the kernel took very little time to compute a single pixel and was sufficiently pipelined. The Total Negative Slack indicates that the hardware met timing as well as the positive values for Worst Negative Slack. Fig. 5.9 depicts the final Simulink output of this step and its verification image.

Fig. 5.9b was generated using the grayscale image depicted in Fig. 5.4b as the base for a Matlab implementation of the "Gaussian Smoothing" versus the Simulink computation. Although briefly mentioned before, Fig. 5.9b clearly illustrates the potential for phase shifts in the final output image compared to the base, gray image depicted in Fig. 5.4b. Upon correctly shifting the image, the subtraction image came out entirely black, indicating the offset was corrected. Future subtraction images will be depicted with this phase shift correctly handled.



(a) Gaussian Smoothing Simulink Image. **(b)** Gaussian Smoothing Subtraction Image.

Figure 5.9: Gaussian Smoothing Simulink Output Image and Verification.

5.3 Gradient Magnitude & Direction

The "Gaussian Smoothing" block was split into two different subsystems. The first subsystem managed the two symmetric, gradient filters, Ix and Iy. The second subsystem performed the CORDIC operation to produce the end result of magnitude and direction.

5.3.1 Differential Gradient Implementation

Fig. 5.10 depicts the high level Simulink block of the gradient filters. Similarly to the "Gaussian Smoothing", these blocks also used an enable signal from the control bus as well as delaying it. As illustrated in Fig. 4.5, both the Ix and Iy filters can be computed in parallel compared to software where they have to be done sequentially.

Although the gradient differential blocks were not separable as their rank was not 1, they still could be optimized as most of the pixel values were ignored. As depicted in Fig. 5.11, the pixels of interest can simply be subtracted from each other, ignoring the other pixels entirely. The slicer block in this implementation was identical to the "Gradient Magnitude and Direction" in Fig. 5.6 although the Ix version of this block does not need to care about other rows other than the current one. This design choice was done to maintain simplicity and clarity. This step was not introduced in

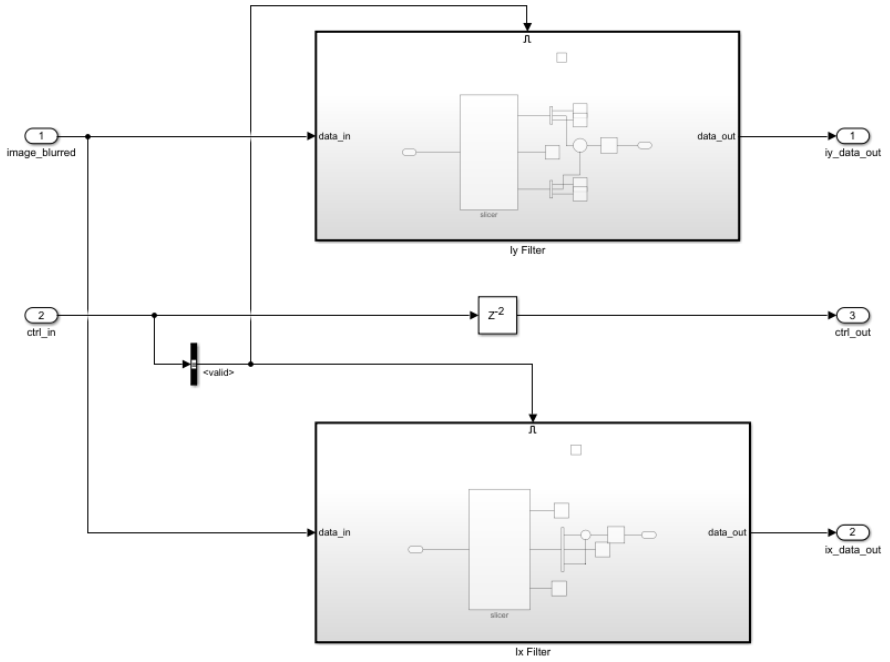


Figure 5.10: Differential Gradient Ix and Iy Simulink Implementation.

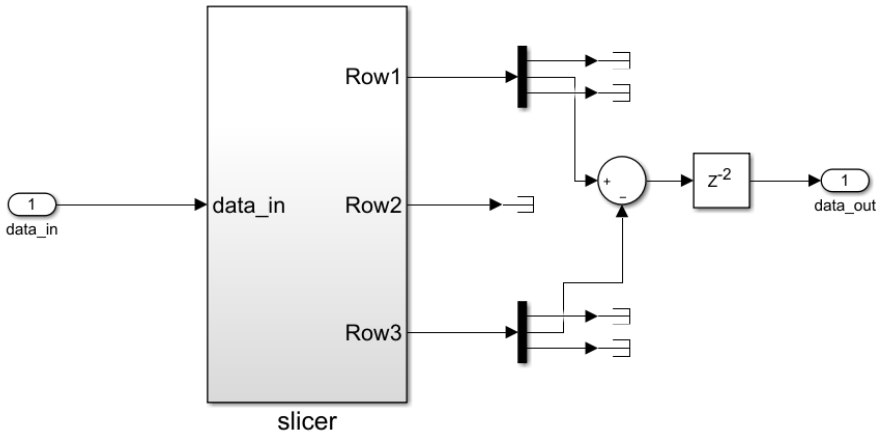


Figure 5.11: Differential Gradient Simulink Implementation.

Table 5.4: Gradient difference HDL Generation Report Summary.

Resource	Utilization	Percentage
LUT	196	0.37
LUTRAM	16	0.09
FF	560	0.53
BRAM	3	2.14

Table 5.5: Gradient difference HDL Timing Report Summary.

Target Frequency: 27 MHz	Setup (ns)	Hold (ns)	Pulse Width (ns)
Worst Negative Slack	33.277	0.07	17.538
Total Negative Slack	0.000	0.000	0.000

Table 4.2 as it was a sub-step for "Gradient Magnitude and Direction". The output for this section was between -1 and 1 as "Normalize" was done within "Gaussian Smoothing".

As shown in Table 5.4, the gradient block took minimal resources as the only computation it did was a subtraction. There was minimal complexity to this system.

5.3.2 CORDIC Implementation

The CORDIC implementation was far more in-depth than the first section to ensure the algorithms accuracy. several steps were taken to ensure that the algorithm worked long before its implementation into Simulink and the hardware. Each step had its own verification process to ensure any bugs did not propagate to the next step and were caught as early as possible. The following subsections breakdown the CORDIC implementation into three parts. Section 5.3.2.1 introduces the Matlab implementation, testing and verification of the CORDIC algorithm. Section 5.3.2.2 discusses how the algorithm was translated into Simulink. Section 5.3.2.3 shows the hardware results of the Simulink block.

```
% Special cases
if ix == 0 && iy == 0
    mag = 0;
    angle = 0;
    return;
end
```

Figure 5.12: Atan2 CORDIC Special Cases.

5.3.2.1 Matlab CORDIC Implementation

The first step to this process was implementing and verifying the algorithm in Matlab. The actual implementation of the Atan2 CORDIC algorithm was simple, having four distinct sections. The first section were the "Special Cases" of the algorithm.

As depicted in Fig. 5.12, the only necessary special case was when both inputs were zero. The CORDIC algorithm would incorrectly estimate the angle and magnitude, returning a number close to π for the angle. This is because atan2 is undefined when the inputs are both zero. By default, undefined behavior should return zero which was mirrored by both Matlab and GNU's implementation of atan2. This was important to maintain the same behavior as the C implementation of the April Tag algorithm. Other special cases can be included and were described in (2.1). The special cases with defined behavior were omitted because this algorithm would eventually be implemented in hardware. Having a simpler pipeline with a few exceptions will have a smaller footprint on hardware resources. Additionally, these optimizations can only be beneficial in software where execution time can be dynamic. In hardware, this block will take the same exact amount of time, regardless of the input. The second section of the CORDIC algorithm was the setup. Fig. 5.13 depicts the first few starting computations of the algorithm.

The setup function illustrates an additional helper function, zsign, as depicted in Fig. 5.14 which computes the sign of the value while treating zero as positive.

This helper function was required to simplify the process to flipping the sign of

```
%% Setup
% The next x value
xo = iy*zsign(iy);

% The next y value
yo = ix*-zsign(iy);

% The Atan
zo = -zsign(iy) * cordicLut(1);
z = zo + cordicLut(2)*-zsign(yo);
zo = z;
```

Figure 5.13: Atan2 CORDIC Setup Code.

```
%% Helper function to treat 0 as positive
function [s] = zsign(v)
    if v < 0
        s = -1;
    else
        s = 1;
    end
end
```

Figure 5.14: Atan2 CORDIC Zsign Helper Function.

```
%% Iterations
for iter = 3:12
    x = xo + bitsra(yo, iter-3) * zsign(yo);
    y = yo + bitsra(xo, iter-3) * -zsign(yo);
    z = zo + -zsign(y) * cordicLut(iter);
    zo = z;
    yo = y;
    xo = x;
end
```

Figure 5.15: Atan2 CORDIC Iterative Code.

a value. As shown in Fig. 5.13, the function was being multiplied into a value to flip the sign. The complexity of imitating a hardware algorithm in software was the way zero and negative numbers were treated. In binary, there was no negative bit, and multiplying by negative one can be simulated as negating the bits and adding one. In software, especially with floating point precision, negating a number requires only flipping the sign bit. Regardless, Fig. 5.13 illustrates the four primary computations required to start the CORDIC algorithm, two of which advances the angle computation with two steps.

The third section of this algorithm was the iterative part. Fig. 5.15 illustrates the primary component of the algorithm that was pipelined. Each of the three computations for x, y, and z can be computed in parallel. This code depicts the algorithm being hardcoded with ten additional iterations making the CORDIC algorithm a depth of twelve. The code mimics the computations described in (2.3).

The fourth section of this algorithm was the optional massaging of the data so it would be ready for use in the CPU. Fig. 5.16 depicts the magnitude as needing to be scaled by K and the angle needing to be inverted. This portion of the code illustrates the final iterations converging onto the values described in (2.3). The primary purpose of performing these operations in hardware was to reduce any overhead required by the CPU. Although the multiplication adds quantization errors as depicted in Table 5.6

```
%% Data fixing
mag = x * K;
angle = -z;
```

Figure 5.16: Atan2 CORDIC Data Fixing Computations.

in hardware fixed point, the speed up gained was significant enough to keep it.

The verification process for the algorithm was an exhaustive test, providing stimuli of all possible inputs to the algorithm. It was anticipated that the atan2 CORDIC algorithm would only be provided an input from -1 to 1 as described in the differential gradients in Section 5.3.1. The atan2 CORDIC algorithm was verified against Matlab's built-in atan2 function for angle comparison and the magnitude computation as described in (2.1) and (4.3) respectively. From there, the percent error formula depicted in (5.2) was computed for the magnitude analysis. This provided the relative error between the two.

$$\frac{|actual - error|}{actual} \tag{5.2}$$

The angle comparison could not utilize any traditional relative error algorithms as the actual answer from the atan2 function can be zero. Having zero as an actual answer for any of the relative error algorithms pushes them two the bounds, even if the error is small. As a result, (5.3) was used instead.

$$|actual - error| \tag{5.3}$$

Although (5.3) does not provide any percentage, a minimal difference between Matlab's atan2 function and the atan2 CORDIC algorithm implemented still needs to be small. These results were summarized in Table 5.6.

As depicted in Table 5.6, the error between the baseline functions and the CORDIC approximations are minuscule when dealing with double floating-point precision. This

Table 5.6: Analysis Results of Baseline vs Atan2 CORDIC.

Category	Angle	Magnitude
MSR	5.594433048696700e-04	1.057732796575861e-06
Max Error	9.7623e-04	2.4354e-06
Min Error	0	5.2876e-07

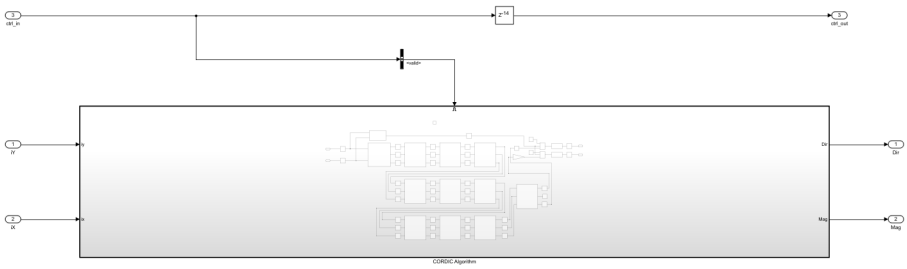


Figure 5.17: High Level CORDIC Simulink Block Implementation.

algorithm was not tested within the scope of fixed point as it would be more time consuming to do it in Matlab than in Simulink. Simulink also provides the possibility of seeing the expansion of the fixed-point numbers of in-between steps for further optimization if necessary.

5.3.2.2 Simulink CORDIC Implementation

This section will go into detail about taking the Matlab Atan2 CORDIC algorithm and creating a hardware version of it. One of the key benefits of taking this algorithm and pushing it through hardware was to take advantage of parallelism between stages as shown in Fig. 5.15 as all three operations can be done simultaneously.

The four sections described in Section 5.3.2.1 was broken down and illustrated for the future purpose of explaining the rationale behind the various subsystems implemented in Simulink. The entire system can be seen in Fig. 5.18. The initial version of the model left everything as double floating-point precision. This was to ensure the structure of the model was working before complicating it with any fixed-point changes. Fig. 5.17 illustrates the high level Simulink implementation of the CORDIC algorithm and was included for completion.

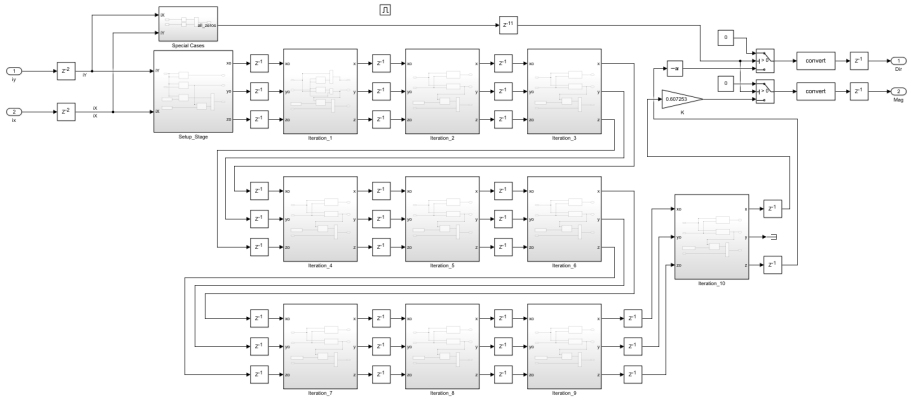


Figure 5.18: CORDIC Simulink Block Implementation.

It shows the same valid signal being peeled off to control when the CORDIC pipeline should advance or not. Fig. 5.18 depicts the entire CORDIC algorithm.

