Fast Star Tracker Hardware Implementation and Algorithm optimisations on a System-on-a-Chip device.

by

Christiaan Lodewyk von Wielligh

Thesis presented in partial fulfilment of the requirements for the degree of Master of Engineering (Electronic) in the Faculty of Engineering at Stellenbosch University

Supervisor:	Dr. L. Visagie
Co-supervisor:	Mnr. A. Barnard

December 2019

BYLAE 1

UNIVERSITEIT·STELLENBOSCH·UNIVERSITY
jou kennisvennoot • your knowledge partner

Plagiaatverklaring / Plagiarism *Declaration*

1 Plagiaat is die oorneem en gebruik van die idees, materiaal en ander intellektuele eiendom van ander persone asof dit jou eie werk is.
Plagiarism is the use of ideas, material and other intellectual property of another's work and to present is as my own.

2 Ek erken dat die pleeg van plagiaat 'n strafbare oortreding is aangesien dit 'n vorm van diefstal is.
I agree that plagiarism is a punishable offence because it constitutes theft.

3 Ek verstaan ook dat direkte vertalings plagiaat is.
I also understand that direct translations are plagiarism.

4 Dienooreenkomstig is alle aanhalings en bydraes vanuit enige bron (ingesluit die internet) volledig verwys (erken). Ek erken dat die woordelikse aanhaal van teks sonder aanhalingstekens (selfs al word die bron volledig erken) plagiaat is.
Accordingly all quotations and contributions from any source whatsoever (including the internet) have been cited fully. I understand that the reproduction of text without quotation marks (even when the source is cited) is plagiarism.

5 Ek verklaar dat die werk in hierdie skryfstuk vervat, behalwe waar anders aangedui, my eie oorspronklike werk is en dat ek dit nie vantevore in die geheel of gedeeltelik ingehandig het vir bepunting in hierdie module/werkstuk of 'n ander module/werkstuk nie.
I declare that the work contained in this assignment, except where otherwise stated, is my original work and that I have not previously (in its entirety or in part) submitted it for grading in this module/assignment or another module/assignment.

Studentenommer / *Student number*	**Handtekening / *Signature***
CL von Wielligh **Voorletters en van / *Initials and surname***	December 2019 **Datum / *Date***

Abstract

Fast Star Tracker Hardware Implementation and Algorithm optimisations on a System-on-a-Chip device.

C. L. Von Wielligh

Department of Electrical and Electronic Engineering,
University of Stellenbosch,
Private Bag X1, Matieland 7602, South Africa.

Thesis: MEng (Elec.)

December 2019

Star trackers are instruments used onboard a spacecraft that utilize digital image sensors, optics and digital hardware to determine the inertial attitude of the spacecraft. Currently, these star trackers are the most accurate sensor system used onboard a spacecraft ADCS (attitude determination and control system). The majority of space missions requires high precision attitude determination which stresses the need for star trackers. Modern ADCS's, especially when making use of Control Moment Gyros (CMG's), demand fast update rates for increased agility. Star detection, a process where centroid locations are extracted from a star image, takes significant time in the star tracker pipeline. This is due to a large number of pixels that needs to be processed, causing a high computational burden on conventional microprocessors. We propose a solution where centroid extraction is implemented through novel design on an FPGA. This architecture makes it possible to extract centroid locations at the same time as the image data is streamed from the sensor. Such parallelization significantly increases the update rate of the star tracker without compromising accuracy or power usage.

The final design is implemented on a Xilinx Zynq SoC (System-on-a-Chip) device, which includes an FPGA and ARM processor. Tests are performed using simulated night sky images, real star images and a live sensor. Optimized matching algorithms are implemented on the processing system and validated independently. Distortion correction and QUEST is implemented, and a fully autonomous, end-to-end star tracker, with 10 Hz update rate is demonstrated under the night sky.

Uittreksel

Vinnige sterkamera hardeware implementering en algoritme optimiserings op ’n SoC toestel.

(“Fast Star Tracker Hardware Implementation and Algorithm optimisations on a System-on-a-Chip device.”)

C. L. Von Wielligh

Departement Electrical and Electronic Ingenieurswese,
Universiteit van Stellenbosch,
Privaatsak X1, Matieland 7602, Suid Afrika.

Tesis: MIng (Electronic)

Desember 2019

’n Sterkamera is ’n apparaat wat die oriëntasie van ’n satelliet bepaal deur gebruik te maak van ’n kamera sensor, optika en digitale hardeware. Hierdie sterkameras is tans die akuraatste sensors beskikbaar. Meeste satelliet missies vereis baie akkurate orentiasie bepaling wat die gebruik van sterkameras noodsaaklik maak. Moderne oriëntasie beheerstelsels, veral as daar van Beheer Moment Gyros gebruik maak, vereis ’n vinnige opdateringstempo. Ster deteksie, die proses waar ster sentroïedes se posisies vanaf ’n foto bepaal word, neem baie tyd as gevolg van ’n groot hoeveelheid spikkels wat geproseseer moet word. In hierdie navorsingsprojek stel ons ’n oplossing voor waar die deteksie deur vernuwende ontwerp in ’n FPGA geimplementeer word. Hierdie argitektuur maak dit moontlik ’n ster sentroïedes te onttrek op die selfde tyd as wat die beeld-data van die sensor oorgedra word. Deur die proses in parallel te doen, is daar ’n merkbare verbetering in die opdateringtempo van die sterkamer.

Die finale ontwerp is op ’n Xilinx Zynq SoC geïmplementeer. Toetse is gedoen met gesimuleerde fotos, regte ster fotos en ’n regte sensor. Die ster-identifiseering algoritme is verder geoptimeer en op ’n mikroverwerker geïmplementeer. Nadat distorsiekorreksie en QUEST toegepas is, word ’n outonome sterkamera met ’n opdateringspoed van 10 Hz gedemonstreer as ’n finale toets.

Acknowledgements

I would like to acknowledge the National Research Foundation (NRF) and Space Commercial Services (SCS) for their research funding through the Technology and Human Resources for Industry Programme (THRIP). Also, SCS for supplying hardware and support that helped complete the research.

I would also like to thank everyone at Stellenbosch University who has helped, specifically Dr. Lourens Visage and Mr. Arno Barnard for their support and expertise in satellite engineering.

Finally also my family, who has supported me throughout my academic career.

Contents

List of Figures

List of Tables

List of Abbreviations

2D	Two Dimensional
3D	Three Dimensional
ADCS	Attitude Determination and Control System
ADC	Analogue to Digital Converter
AXI	Advanced eXtensible Interface
CPU	Central Processing Unit
CAD	