To break down the model, there were two distinct paths: a special cases path and the CORDIC algorithm path. As mentioned before, the special cases path covered the undefined behavior that atan2 algorithm exhibited when both inputs were zero. This path was selected when that case is true, thus setting the output to zero when necessary. Referring to Matlab, the "Setup", "Iterative loop", and "Data Fixing" portions were the standard steps. Each step was pipelined as depicted by the $z^{-}1$ delay blocks. Each subsystem should be considered a cloud of logic when viewing it in terms of a hardware system. Though not in its own subsystem, the "Data Fixing" portion was the final gain labeled K and the unary inversion prior to a multiplexer which combines both signals onto a single bus. In total, there were 12 iterations that were used to implement the CORDIC algorithm as described in Section 5.3.2.1. Two of those iterations were completed in the "Setup" stage of the algorithm. The other ten can be seen as individual blocks pipelined together.

As depicted in Fig. 5.19, the only special case utilized in this design was the input condition where the CORDIC algorithm would fail, when Ix and Iy were zero. More complicated logic could be implemented to facilitate additional paths for the other special cases mentioned in (2.1) but simplicity was preferred.

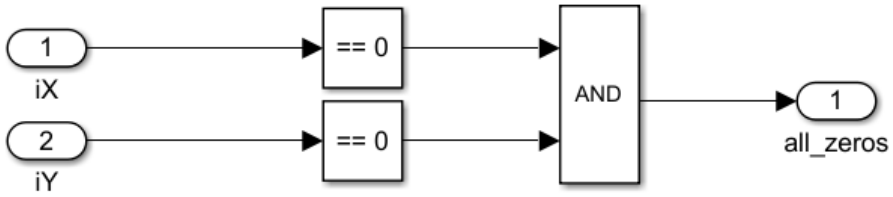


Figure 5.19: CORDIC Special Cases Simulink Block Implementation.

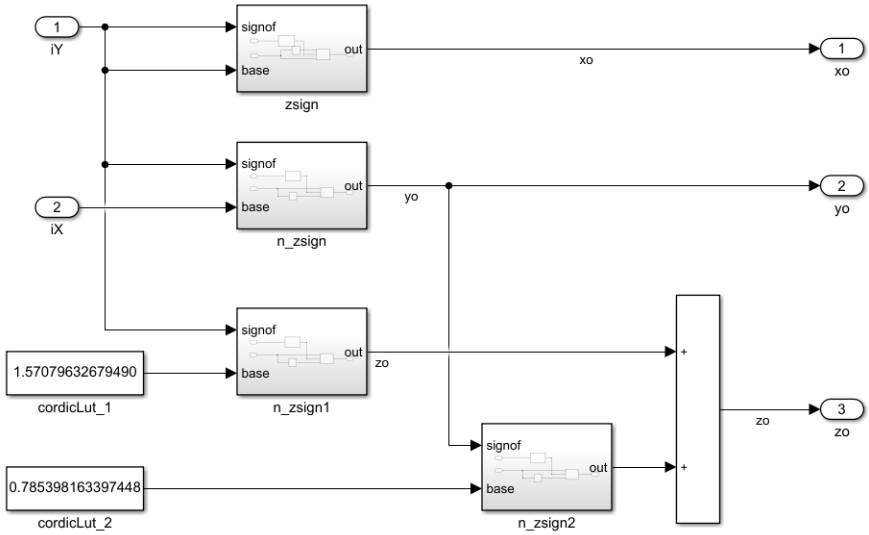


Figure 5.20: CORDIC Setup Simulink Block Implementation.

(a) zsign **(b)** Negated zsign

Figure 5.21: Zsign Helper Function Logic Clouds.

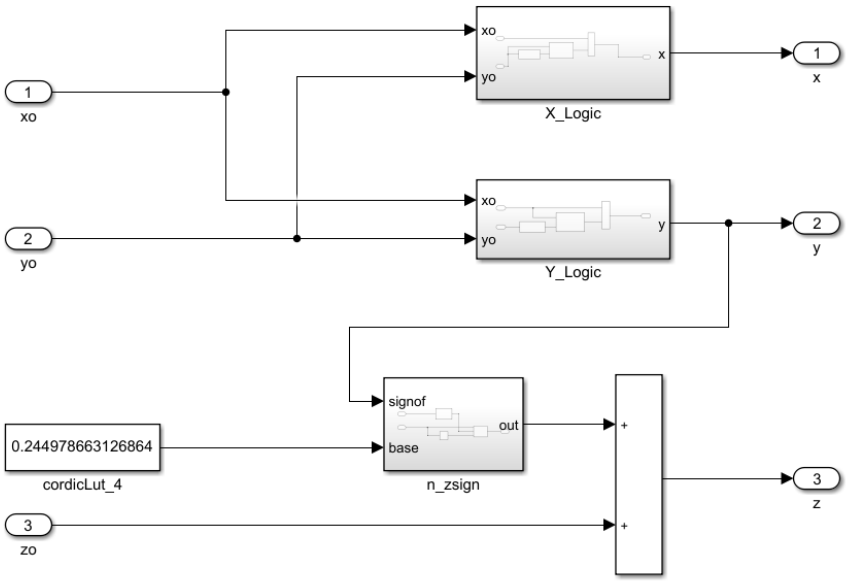


Figure 5.22: CORDIC Iteration Simulink Block Implementation.

The inner portions of the model were simpler than their code counter parts. Most of the complexity lies within the zsign function as depicted in Fig. 5.14. Due to the advantage of being in hardware, slicing the bits and inverting them no longer required a multiplication. Fig. 5.21a and Fig. 5.21b depict zsign and its inverted counterpart, respectively.

The Simulink model illustrates that the zsign function becomes nothing more than a multiplexer choosing the positive or negative version of the number, using the most significant bit (MSB) of the number. In this case, a slice of '15 downto 15' was used since the anticipated word length was 16. This was parametrized as well if expansion was necessary. The following figures breaks into each iteration block, all mimicking the code depicted Fig. 5.15.

(a) X Logic Cloud. **(b)** Y Logic Cloud.

Figure 5.23: X and Y Logic Cloud Simulink Implementation.

Table 5.7: Analysis Results of Baseline vs Atan2 CORDIC in Simulink.

Category	Angle	Magnitude
MSR	5.8047e-04	1.1050e-04
Max Error	1.1891e-03	5.0516e-02
Min Error	7.9911e-07	0
Quantization	1.2207e-04	

Fig. 5.22 illustrates typically how each iteration block is organized. With each iteration, there was a different cordicLUT constant applied and a different arithmetic shift right length where 'w' was included to indicate that the value of 'w' was the iteration block number minus one. The first iteration block omits the arithmetic shift entirely as it would be a shift of zero, indicating to do nothing. The primary idea behind these logic clouds was to have as little logic between each pipelined stage as possible. This will maximize the frequency the FPGA can move through each pipelined stage, and evidently reduce the potential chance for error between stages.

Table 5.7 illustrates similar statistics to Table 5.6, with a small indication of the quantization error, the minimum step between two different numbers for the fixed point number. This was done comparing the double precision float-point input values going into the baseline functions vs the signed, Q16.13 values going into the CORDIC block. Fig. 5.24 illustrates the basic blocks that were tested against the CORDIC implementation.

Since Simulink does not have the ability to iterate over two numbers easily, the verification process was split between the magnitude and the angle. The magnitude data was set by simply taking the magnitude of the same number. This was to see how the calculation would work over the entire range. The angle data was a bit more

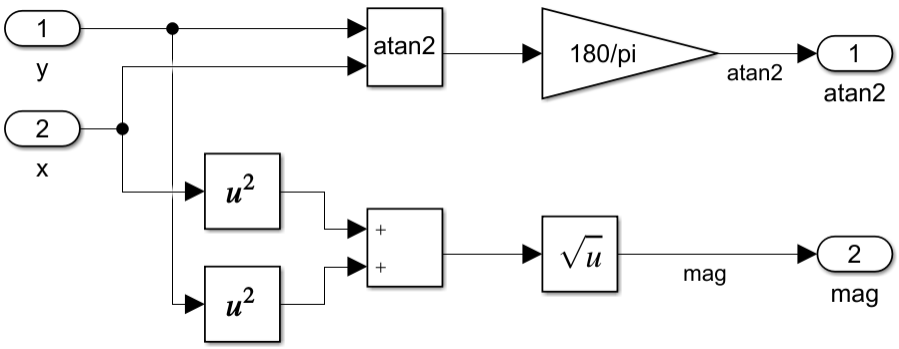


Figure 5.24: Baseline Simulink Block Implementation.

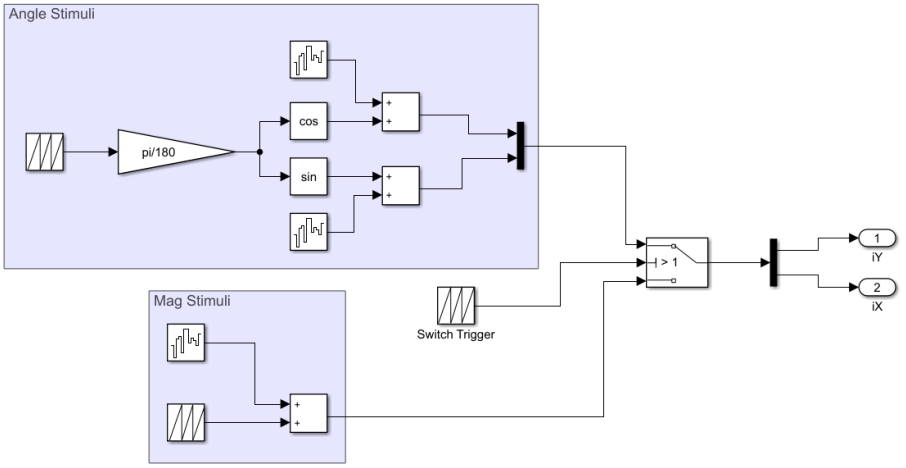


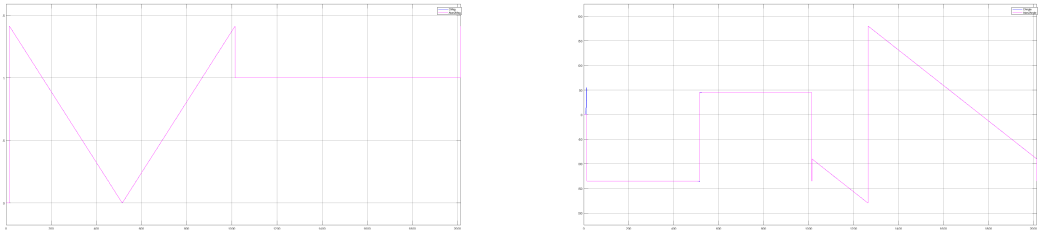
Figure 5.25: Simulink Testing Stimuli.

complex, taking a single input and computing both the sin and cos of that input as using those as the Iy and Ix values. Fig. 5.25 depicts the stimuli described.

After iterating over a thousand points of equal steps, the switch function was used to toggle between the angle stimuli and the magnitude stimuli to run the next thousand points. Fig. 5.26 depicts the scoped outputs after running over the whole range.

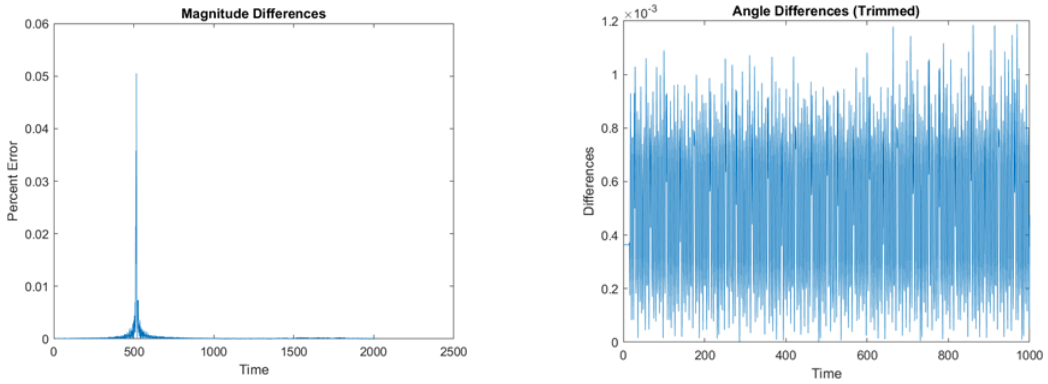
Both the magnitude and the angle depicted in Fig. 5.26a and Fig. 5.26b as blue lines stuck very close to the baseline implementation shown in magenta. Clearer results can be seen in Fig. 5.27.

Fig. 5.27a, the magnitude had a slight spike in the error around the double value of 0.03. After some digging, it became apparent that 0.03 cannot be represented



(a) Magnitude Scope. **(b)** Angle Scope.

Figure 5.26: Scoped Outputs Comparing the Baseline and CORDIC Implementations.

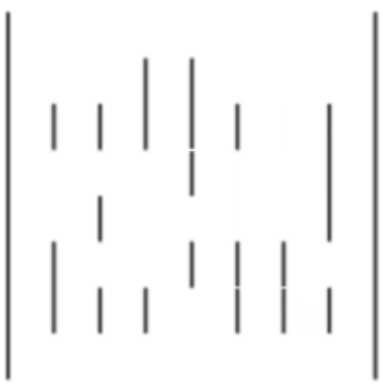


(a) Magnitude Percent Error vs Time. **(b)** Angle Differences vs Time (Trimmed).

Figure 5.27: Magnitude and Angle Errors Plotted against Time.

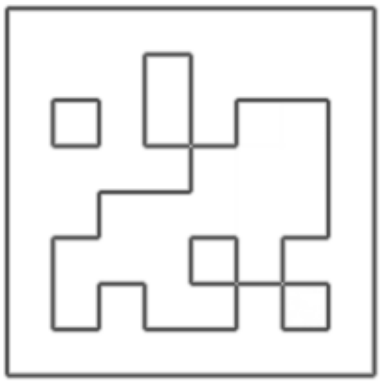
accurately as a signed Q16.13. The fixed-point approximation of 0.03 was 0.0299, thus creating that quantization error. Removing those outliers, most of the data remains below 1% error. Fig. 5.27b, the angle data also had a similar spike during the magnitude test but was omitted during the analysis of the angle data. This was because the angle output drifted as the inputs moved through the pipeline. As mentioned before, the angle data cannot be put through a percent error or any relative error algorithm since some of the accepted values can be zero. As a result, only the differences were illustrated in Fig. 5.27b. Here, the difference between the baseline and the CORDIC implementation remains minuscule, hovering less than 9 times the Q16.13 epsilon value. This was the most ideal case, illustrating that the CORDIC algorithm mostly matched the atan2 operation even with the fixed point quantization. Fig. 5.28 depicts the final output images given the test image displayed in Fig. 5.1.

Fig. 5.28a and Fig. 5.28b illustrate the output portions of the gradient differential

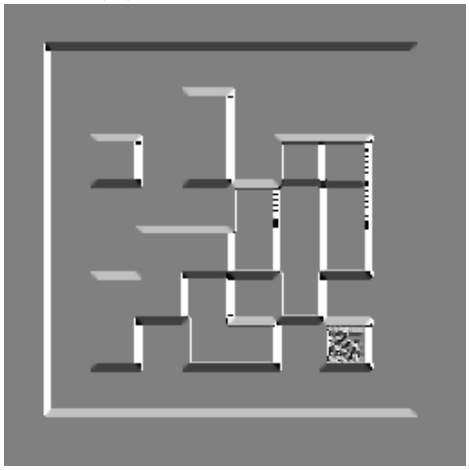


(a) Ix image (negative)

(b) Iy image (negative)



(c) Magnitude image (negative)



(d) Directional image

Figure 5.28: Gradient Magnitude and Direction Output Images from Each Step.

block that was fed into the CORDIC algorithm. These images were displayed inverted, where white was zero and black was 1. Additionally, the image depicts the absolute value of the block to also show negative numbers. Fig. 5.28c depicts the CORDIC algorithm's ability to reconstruct the original tag, outlining only the edges. Fig. 5.28d shows the direction or angles of the corresponding pixels. Since the value of this image was between $-\pi$ and π, the gray indicates an angle of zero. This means the pixel value stayed the same and there was no transition. It can also indicate a transition from black to white horizontally as the concept of -0 was difficult to implement in hardware which would make the output of atan2 $-\pi$ in the case of $ix = -1$. High pixel values, white, represent a shift from white to black horizontally which was near a value of π. A light gray pixel value was a shift from black to white vertically, having a value near $\frac{\pi}{2}$. Dark gray pixel values were a shift from white to black vertically, having a value near $-\frac{\pi}{2}$.

Upon inspecting Fig. 5.28d further, the various noise seen in the image was attributed to the sensitivity of the CORDIC algorithm to minor changes in pixel values. The operation of arctan does not depend on the actual value of the pixels but rather the ratio between them. This means pixels with a difference of 1, 10, and 100 can have the same angle output if their ratios remain the same. As depicted in Fig. 5.28d, the error manifests as noise which was typically rejected later in the process as these pixels had very low magnitudes as depicted in Fig. 5.28c.

5.3.2.3 Hardware CORDIC Implementation

The hardware implementation of the CORDIC algorithm was done directly through the HDL Coder interface provided by Simulink. For verification testing, there were small tweaks done to the generated code as it was unable to synthesize for testbenches due to conflicting signal names during compilation. Table 5.8 shows the timing report done during generation.

Table 5.8: CORDIC HDL Coder Timing Report Summary.

Target Frequency: 27 MHz	Setup (ns)	Hold (ns)	Pulse Width (ns)
Worst Negative Slack	28.053	0.150	17.538
Total Negative Slack	0	0	0

Table 5.9: CORDIC HDL Generation Report Summary.

Resource	Utilization	Available	Utilization %
LUT	855	53200	1.61
LUTRAM	12	17400	0.07
FF	732	106400	0.69
DSP	4	220	1.82

Table 5.8 illustrates the timing report summary compared against the minimum frequency of 27 MHz. This frequency was chosen as the baseline because the camera being used will operate with 27 MHz per pixel. Refer back to Table 2.2 for additional details. Since this algorithm must work per pixel, it must operate at least 27 MHz or, ideally, faster.

Table 5.9 illustrates the utilization report summary of the atan2 algorithm as synthesized by the HDL Coder output. Overall, the CORDIC algorithm does not take a lot of resources but can certainly be optimized more within the Simulink portion of the design. One potential optimization would be to maintain the signed Q16.13 fixed point notation throughout the entire implementation which will drastically reduce the LUT and FF utilization. The downside of this effort may introduce higher quantization errors aside from 0.03 as discussed before.

To verify the accuracy of the generated HDL code, a post functional simulation was conducted at each step of the process. Only the post functional simulation of the implementation will be shown. The simulations were done with a 20 ns clock period. The data for the testbench was read in through files generated from the Simulink model.

As depicted in Fig. 5.29, this testbench was also exhaustive, starting from the minimum of a signed Q16.13 fixed point number, -4, to the upper bound of 3.99999.

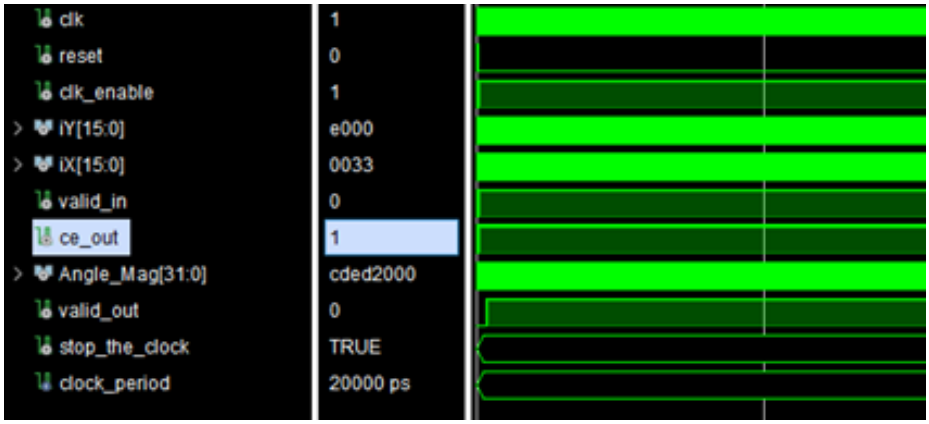


Figure 5.29: CORDIC Testbench Waveforms.

Here, the outputs were compared to the outputs in the Simulink model, reporting if any discrepancies occurred. From this testbench, there were no errors reported, signifying the hardware implementation matches the Simulink model completely. It was also important to note that while the Simulink model was designed to have a pipeline of 14 stages, the HDL Coder increased it to 16 stages. This was most likely due to any pipeline matching or to meet timing.

5.4 Connected Component Analysis

The connected component analysis segmentation block was a much larger and more complicated block to implement compared to the three previous blocks. Similarly to "Gradient Magnitude and Direction", this block was also split into three primary components. Section 5.4.1 introduces the Kirsch filter implementation. Section 5.4.2 discusses how the Kirsch filters were binarized and prepared for the last section. Section 5.4.3 goes into detail on how the CCA algorithm was implemented in hardware, how the feature extraction process worked differently from a software implementation, and how the data was sent to the software pipeline.

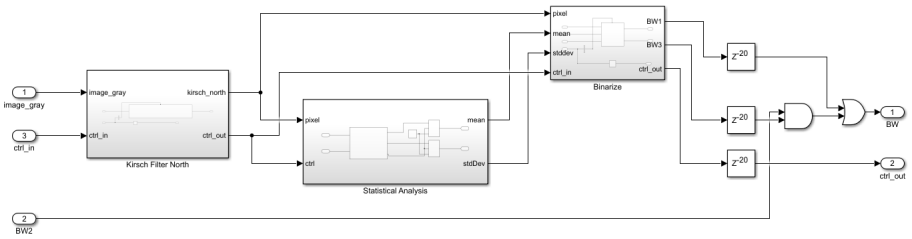


Figure 5.30: High Level Pre-Processing Block.

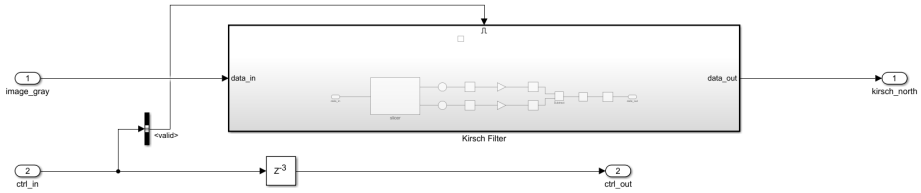


Figure 5.31: High Level Kirsch Filter Block.

5.4.1 Kirsch Filter

The Kirsch operator lent itself for easy optimization in the hardware considering it used only two coefficients as described in Fig. 4.12. Similarly to the separable filter optimization done for "Gaussian Smoothing" and symmetric optimization for "Gradient Magnitude and Direction", the Kirsch filter's coefficients were factorized, adding pixels with the same coefficients.

Depicted in Fig. 5.30 was the final design block that incorporated the Kirsch filters and the binarization. These were combined into a single subsystem as this can be considered the pre-processing steps in the CCA pipeline. As shown, the pre-processing block utilized two different inputs which were valid at different points. For the Kirsch filter, it needed the grayscale image while the binarization block needed both the Kirsch operated image and 'BW2'. In this block, 'BW2' was the binarized magnitude. The 'BW2', image statistics, and binarization blocks will be discussed in depth in Section 5.4.2. Fig. 5.31 depicts the typical high level block for all the subsystems in this application.

As Depicted in Fig. 5.32, the two coefficients were specifically pulled out of the

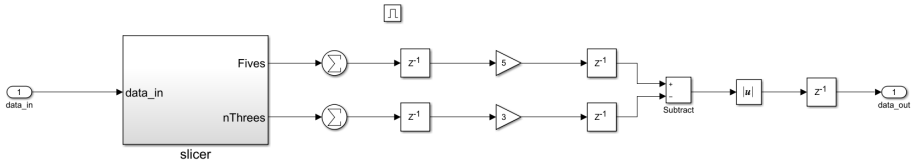


Figure 5.32: Kirsch Filter Block.

Table 5.10: Kirsch Filter HDL Coder Utilization Report Summary.

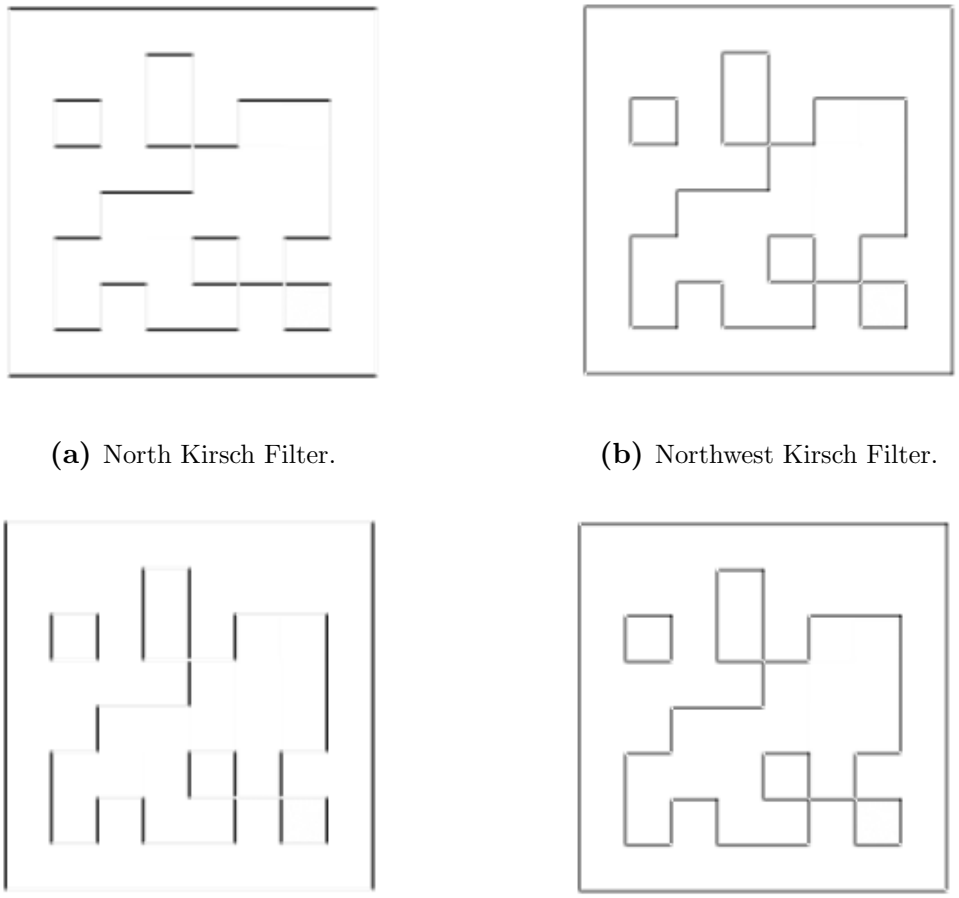
Resources	Utilization	Percentage
LUT	118	0.22
LUTRAM	4	0.02
FF	230	0.22
BRAM	1	0.71
DSP	2	0.91

slicer block, summed, and then multiplied by their respective coefficients. In total, this reduced the number of multiplications to 2 down from 9. Additionally, the "threes" were multiplied by a positive three to prevent the need for a signed bit until the subtraction. In the end, this sign bit was dropped as only the magnitude was required for the final output.

The hardware utilization for a single Kirsch filter can be found in Table 5.10. This design was replicated four times for each Kirsch filter compass direction, North, Northwest, West, and Southwest. As a result, the utilization of the filter was expected to be the same across each filter. As depicted in Table 7.10, the Kirsch filter implementation was able to significantly reduce the number of multipliers from its classical implementation from 9 down to 2. Table 5.10 illustrates the minimal resources required to implement this filter. Similarly to "Gaussian Smoothing", the Kirsch filter required a BRAM to buffer the incoming data and DSPs to perform the computations but at a much less overall utilization.

Table 5.11: Kirsch Filter HDL Timing Report Summary.

Target Frequency: 27 MHz	Setup (ns)	Hold (ns)	Pulse Width (ns)
Worst Negative Slack	31.100	0.094	17.538
Total Negative Slack	0.000	0.000	0.000



(a) North Kirsch Filter.

(b) Northwest Kirsch Filter.

(c) West Kirsch Filter.

(d) Southwest Kirsch Filter.

Figure 5.33: Kirsch Filter Simulink Output Images.

With the highly simplistic logic, Table 5.11 illustrates that the Kirsch filter implementation also met timing. There were no complex paths or logic that changed the output. With only a few multipliers and adders, there was minimal delay that could be introduced. Given the four other filters, this was expected to remain the same as each filter operates in parallel with each other.

The final Simulink model outputs were depicted in Fig. 5.33. As mentioned before, these filters were the magnitude of the compass direction. Since the Kirsch filter outputs were between 0 and 3825, the images depicted were normalized between 0 and 1 as well as inverted. One thing to note was the non-linear nature of the Kirsch

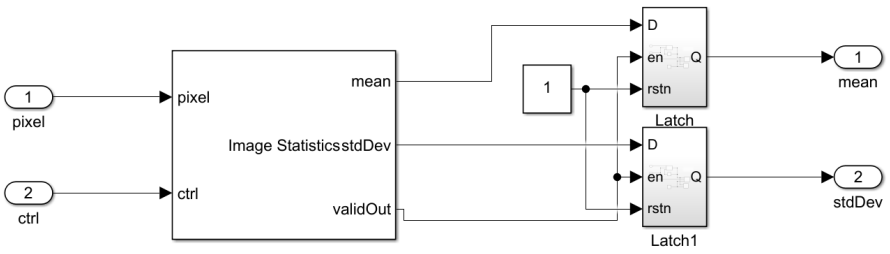


Figure 5.34: Binarization image statistics block

operator. As depicted in Fig. 5.33a and Fig. 5.33c, the image shows faint lines in compass directions that should not have been captured. The North compass filter should ignore any transitions that were horizontal and vice versa, similar to how the gradient differences behaved. Section 5.4.2 will discuss how this non-linearity was handled.

5.4.2 Binarization

The binarization of the Kirsch filter image outputs and the magnitude provided by the CORDIC block were essential steps when implementing the CCA Segmentation algorithm. As discussed in Section 2.4, strictly limiting the output to a binary image greatly simplifies the CCA operations. This portion also allows for the two output images to be combined into a single one to perform the operation. To understand, the image statistics block depicted in Fig. 5.34 will be explored first.

As depicted in Fig. 5.34, Simulink's built-in "Image Statistics" block was leveraged to calculate the mean and standard deviation of the Kirsch filter image [29]. This block implements the mean and standard deviation by utilizing summed-area tables. They were introduced by Crow to quickly compute statistics on blocks of an image [30]. These images contain pixel values that were the sum of all the pixel values up and to the left of them. The total pixel value can be summarized in (5.4) where 'TL' was the top-left pixel, 'BR' was the bottom-right pixel, 'TR' was the top-right pixel,

and 'BL' was the bottom-left pixel.

$$T = TL + BR - TR - BL \tag{5.4}$$

Following the same idea, Simulink's "Image Statistics" block implemented the mean as a combination of 64 pixel windows followed by several depths of that window [31]. The mean for the first window was first summed and then propagated to the other levels which summed those means given the frame's size. At the end of this chain, it would be multiplied by a value that would normalize any pixels that were not evenly divided by 64, 64^2, or 64^3. These constants were held in a LUT that would optimally into a DSP block on an FPGA. The normalization process incurred errors for frames outside of the dividers mentioned. Simulink provided an error graph for the error due to this approximation versus the number of pixels in the frame. It had a maximum error of 0.19. Additionally, the squared version of the summed-area table was calculated to quickly calculate the variance. The standard deviation was calculated by computing the square root using additions and shifts rather than multipliers.

For this application, Simulink's image statistic block had the limitation of being able to perform its operation on fixed-point data types that included fractional bits. This created an error for the mean and standard deviation by quantizing them to whole numbers. Under this expectation, the Kirsch filters were implemented using the grayscale image prior to normalization. This would limit the effect of the quantization. Additionally, it was unknown if the block would maintain the previous output while calculating the frame's statistic so a simple latch was implemented.

As depicted in Fig. 5.35 and Fig. 5.36, this block also required an enable signal to handle pixels outside of the active frame. Fig. 5.36 depicts how the mean and standard deviation were used to compute the two binarized images, 'BW1' and 'BW3'. These two different thresholds were used to produce a single binarized image that isolated

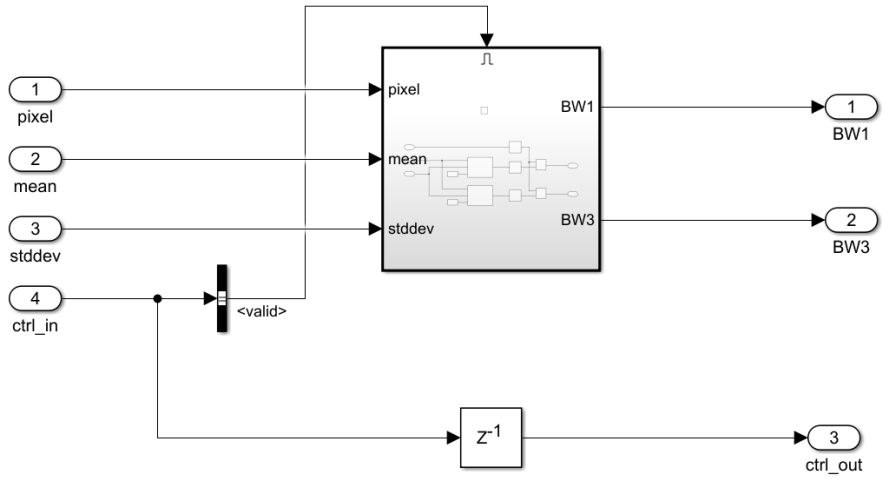


Figure 5.35: Binarization High Level Simulink Block.

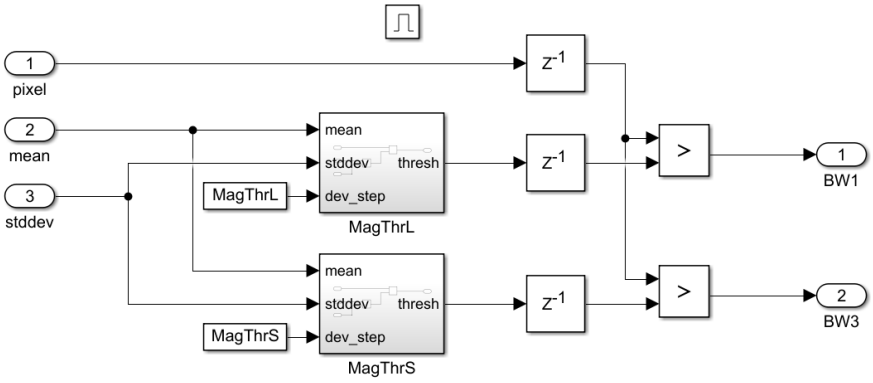


Figure 5.36: Binarization Simulink Block.

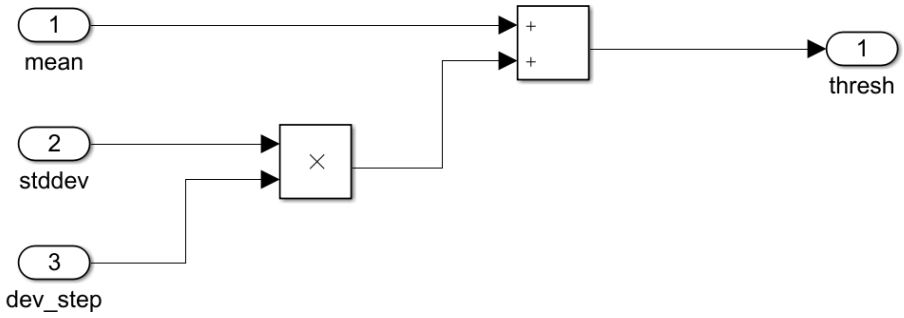


Figure 5.37: Binarization Thresholding Simulink Block.

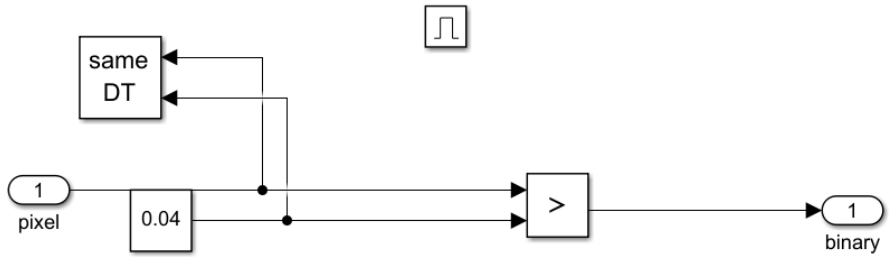


Figure 5.38: Binarized Magnitude Simulink Block.

the pixel values that were high in the direction of the Kirsch filter and were edges extracted from the CORDIC algorithm. As illustrated in (4.4), 'BW1' was the filter with the higher threshold versus 'BW3'.

As depicted in Fig. 5.37, the actual standard deviation steps chosen were done primarily through experimentation. Initially, standard deviations which extracted the top 3% and 6% of the pixel values were selected, but had poor performance. The values that worked for this application were $\sigma = 3$ and $\sigma = 7$ for the lower and higher threshold bounds respectively. These values hold little significance and Section 6.1 will discuss approaches to pick more meaningful σ values.

The last component of the binarization computation was 'BW2' as depicted in Fig. 5.38 which was the binarized version of the magnitude produced by the CORDIC algorithm. The threshold used was derived from Olson's code base when performing the edge extraction. Magnitudes below that value were ignored and as it worked in this application as well, it was left unchanged.

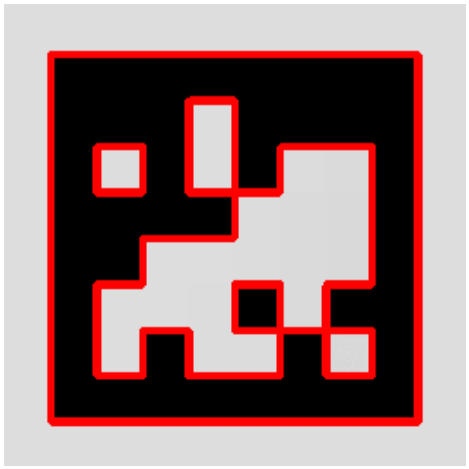
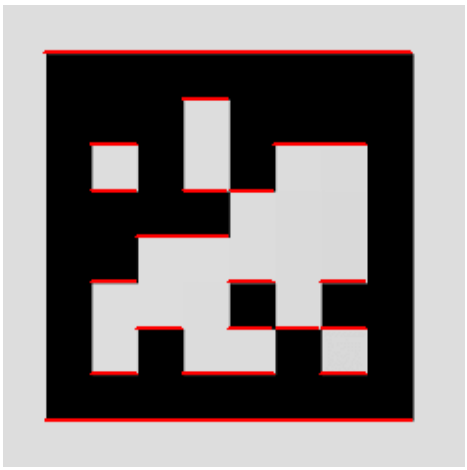
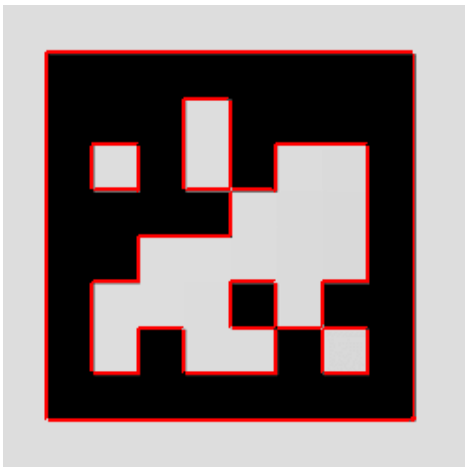


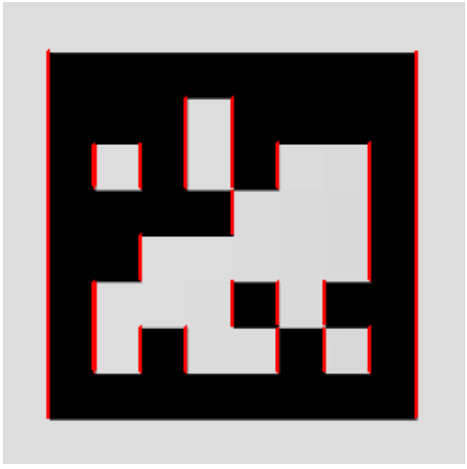
Figure 5.39: Binarized Magnitude Output Image.



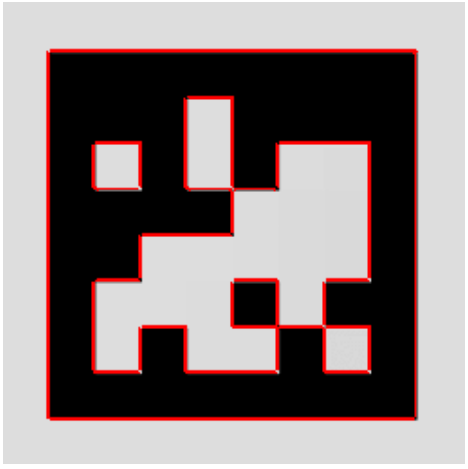
(a) BW1 North



(b) BW1 Northwest

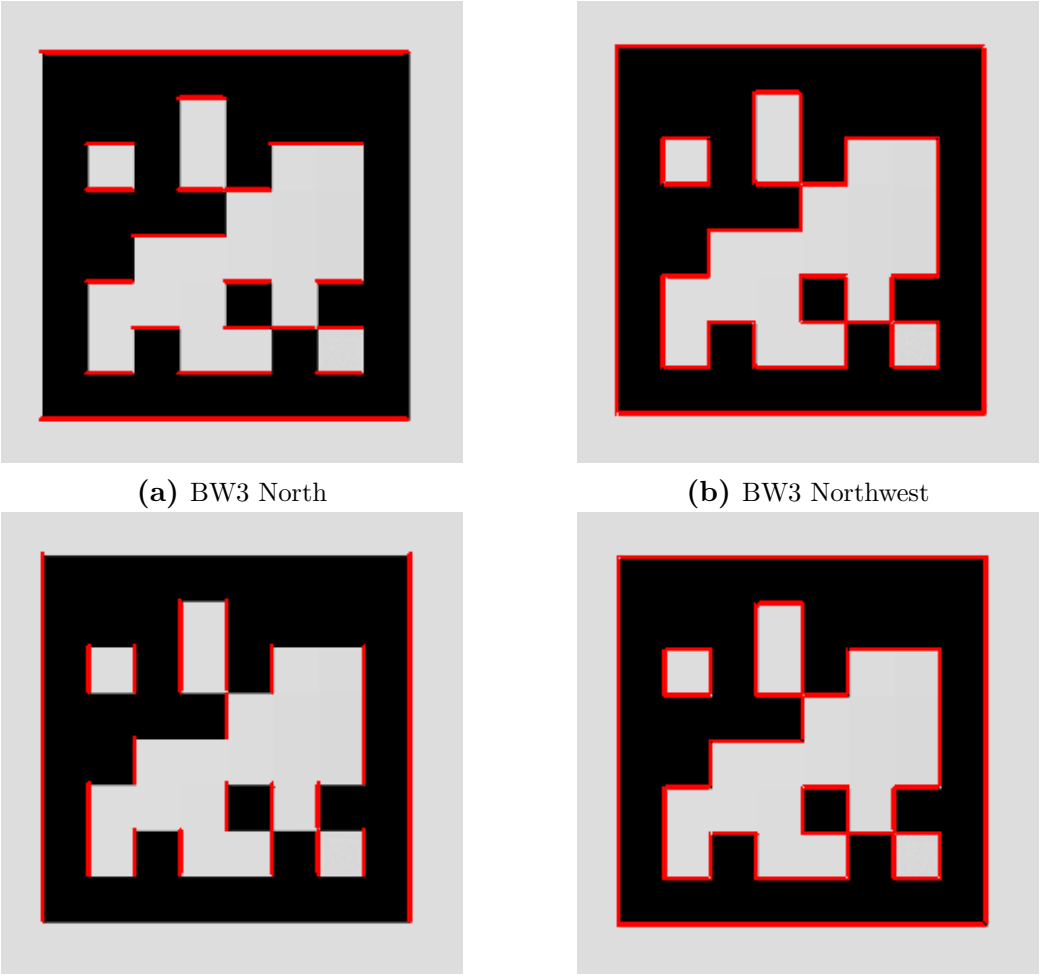


(c) BW1 West



(d) BW1 Southwest

Figure 5.40: BW1 Binarized Images Overlayed with Grayscale Output Image.



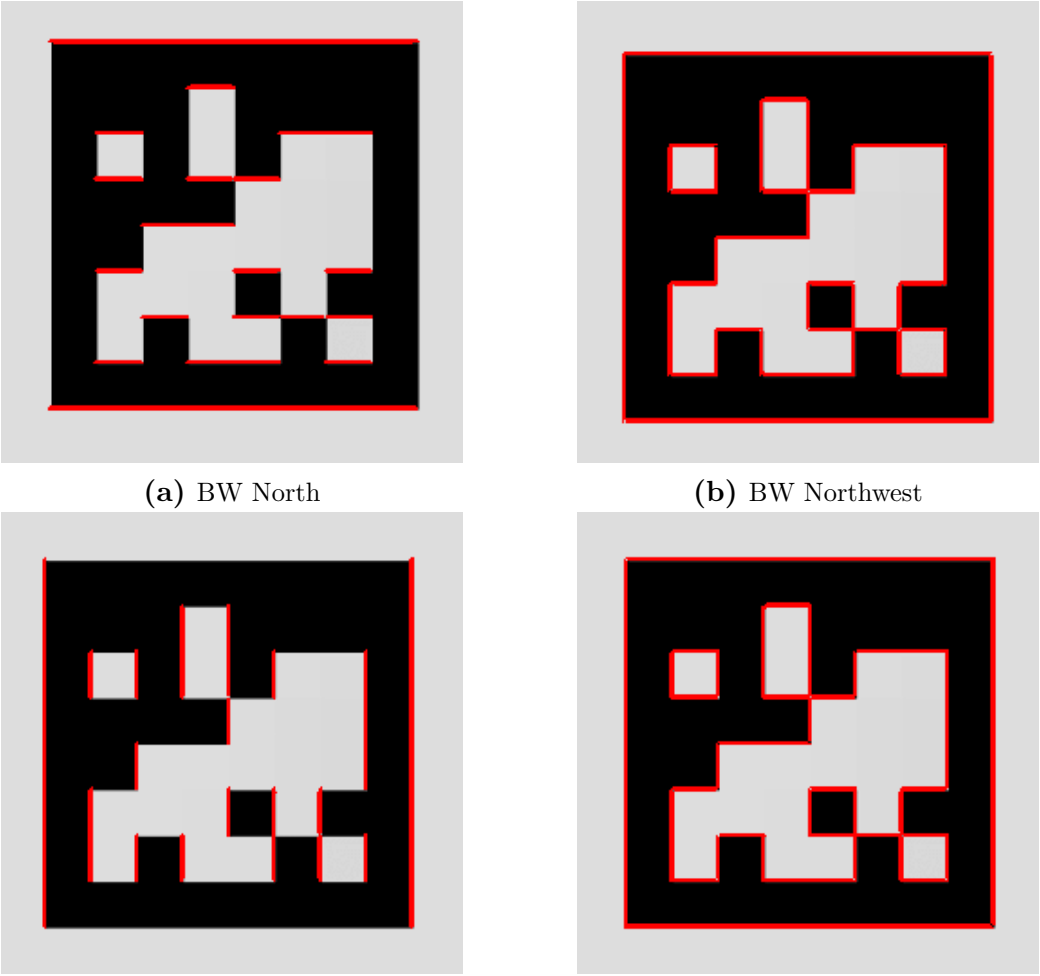
(a) BW3 North

(b) BW3 Northwest

(c) BW3 West

(d) BW3 Southwest

Figure 5.41: BW3 Binarized Images Overlayed with Grayscale Output Image.



(a) BW North **(b)** BW Northwest

(c) BW West **(d)** BW Southwest

Figure 5.42: BW Binarized Images Overlayed with Grayscale Output Image.

Table 5.12: Kirsch Filter Binarization HDL Utilization Report Summary.

Resource	Utilization	Percentage
LUT	94	0.18
FF	212	0.20
DSP	2	0.91

Table 5.13: Kirsch Filter Binarization HDL Timing Report Summary

Target Frequency: 27 MHz	Setup (ns)	Hold (ns)	Pulse Width (ns)
Worst Negative Slack	31.969	0.122	18.018
Total Negative Slack	0.000	0.000	0.000

As depicted in Fig. 5.39, Fig. 5.40, Fig. 5.41, and Fig. 5.42, the final Simulink output say overlapped and highlighted in red with the output image from "Grayscale". This was done to clearly see where the binarization sat on the actual image. Between 'BW1' and 'BW3', the thicknesses of the highlighted lines can be observed, depicting the differences in pixels that were included within the threshold. Unrealistic images such as this testing image proved difficult as the pixel distribution poorly fit the Gaussian distribution that was assumed for the typical image.

The binarization logic depicted in Table 5.12 requires few resources. Most of the resources were needed to store the thresholding step constants and calculate the number of standard deviations from the mean to cutoff. Overall, this block took minimal resources and the high level resources can be seen in Table 7.6 of the Appendix. Instead, a majority of the resources were consumed by the image statistics block as depicted in Table 7.8. This was expected as the algorithm described previously required significant resources when computing the running sum of a large image. The high level resources for this block can be seen in Table 7.7.

As depicted in Table 5.13, the binarization process met timing. Additionally, Table 7.9 shows that the image statistics block also met timing. For the image statistics, the timing illustrates that it will be able to perform computations on each pixel as they stream in but the phase delayed created by having to wait for the entire

image to be read through still exists.

Overall, the Kirsch filter, binarization, and image statistics complete HDL utilization is summarized in Table 7.11. This depicts the total number of resources used for the block, including the external bitwise operators performed on 'BW1', 'BW2', and 'BW3', to calculate the final output, 'BW'. It was expected that the other blocks would produce nearly identical results as there was no significant difference between the blocks. Table 7.12 shows that the filter in its entirety also passes timing. The Worst Negative Slack timing results were higher than the image statistics which can be assumed that the implementation algorithm found a slightly faster path for all the interconnected components.

5.4.3 Segmentation

The CCA implementation to perform steps "Segmentation" was a complex block similar to the CORDIC where rigorous testing was done at each stage of the algorithms implementation. This was the last step to be moved into hardware and provided the last bit of speed-up to achieve the desired FPS. Section 5.4.3.1 illustrates the Matlab implementation of the algorithm, showing the various intermediate steps of the algorithm. Section 5.4.3.2 depicts the final Simulink model constructed to complete this step. Section 5.4.3.3 will discuss the final implementation results of the algorithm.

5.4.3.1 Matlab Segmentation Implementation

Matlab has an internal function called bwlabeln that performs the CCA algorithm and was used to initially test the classification portion of the segmentation. The classification identified the minimum and maximum coordinate pairs of four points which were then used to find the longest line between them.

As depicted in Fig. 5.43, finding those coordinates was a simple task, requiring only a few lines of code. The top left pixel of the image was the origin (0,0). This

```
% find min x coord
minXx = min(col);
index = find(col == minXx);
minXy = row(index(1));
```

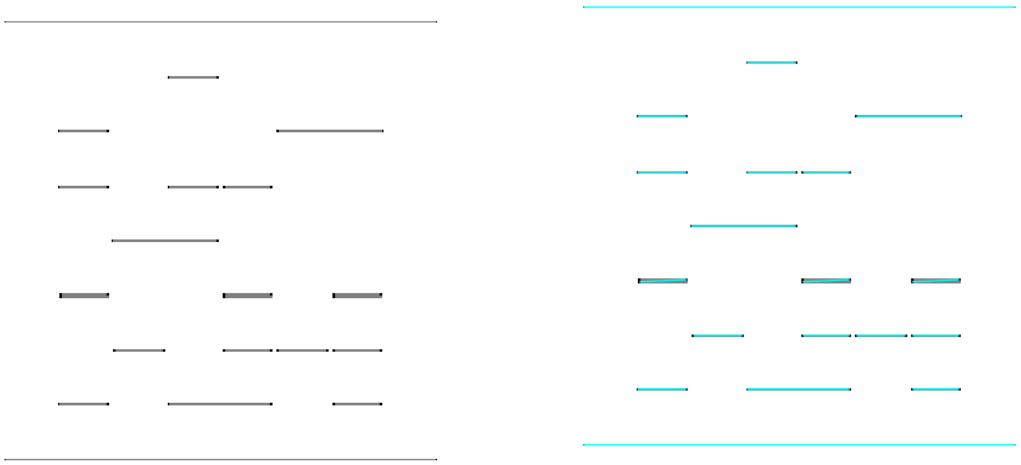
Figure 5.43: Finding the Minimum X Coordinate Code Snippet.

was done after the CCA algorithm was performed on the image where each blob has correctly been identified. Converting this to a streaming fashion was more complex as shown in Fig. 7.2. The first, overarching if statement determine whether the current pixel is a part of a new blob or if it needs to be merged. A new blob would take a new blob ID, incrementing it, and initializing the feature coordinates to the current coordinates. For mergers, more layers of logic were used to determine if it should merge with the above or to the left pixels. Most importantly was how 'the features' were extracted. When merging to the left, with no valid pixels above the current one, only the maximum X coordinate needed to be updated given it was smaller than the current X coordinate. Similarly, the maximum Y coordinate only needed to be updated when merging to the above pixel with no valid pixels to the left. When merging two different blobs as indicated by if ((left ~= 0) && (above ~= 0)), an invalidation was performed on one of the feature sets to remove the duplication during updates. During the merger, the labels that were already established for pixels were not updated but instead the merger table was updated to associate the larger IDs with the smaller IDs.

Fig. 5.44 demonstrates how the hardware implementation was simulated in Matlab by implementing a circular buffer was used to hold the last row of data. The left blob ID is inserted into the buffer and then update with the current blob ID. The above blob ID has to be checked prior to direct assignment as it could have changed since it was inserted into the buffer. After this, the next pixel is retrieved and the process is repeated.

```
buffer = circshift(buffer,1);
buffer(1) = left;
left = assignment;
aboveTemp = buffer(width-1);
if (aboveTemp ~= 0)
    above = mergerLookup(aboveTemp);
else
    above = 0;
end
```

Figure 5.44: Matlab CCA Hardware Simulation of Circular Buffer.



(a) CCA Feature Output.

(b) CCA Feature Output with Found Segments Overlaid.

Figure 5.45: Matlab CCA Feature Outputs.

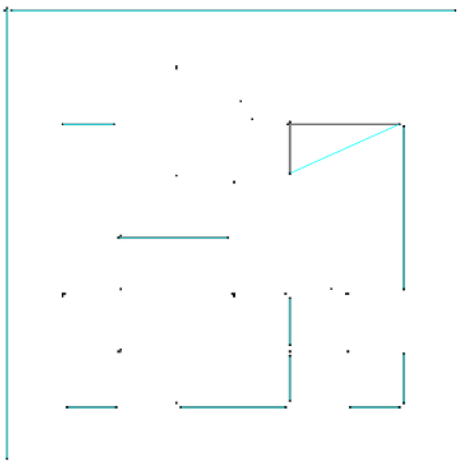


Figure 5.46: CCA Feature Output with Undesired Segments Overlaid.

Depicted in Fig. 5.45 were the output results of Fig. 5.1 at this stage. Fig. 5.45a illustrates the coordinate points that were detected. They were highlighted in black with the gray portions of the image indicating a blob has been found. Typically each blob will have four unique coordinates but there are times when the coordinates overlap, especially for an ideal image such as Fig. 5.1. Fig. 5.45b depicts the longest segment that was found for each blob, estimating the line segment that represents the blob the best. These are highlighted in blue.

As depicted in Fig. 5.46, this approach often leads to undesired artifacts. In this case, the undesired segments were the ones that were diagonal, connecting between two different edges. Although this occurs often due to all eight cardinal directions being considered, it was alleviated as these segments were often ignored and remained unconnected because of their extremely different directional values. Regardless, these segments were considered noise and should be filtered out wherever possible.

5.4.3.2 Simulink Segmentation Implementation

The Simulink implementation of the algorithm proved to be more difficult than its Matlab counterpart as the need for memory was externalized in the Simulink model. Utilizing Simulink's Matlab function block, a majority of the algorithm could remain

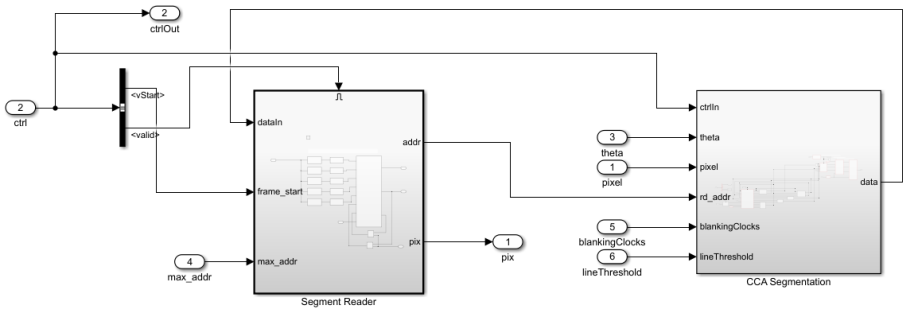


Figure 5.47: Top Level CCA Simulink Blocks.

in Matlab with synthesizable code. The memory functionality needed to be brought to the forefront as it played a role in sending the data back out to the software as well. This was done using Simulink's dual port RAM blocks to emulate the BRAMs that would be used in the FPGA.

At the top most level of the segmentation implementation was one block that performs the algorithm and stores the segment data inside a BRAM and another that reads it out and puts it on the pixel stream. This process was required as the feature extraction cannot be completed in a streaming fashion as the whole image was required to finish the processing, thus fundamentally giving a phase delay of a single frame. To handle reading out data from each Kirsch direction, only one portion of the feature was put on the stream at a time and later meshed together to fit within the 64 bit-wide VDMA. The reader handles this by having an internal counter that puts each segment feature on the stream in five cycles before reading the next segment. To reiterate, the segment feature being extracted was the points making up that segment and the theta from the maximum X point.

As depicted in Fig. 5.48, the segmentation process was broken down into 3 major blocks as described in Section 2.4. The 'CCA Algorithm' block performs the CCA implemention, extracting out minX, maxX, minY, and maxY components that would be fed into the 'Feature Extraction' process. Between them was a set of BRAMs that saved the outputs from the 'CCA Algorithm'. The last set of blocks were the 'Write

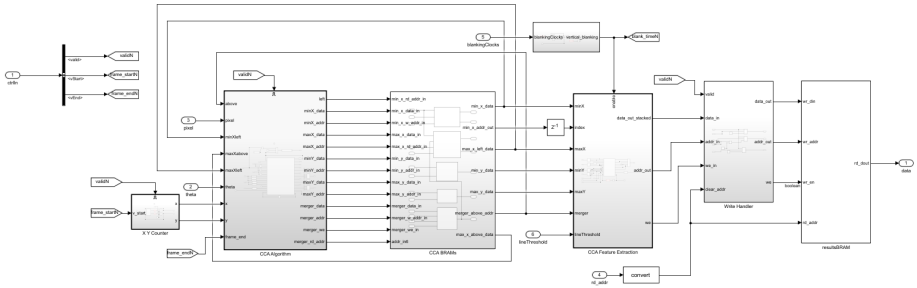


Figure 5.48: CCA Segmentation Simulink Blocks.

Handler' and the results BRAM. The 'Write Handler' simply controlled where each feature was put into memory and how it was cleared when reading. Although most of the blocks were implemented using native Simulink blocks, more complex algorithms such as the 'CCA Algorithm' and the 'Feature Extraction' leveraged Matlab function blocks to simplify the design. They required slight modifications to the algorithm to make the code HDL friendly by using very simple code.

As depicted in Fig. 5.49, the code could stay mostly the same with minor tweaks to how the data was accessed and stored. Instead of being able to access data from a matrix as shown in the comments, this was done by updating the data of that coordinate with the current address. Similarly, the assignment buffer which exists in the pure Matlab version in Fig. 5.44 was depicted as an actual circular buffer. The buffer was represented as two delays, one unit delay for the left assignment (the previous group) and the $rowlength - 2$ for the above assignment. One of the major complications of this design was having the data pre-fetched before knowing which data was needed. This required a few extra BRAMs to store this data and fetch it whenever needed as depicted in Fig. 5.50. The max X point was special relative to the other points as it needed to save both the above group's max X point and the left group's max X point. Only the left group's max X point was passed to the feature extraction. The other points could be left within a single BRAM but utilized a dual port BRAM that could read and write data at the same time.

Within the 'Feature Extraction' block, the line segment length computation often

```
% new ID
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%blobID = blobID + 1;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
blobIDout = blobIDin + uint16(1);
assignment = blobIDin;

% load in all bram entries and mark as valid
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%minX(blobID,:) = [x,y,theta(counter),1];
%maxX(blobID,:) = [x,y,theta(counter),1];
%minY(blobID,:) = [x,y,theta(counter),1];
%maxY(blobID,:) = [x,y,theta(counter),1];
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
minX_addr = blobIDin;
maxX_addr = blobIDin;
minY_addr = blobIDin;
maxY_addr = blobIDin;
minX_data = [x ;y; theta];
maxX_data = [x ;y; theta];
minY_data = [x ;y; theta];
maxY_data = [x ;y; theta];
```

Figure 5.49: CCA Simulink Matlab Function Block New Blob ID Code Snippet.

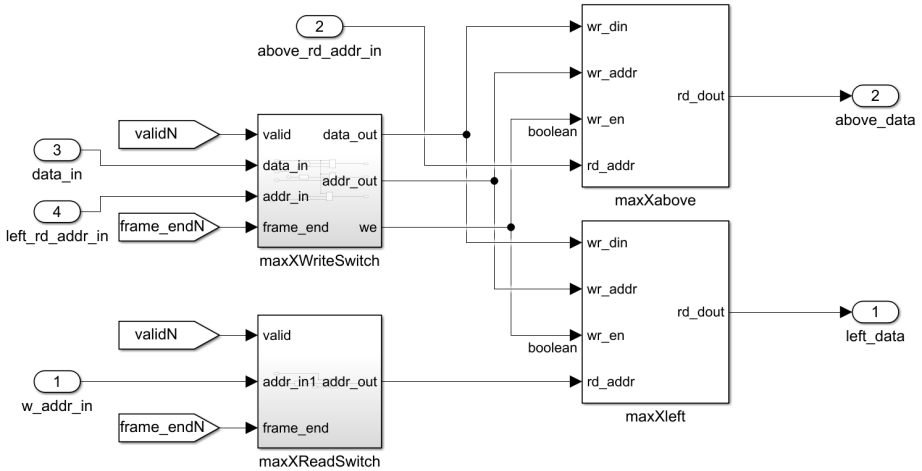


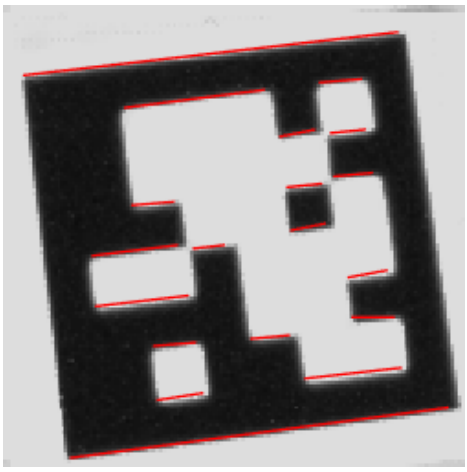
Figure 5.50: Max X Simulink BRAM Blocks.

requires a square root computation. This was avoided by simply ignoring the square root, summing the absolute values of the δx and δy. It was unnecessary to compute the actual length of the line segment at this point as the software would still need to iterate over all of the found segments prior to continuing the software pipeline as each segment will need to be reoriented to follow the correct winding. Instead, only the relativity between line segment lengths was required so the simple equation depicted in (5.5) was used.

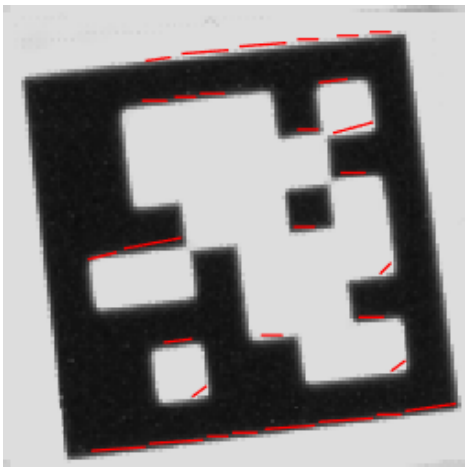
$$|x1 - x2| + |y1 - y2| \tag{5.5}$$

The final results BRAM depicted in Fig. 5.48, stores all the segments for a given image. Sizing this BRAM and the intermediate BRAMs was a difficult choice between the number of segments to store and the size. With the limited resources provided by the Zynq 7020, the addresses was kept at a bit width of 11, giving 2048 segments. It was assumed that this was sufficient as the testing samples given produced less than 1000 segments and blobs on average. The sizing impacts of using a 11 bit width will be explored in Section 5.4.3.3. Simulations of the model were done prior to hardware generation to ensure the model worked appropriately.

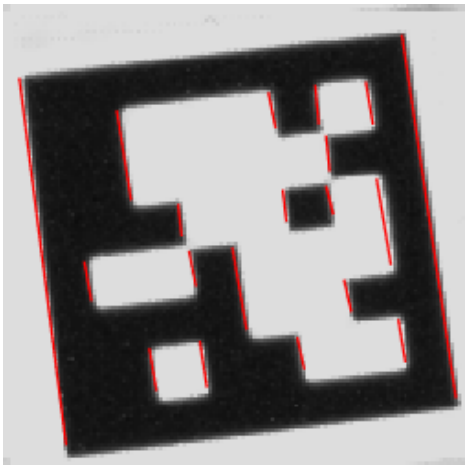
As depicted Fig. 5.51, the segments detected for each Kirsch direction was well lined up with the underlying gray image used as a reference. It can be clearly seen that the directions that were off-axis of the April Tag image segmented poorly, creating many short, disjointed lines that do not fit well with the gray image. As illustrated in Fig. 5.33d, the blobs that were detected spanned too far, joining corners that should not be joined which would create segments that have a greatly different theta than the underlying pixels. On-axis Kirsch directions gave clean segments that closely followed the image which provides the final quad and detected tag as illustrated in Fig. 5.52.



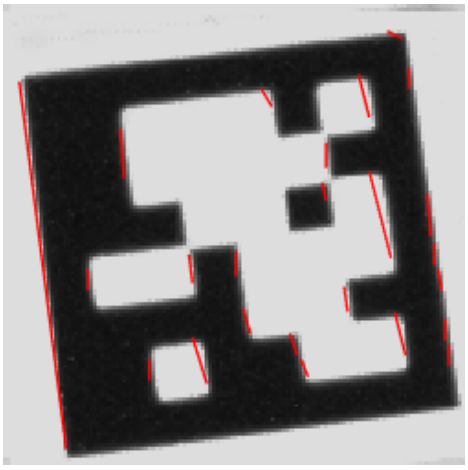
(a) CCA North Image.



(b) CCA Northwest Image.

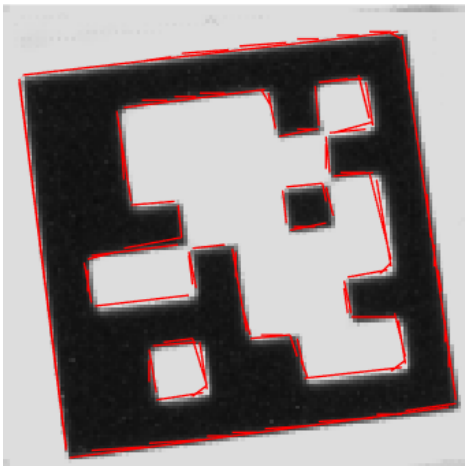


(c) CCA West Image.

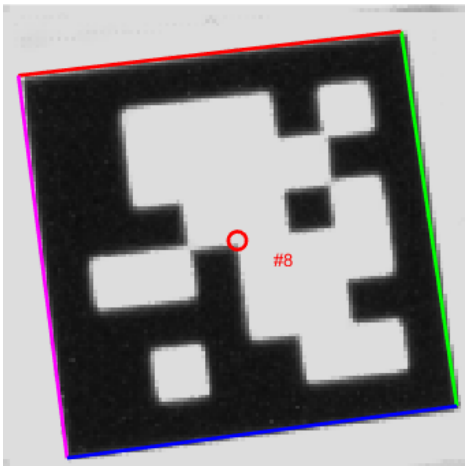


(d) CCA Southwest Image.

Figure 5.51: CCA Kirsch Output Images.



(a) CCA All Directions Image.



(b) CCA Detected Tag.

Figure 5.52: CCA Output Images.

Table 5.14: CCA Segmentation HDL Coder Utilization Report Summary.

Resource	Utilization	Available	Utilization %
LUT	1324	53200	2.49
FF	325	106400	0.31
DSP	14.50	140	10.36

Table 5.15: CCA Segmentation HDL Coder Timing Report Summary.

Target Frequency: 27 MHz	Setup (ns)	Hold (ns)	Pulse Width (ns)
Worst Negative Slack	10.627	0.124	18.018
Total Negative Slack	0	0	0

5.4.3.3 Hardware Segmentation Implementation

As mentioned before, the size of the BRAMs was a major concern for this portion as the original design targeted a 16 bit width. An address bus this wide failed implementation as the on-board BRAMs was not sufficient to hold all of the BRAMs, forcing a numerous amount of LUTs to be used for memory which the board does not have. Reducing it down to 11 bits gave the following results.

As depicted in Table 5.14, this algorithm consumed a lot more BRAMs than any other others. An addition to saving space through smaller BRAM capacities, the BRAM data was also shrunk from 16 bit to 11 bit for all data types. For this implementation, the x and y coordinates were less than 752, thus allowing it to fit within a 11 bit bus. For simplicity, the theta values were also truncated down to 11 bits, removing 5 of the least significant bit (LSB) fractional bits. This was assumed to be okay as the segmentation algorithm only uses the theta values as an estimation to which direction the line was facing, relying mostly on the computed theta that was based on the changes in x and y values. Overall, this choice limited the number of BRAMs required by a single segmentation implementation to something reasonable and could be scaled for different hardware.

Unlike the other blocks, the segmentation algorithm required a lot more time to complete. Regardless, it was able to meet timing with the 27 MHz clock frequency

dictated by the camera clock frequency.

5.5 Sizing, Accuracy, & Timing

This section introduces the final hardware size, accuracy, and execution timings of the various algorithms implemented. For this thesis, there were three algorithms that were implemented and two baseline algorithms. The first algorithm that was implemented was a simple software optimization of Olson's code base. This consisted of restructuring the code, utilizing OpenCV functionality and other C/C++ optimization techniques. It will be referred to as "April Tag Baseline Optimized". The second algorithm was depicted in Fig. 4.5 which was the hardware pipeline that moved "Grayscale", "Normalize", "Gaussian Smoothing", and "Gradient Magnitude and Direction" into hardware. This will be referred to as "Ravven Tag MagTheta". The third algorithm, referred to as "Ravven Tag CCA", was depicted in Fig. 4.11 which additionally moved "Edge Extraction", "Clustering", and "Segmentation" into the hardware, improving upon "Ravven Tag MagTheta". The first baseline algorithm was Olson's original C/C++ code base, referred to as "April Tag Baseline". The second baseline algorithm was the one implemented in Python. This algorithm was recently discovered was convenient to work against considering it worked within Python without any additional configurations. It will be referred to as "PyApril Tag".

When performing benchmarks on the algorithms described above, it was important to choose appropriate input images to accurately represent real-world scenarios. The testing image depicted in Fig. 5.1 was a fine image to test for a functioning algorithm, but did not provide accurate results for accuracy or timing, especially with software that can have variable execution time. There was a wide range of images that were generated to test the algorithm, Fig. 5.53 depicts such an image. The image was captured with the Fusion 2 on-board camera.

The final benchmark results were split into two categories. Section 5.5.2 discusses

Figure 5.53: Real-World Input Image.

the accuracy results for each algorithm. Section 5.5.3 discusses the timing results for each algorithm.

5.5.1 Sizing

The final sizing of the algorithm illustrates not only the combined usage of all the sub modules explored in the thesis but the surrounding logic used to interface with the camera and processor as illustrated in Fig. 4.2 and Fig. 4.3. It was important to consider the surrounding logic for this thesis as it consumes a non-trivial amount of resources. For this implementation this logic has to consider the translation logic of the video input stream from the camera to an AXI4-Video Stream interface, the VDMA AXI4-Video Stream logic up to the processor, and any configuration parameters sent to the implemented system through AXI4-Lite buses.

The final utilization consumed a large portion of the hardware resources available as illustrated in Table 5.16. Without the surrounding hardware, the April Tag optimization consumed a large portion of the LUTs, BRAMs, and DSPs as expected. Collectively, each sub module contributed to the total number of LUTs used, with no

Table 5.16: Final HDL Coder Utilization Report Summary.

	Complete without Wrappers		Complete with Wrappers		
Resource	Utilization	Utilization %	Utilization	Utilization %	Available
LUT	12673	23.82	27970	52.58	53200
LUTRAM	278	1.60	1308	7.52	17400
FF	10894	10.24	29250	27.49	106400
BRAM	64	45.71	94.50	67.50	140
DSP	81	36.82	89	40.45	220
I/O	0	0	35	28	125
MMCM	0	0	1	25	4

Table 5.17: Final HDL Coder Timing Report Summary.

Target Frequency: 27 MHz	Setup (ns)	Hold (ns)	Pulse Width (ns)
Worst Negative Slack	0.364	0.040	7.000
Total Negative Slack	0	0	0

specific sub module consuming a large portion of it. The BRAMs were mostly consumed by the CCA segmentation algorithm described in Section 4.3.4.3. The DSPs were mostly consumed by the various filters that were implemented. Including the surrounding hardware, a majority of that hardware consumed LUTs and FFs, adding very little to the BRAMs, LUTRAM, and DSPs. Additionally, since it connected with external hardware, it also consumed I/O to connect to external memory for VDMA and the processor and mixed mode clock manager (MMCM) to manage the different clock rates between the processor, external memory, and the camera.

It was expected that the final solution barely met timing with very little slack as depicted in Table 5.17. This was due to a combination of having to travel through each sub module depicted before as well as moving through the CCA segmentation algorithm that consumed the most amount of time. Overall, the hardware implementation fit well within the constraints provided by the camera and the limited resources given by the Zynq 7020 SoC. Using the same hardware, other algorithms can be placed adjacent to this one with a significant number of resources still available to it. The left over resources also leaves space for optimizations in the algorithm to

Table 5.18: Accuracy Test Summary.

	Accuracy
April Tag Baseline	95%
PyApril Tag	90%
April Tag Baseline Optimized	95%
Ravven Tag MagTheta	95%
Ravven Tag CCA	95%

improve accuracy or timing of the algorithm such as increasing the bit depth of the CORDIC algorithm or BRAM depth.

Although the algorithm could be synthesized and implemented through Simulink's HDL Coder, the model itself did not operate correctly on the hardware. With the process presented throughout this thesis, it was the CCA Segmentation portion of the algorithm that failed to produce the correct segments in the image on hardware. Regardless of this inability to get an end to end solution, the timing presented in Section 5.5.3 was assumed to be highly accurate as the HDL code would effectively be ignored in the processing time of the image due to the whole image being processed prior to being written into memory for the software pipeline to operate on. Chapter 6 will go into further detail on how this will be resolved.

5.5.2 Accuracy

The workflow regarding the accuracy tests utilized the exported images from the Simulink models to skip through some of the steps in software. Additional testing was also done on the Snickerdoodle board itself, leveraging Python to load and capture whether or not the April Tag was detected. Table 5.18 summarizes the results of twenty test images that had varying April Tag images as depicted in Fig. 5.53.

In this small sample test suite, an April Tag was present in all the images. Although all the images had tags present, there was one image that did not have a complete tag visible as part of it was clipped at the edge of the image. This proved to fail for all the algorithms. A complete end-to-end solution was not fully realized

Table 5.19: Timing Breakdown for "April Tag Baseline".

Step	Timing (ms)
Grayscale	16.8340
Normalization	17.54
Gaussian Smoothing	75.3419
Gradient Magnitude & Direction	90.3862
Edge Extraction	233.204
Clustering	26.8581
Segmentation	8.32009
Segment Connection	1.1518
Create Quads	0.36788
Decode Quads	17.3571
Duplication Removal	0.004053
Pose Estimation	15.570
Total	520.505

but the testing was done using various stages and estimations with the components that were working. A majority of the basic tests were done using outputs from the Simulink or Matlab results that were saved off as either images or CSV files that could be loaded in Python to imitate the hardware process. The results depicted in Table 5.18 were genearted based on this flow.

5.5.3 Timing

The workflow regarding the timing tests were done completely on the Fusion 2. After verifying the algorithms were accurate enough, Python was leveraged to perform timing on each algorithm over the sample test suite. Timing results were captured for each algorithm, either using built-in timing options or implementing the timers inside the code. The following results were small snapshots of running the algorithm with Fig. 5.53.

Table 5.19 depicts the timing results for the "April Tag Baseline". As mentioned by [3], the "Edge Extraction" process took the most amount of time. Initial testing with Fig. 5.1 had a significantly lower "Edge Extraction" time compared to Fig. 5.53, thus illustrating the importance of using testing images that provide accurate repre-

Table 5.20: Timing Breakdown for "April Tag Baseline Optimized".

Step	Timing (ms)
Pre-Processing Time	0.02503
Grayscale	16.8340
Normalization	9.4039
Gaussian Smoothing Time	37.9879
Gradient Time	96.2949
Establish Constants Time	0.00501
Edge Extraction Time	240.420
Cluster Time	28.265
Segmentation Time	9.4049
Establish Constants Time	0.00620
Segment Connection Time	1.230
Create Quads Time	0.43607
Decode Quads Time	13.7770
Duplication Removal Time	0.00787
Pose Estimation Time	21.8909
Total	475.989

sentation of the user space as well as pushing the algorithm to its limit.

The "PyApril Tag" provided a breakdown as well in the form of a binary but failed to produce consistent results with its Python counterpart. The Python wrapper produced results around two times faster than the C code base. The final results for "PyApril Tag" was summarized in Table 5.23.

Table 5.20 depicts the minor optimization provided to the "April Tag Baseline" library. The key points that have been reduced was "Grayscale" and "Gaussian Smoothing", providing a speed up of 1.87 and 1.98 respectively. The overall speed up of this slight optimization was 1.09.

As depicted in Table 5.21, "Grayscale", "Normalize", "Gaussian Smoothing", and "Gradient Magnitude and Direction" have been removed and replaced by a pre-processing step. This pre-processing step was required to convert the raw data provided by the hardware into meaniful data that can be utilized by the software. Primarily, this took the form of converting and normalizing the fixed point output of "Gradient Magnitude and Direction" back into floating point and normalizing the

Table 5.21: Timing Breakdown for "Ravven Tag MagTheta".

Step	Timing (ms)
Pre-Processing Time	25.7900
Establish Constants Time	0.00715
Edge Extraction Time	1037.76
Cluster Time	129.030
Segmentation Time	59.1440
Establish Constants Time	0.01597
Segment Connection Time	13.6719
Create Quads Time	8.65602
Decode Quads Time	51.8641
Duplication Removal Time	0.00906
Pose Estimation Time	6.85096
Total	1332.80

(a) "April Tag Baseline" "Edge Extraction" Image Representation.

(b) "Ravven Tag MagTheta" "Edge Extraction" Image Representation.

Figure 5.54: "Edge Extraction" image representation comparison

"Grayscale" image.

Additionally, as depicted in Table 5.21, "Edge Extraction" took a significantly longer time to compute. This was assumed to be because of the quantization experienced by first converting the magnitude and direction outputs of "Gradient Magnitude and Direction" to images and then converting back into their floating point counterparts. This was done to make it easier to move those images around and visualize them. Fig. 5.54 depicts the inverted versions of the edges that were found. The larger the edge that was detected, the darker the color was.

Fig. 5.54b clearly illustrates the significant increase in edges that were extracted

Table 5.22: Timing Breakdown for "Ravven Tag CCA".

Step	Timing (ms)
Pre-Processing Time	0.02885
Segment Correction Time	0.4690
Establish Constants Time	0.00787
Segment Connection Time	1.864
Create Quads Time	3.088
Decode Quads Time	17.827
Duplication Removal Time	0.00906
Pose Estimation Time	3.7260
Total	27.0197

although the same image was provided. As depicted in Fig. 5.38, the 0.004 constant was also used to ignore edges of insignificant strength. For "April Tag Baseline", this was a much larger set of pixels since it did not need to be quantized at any point. As for "Ravven Tag MagTheta", that 0.004 constant indicates that any pixel above the value of 1 (ranging from 0 to 255) would be considered, giving a much easier benchmark for any noise to be propagated through. Upon changing this constant to 0.08 which ignores pixel values 20 and below, the output was far less noisy and much faster. The speed increased from 1225.93 ms to 220.19, counting "Edge Extraction", "Clustering", and "Segmentation", with that minor tweak. Overall, without the tweak, the "Ravven Tag MagTheta" had an overall slowdown of 0.39.

As depicted in Table 5.22 and mentioned before, "Ravven Tag CCA" performed the fastest with an overall speed up of 19.26 compared to "April Tag Baseline", 17.62 compared to "April Tag Baseline Optimized", 49.33 compared to "Ravven Tag MagTheta", and 3.35 compared to "PyApril Tag". The final results of the "Ravven Tag CCA" algorithm provided an operating speed of 37 FPS at its slowest, reliably providing more than 30 FPS. The pre-processing required in this step was a combination of converting the raw data into structured data and providing the orientation correction of the raw segment data in "Segment Orientation Correction". By offloading the bulk of the workload described in Table 5.19 onto the hardware, it was possible

Table 5.23: Timing Test Summary.

Algorithms	Timing (ms)
April Tag Baseline	478.808
PyApril Tag	90.4473
April Tag Baseline Optimized	412.244
Ravven Tag MagTheta	1508.72
Ravven Tag CCA	23.2363

to see these drastic increases in speed. This comes at the cost of requiring an FPGA and introduces a phase delay between the captured image and the data processing. For "Ravven Tag CCA", the phase delay was a total of two frames once the pipeline was filled. One frame originates from the statistical analysis being calculated for the next frame which introduces error for rapidly changing scenes and another from the complicated BRAM output to VDMA for the CCA Segmentation implementation.

Table 5.23 depicts the average results of a hundred runs for each of the twenty images in the sample test suite. This was a total of 2000 samples.

Chapter 6

Conclusion

We have described a hardware optimization for April Tag's visual fiducial system that depicts a significant improvement upon previous methods. The algorithm was built against the Fusion 2 Snickerdoodle board and implemented on the SoC platform. Python was leveraged to establish a separated connection between the C/C++ code base and the Simulink generated HDL for testing and verification. Using Matlab and Simulink, rapid prototyping was conducted to ensure algorithms operated as expected before implementing them in hardware for ease of debugging. With Python.Boost, it was possible to provide a clean and simple interface between the Python and C/C++ code for testing. Given the small sample test suite, the algorithm was showed to have performed at the desired operating frequency for real-time flight of 30 FPS when utilizing the "Ravven Tag CCA" algorithm implementation.

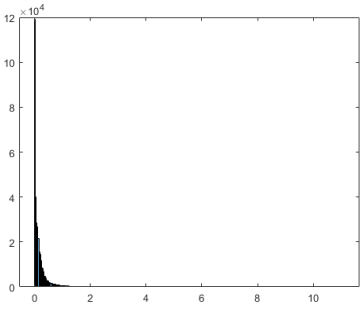
6.1 Future Work

Although the optimizations presented were promising, several additional improvements can be done to further increase the efficiency of using April Tag fiducial system for GPS-denied navigation and IMU replacement or drift correction. One of the major bottlenecks for "Ravven Tag CCA" was the BRAM chain implementation. This created an unnecessary frame phase delay that can be removed with the usage of a more immediate retrieval mechanism such as DMA. Unfortunately, this would

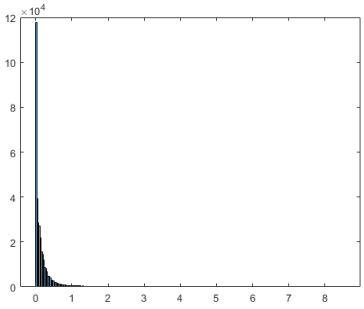
not remove the slight delay between frames that needs to be used to perform feature extraction on the image. It was not implemented in this application to maintain a common interface between all hardware implementations. This may also reduce the amount of resources required to maintain a secondary BRAM that does not hold or produce any new information, allowing the other BRAMs to have a greater depth that would make them more robust to noisier images.

Throughout the implementation of the April Tag algorithm, shortcuts were taken to simplify the algorithm as much as possible without impeding on the accuracy and reliability of it. These can slowly be stripped away to further reduce the sizing of the final implementation. For example, the symmetrical filters presented in the paper were not fully utilized due to a simple buffering mechanism. Additional shift registers can be used to properly merge the horizontal and vertical versions of the filter, removing more arithmetic logic. These shift registers would need to leverage other signals provided on the control bus such as end of line (represented as hend in Simulink) and the start of line (represented as hstart in Simulink) so it would properly shift the saved values to build the full three by three kernel. Another optimization would be to pull out the Kirsch buffering mechanism that extracts the same three by three pixels but performing different arithmetic to it. This would be a simple shift in architecture but may make it more complicated in the model design. Overall, these optimizations would reduce the sizing of the overall hardware implementation.

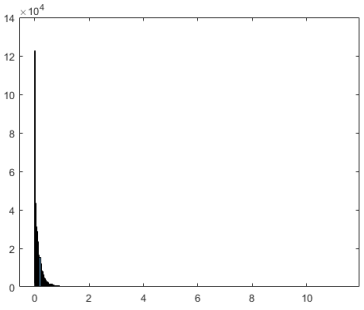
Although the accuracy presented in Table 5.18 illustrated a fairly robust algorithm, it would occasionally fail for unrealistic images such as Fig. 5.1 due to the pixels of the Kirsch image that were assumed to be a Gaussian distribution with the black and white pixels being the minority of the image. Olson circumvents this problem by utilizing a mixture of local differences between the maximum and minimum value of a region. That were then averaged again with the surrounding averages, using this average as the threshold for edge detection. This algorithm is difficult to



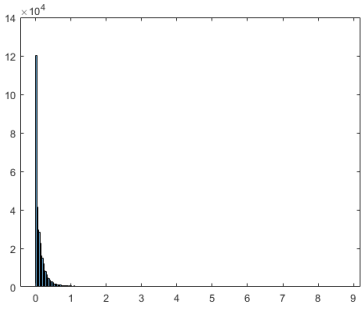
(a) Matlab Histogram Plot of Input1.bmp Kirsch North Output.



(b) Matlab Histogram Plot of Input1.bmp Kirsch Northwest Output.



(c) Matlab Histogram Plot of Input1.bmp Kirsch West Output.



(d) Matlab Histogram Plot of Input1.bmp Kirsch Southwest Output.

Figure 6.1: Matlab Histrogram Plots of Input1.bmp.

implement in hardware as the output of each kernel depends on its neighbors. Given the nature of the tag, experimentation can be done assuming that the pixel values in a Kirsch image are bi-modal, with the black and white pixels being the majority of the image. The bi-modal distribution has two peaks, one being at the assumed maximum value and the other at the assumed minimal value.

Although this assumption may not hold true for all images, this property can be observed by using Fig. 5.1. Most of the image has low values as observed in Fig. 6.1, and drastically skew the variance and the standard deviation of the population, making the threshold consider too few pixels. Instead, separating the Kirsch image into a bi-modal distribution would provide a more accurate average for the higher values. This would be more robust to darker images which would have even more low value pixels as well as to images that have a lot of contrast. Additional resources would

need to be invested in determining algorithms that could optimally separate the two distributions while determining the average and standard deviation of the higher pixel value distribution.

The final step to this research would be to implement the entire system as an end to end product that would process the input image, provide real-time localization, and be fed into a larger controller for UAV stabilization. This will require surrounding mechanisms to UAV flight control such as control loops, sensor fusion, and a streamlined processing framework. Although the Fusion 2 framework was optimal for prototyping by providing a clean interface to the executing code, a fully functional system would require real-time components that embedded Linux traditionally does not offer. Implementing the proposed algorithm would experience much better and reliable performance on real-time components without the unnecessary overhead of an fully-fledged operating system.

Bibliography

[1] *MT9V034*, ON Semiconductor, Jan 2017, rev D.

[2] Mar 2021. [Online]. Available: https://www.faa.gov/about/office_org/headquarters_offices/ato/service_units/techops/navservices/gnss/faq/gps/

[3] E. Olson, "AprilTag: A robust and flexible visual fiducial system," in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, May 2011, pp. 3400–3407.

[4] H. Kato and M. Billinghurst, "Marker tracking and hmd calibration for a video-based augmented reality conferencing system," in *Proceedings 2nd IEEE and ACM International Workshop on Augmented Reality (IWAR'99)*, 1999, pp. 85–94.

[5] M. Fiala, "Artag, a fiducial marker system using digital techniques," in *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, vol. 2, 2005, pp. 590–596 vol. 2.

[6] D. Wagner, G. Reitmayr, A. Mulloni, T. Drummond, and D. Schmalstieg, "Pose tracking from natural features on mobile phones," in *2008 7th IEEE/ACM International Symposium on Mixed and Augmented Reality*, 2008, pp. 125–134.

[7] J. Wang and E. Olson, "AprilTag 2: Efficient and robust fiducial detection," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, October 2016.

[8] R. Zhang, "Fpga soc fiducial system for unmanned aerial vehicles," Master's thesis, Rochester Institute of Technology, 2018. [Online]. Available: http://www.ravvenlabs.com/publications.html

[9] "Home." [Online]. Available: https://www.craftcameras.com/

[10] M. Zucker and I. Dulin, "swatbotics/apriltag." [Online]. Available: https://github.com/swatbotics/apriltag

[11] "Unmanned aerial vehicle (uav) market by product type, by wing type, by operation mode, by range, by maximum takeoff weight (mtow), by system, by application, by end-user, forecasts to 2027," Dec 2020. [Online]. Available: https://www.emergenresearch.com/industry-report/unmanned-aerial-vehicle-market

[12] "Amazon prime air," Mar 2021. [Online]. Available: https://en.wikipedia.org/wiki/Amazon_Prime_Air

[13] "Zynq-7000 soc." [Online]. Available: https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html

[14] "Documentation," Nov 2020. [Online]. Available: https://krtkl.com/resources/docs/

[15] D. L. N. Hettiarachchi, V. S. P. Davuluru, and E. J. Balster, "Integer vs. floating-point processing on modern fpga technology," in *2020 10th Annual Computing and Communication Workshop and Conference (CCWC)*, 2020, pp. 0606–0612.

[16] F. De Dinechin and M. Istoan, "Hardware implementations of fixed-point atan2," in *2015 IEEE 22nd Symposium on Computer Arithmetic*, 2015, pp. 34–41.

[17] A. Rosenfeld and J. L. Pfaltz, "Sequential operations in digital picture processing," *J. ACM*, vol. 13, no. 4, p. 471–494, Oct. 1966. [Online]. Available: https://doi-org.ezproxy.rit.edu/10.1145/321356.321357

[18] D. Bailey and C. Johnston, "Single pass connected components analysis," *Proceedings of Image and Vision Computing*, 01 2007.

[19] D. G. Bailey, C. T. Johnston, and Ni Ma, "Connected components analysis of streamed images," in *2008 International Conference on Field Programmable Logic and Applications*, 2008, pp. 679–682.

[20] C. T. Johnston and D. G. Bailey, "Fpga implementation of a single pass connected components algorithm," in *4th IEEE International Symposium on Electronic Design, Test and Applications (delta 2008)*, 2008, pp. 228–231.

[21] M. J. Klaiber, D. G. Bailey, Y. O. Baroud, and S. Simon, "A resource-efficient hardware architecture for connected component analysis," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 26, no. 7, pp. 1334–1349, 2016.

[22] B. Mah, "Model-based design for visual localization via stereoscopic video processing," Master's thesis, Rochester Institute of Technology, 2017. [Online]. Available: http://www.ravvenlabs.com/publications.html

[23] S. Asano, T. Maruyama, and Y. Yamaguchi, "Performance comparison of fpga, gpu and cpu in image processing," in *2009 International Conference on Field Programmable Logic and Applications*, 2009, pp. 126–131.

[24] M. Qasaimeh, K. Denolf, J. Lo, K. Vissers, J. Zambreno, and P. H. Jones, "Comparing energy efficiency of cpu, gpu and fpga implementations for vision kernels," in *2019 IEEE International Conference on Embedded Software and Systems (ICESS)*, 2019, pp. 1–8.

[25] D. Rosa, "Remi." [Online]. Available: https://github.com/dddomodossola/remi

[26] R. Rigamonti, A. Sironi, V. Lepetit, and P. Fua, "Learning separable filters," *2013 IEEE Conference on Computer Vision and Pattern Recognition*, 2013.

[27] R. A. Kirsch, "Computer determination of the constituent structure of biological images," *Computers and biomedical research*, vol. 4, no. 3, pp. 315–328, 1971.

[28] "Simulink - simulation and model-based design." [Online]. Available: https://www.mathworks.com/products/simulink.html

[29] "Frame to pixels." [Online]. Available: https://www.mathworks.com/help/visionhdl/ref/imagestatistics.html

[30] F. C. Crow, "Summed-area tables for texture mapping," *Computer graphics (New York, N.Y.)*, vol. 18, no. 3, pp. 207–212, 1984.

[31] "Frame to pixels." [Online]. Available: https://www.mathworks.com/help/visionhdl/ref/imagestatistics.html

Glossary

April Tag Baseline

A C/C++ April Tag library provided by Olson [3] from University of Michigan.

April Tag Baseline Optimized

A C/C++ April Tag library that provides slight optimization on Olson's [3] existing code base.

Clustering

Process of the April Tag algorithm that groups edge pixels together based on their gradient magnitude and directions.

Create Quads

Process of the April Tag algorithm that performs a depth-first-search on a given list of segments to determine if they create a quad.

Decode Quads

Process of the April Tag algorithm that determines if a given quad contains an April Tag based on the expected dimensions of a tag.

Duplication Removal

Process of the April Tag algorithm that removes decoded quads that are overlapping.

Edge Extraction

Process of the April Tag algorithm that separates edges of interest from the background.

Gaussian Smoothing

A common image processing step that utilizes a Gaussian distribution to create a kernel. Typically used to remove high frequency noise.

Gradient Magnitude and Direction

Computes the magnitude and direction for a pixel based on the change of pixel values of the neighboring pixels.

Grayscale

Grayscale version of an image.

Image from Camera

The hardware component that provides the initial image either in a streaming fashion (for hardware) or as a complete image (for software). The image is RGB888.

Normalize

A common image processing step that converts an image between the values of 0 to 255 linearly.

Pose Estimation

Process of the April Tag algorithm that determines the detected tag's homography relative to the camera. It provides the distance, x, y, z, roll, pitch, yaw.

PyApril Tag

A Python April Tag library provided by Swatbotics that was also built against [3] but with optimizations [10].

Ravven Tag CCA

An April Tag algorithm that builds off "Ravven Tag MagTheta" by moving "Edge Extraction", "Clustering", and "Segmentation" onto hardware to provide even greater optimizations.

Ravven Tag MagTheta

An April Tag algorithm that moves "Grayscale", "Normalize", "Gaussian Smoothing", and "Gradient Magnitude and Direction" portions of Olson's April Tag algorithm onto hardware, leveraging the Fusion 2 SoC to provide optimizations.

Segment Connection

Process of the April Tag algorithm that connects segments together in head-to-tail fashion where the gradient directions determine the winding of each segment.

Segment Orientation Correction

Process of the April Tag algorithm that corrects the orientation of a segment to abide by the winding rule proposed by [3] when performing Create Quads.

Segmentation

Process of the April Tag algorithm that creates line segments along the edges that were clustered together.

Chapter 7

Appendix

Table 7.1: Final Hardware Output Data Breakdown.

Byte	Description	Value
7	Grayscale	uint8
6	Kirsch North	uint8
5	Kirsch Northwest	uint8
4	Kirsch West	uint8
3	Kirsch Southwest	uint8
2	Segment Data	uint16
1		
0	Unused	–

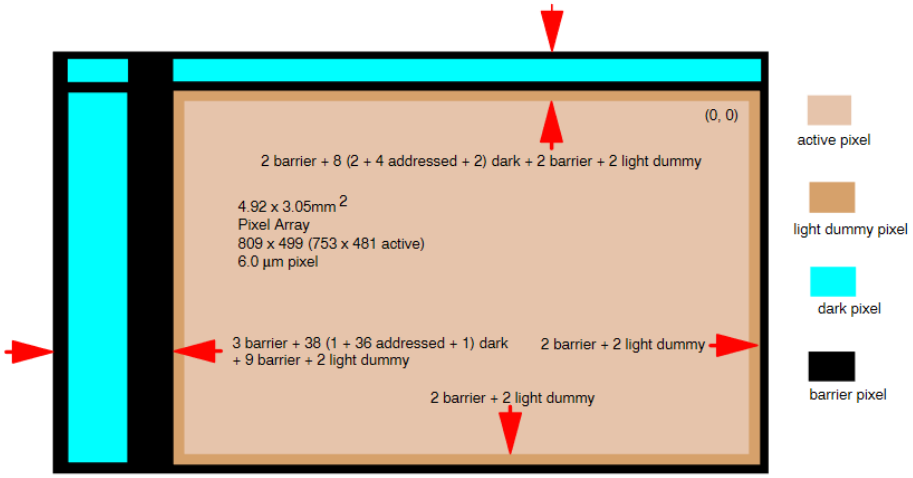


Figure 7.1: Pixel Array Description [1]

Table 7.2: Grayscale high-level resource consumption

High-Level Resource	Number
Multiplier	0
Adders/Subtractors	2
Registers	0
Total 1-Bit Register	0
RAMs	0
Multiplexers	0
I./O Bits	56
Static Shift operators	3
Dynamic Shift operators	0

Table 7.3: Gaussian Smoothing high-level resource consumption

High-Level Resource	Number
Multiplier	9
Adders/Subtractors	20
Registers	180
Total 1-Bit Register	2746
RAMs	2
Multiplexers	5
I./O Bits	246
Static Shift operators	0
Dynamic Shift operators	0

Table 7.4: Gradient Difference high-level resource consumption

High-Level Resource	Number
Multiplier	0
Adders/Subtractors	6
Registers	38
Total 1-Bit Register	794
RAMs	2
Multiplexers	6
I./O Bits	110
Static Shift operators	0
Dynamic Shift operators	0

Table 7.5: Kirsch Filter high-level resource consumption

High-Level Resource	Number
Multiplier	2
Adders/Subtractors	12
Registers	57
Total 1-Bit Register	326
RAMs	2
Multiplexers	6
I./O Bits	34
Static Shift operators	0
Dynamic Shift operators	0

Table 7.6: Kirsch Binarization high-level resource consumption

High-Level Resource	Number
Multiplier	2
Adders/Subtractors	2
Registers	34
Total 1-Bit Register	341
RAMs	0
Multiplexers	4
I./O Bits	76
Static Shift operators	0
Dynamic Shift operators	0

Table 7.7: Kirsch image statistics high-level resource consumption

High-Level Resource	Number
Multiplier	4
Adders/Subtractors	43
Registers	151
Total 1-Bit Register	2317
RAMs	0
Multiplexers	98
I./O Bits	50
Static Shift operators	12
Dynamic Shift operators	0

Table 7.8: Kirsch image statistics HDL utilization report summary

Resource	Utilization	Percentage
LUT	1036	1.95
LUTRAM	37	0.21
FF	1546	1.45
BRAM	0.5	0.36
DSP	7	3.18

Table 7.9: Kirsch image statistics HDL timing report summary

Target Frequency: 27 MHz	Setup (ns)	Hold (ns)	Pulse Width (ns)
Worst Negative Slack	27.933	0.090	17.538
Total Negative Slack	0.000	0.000	0.000

Table 7.10: Kirsch complete block high-level resource consumption

High-Level Resource	Number
Multiplier	8
Adders/Subtractors	57
Registers	418
Total 1-Bit Register	3204
RAMs	2
Multiplexers	108
I./O Bits	48
Static Shift operators	12
Dynamic Shift operators	0

```
if ((left ~= 0) && (above ~= 0))
    % merger
    if (left < above)
        assignment = left;
        mergerLookup(above) = left;

        maxX(left,:) = maxX(above,:);

        minX(above,4) = 0;
        maxX(above,4) = 0;
        minY(above,4) = 0;
        maxY(above,4) = 0;

        maxX(left,3) = theta(counter);
    elseif (above < left)
        assignment = above;
        mergerLookup(left) = above;

        minX(above,:) = minX(left,:);

        minX(left,4) = 0;
        maxX(left,4) = 0;
        minY(left,4) = 0;
        maxY(left,4) = 0;

        maxX(above,3) = theta(counter);
    else
        maxX(left,:) = maxX(above,:);
    end
elseif ((left ~= 0) && (above == 0))
    % take left id
    assignment = left;

    % only update the maxX point
    if (x > maxX(left,1))
        maxX(left,:) = [x,y,theta(counter),1];
    end
elseif ((left == 0) && (above ~= 0))
    % take above id
    assignment = above;

    % only update the maxY point
    maxY(above,:) = [x,y,theta(counter),1];
```

Figure 7.2: Finding the minimum x coordinate during CCA

Table 7.11: Kirsch complete HDL utilization report summary

Resource	Utilization	Percentage
LUT	1334	2.51
LUTRAM	73	0.42
FF	2049	1.93
BRAM	1.5	1.07
DSP	11	5.00

Table 7.12: Kirsch complete block HDL timing report summary

Target Frequency: 27 MHz	Setup (ns)	Hold (ns)	Pulse Width (ns)
Worst Negative Slack	28.464	0.038	17.538
Total Negative Slack	0.000	0.000	0.000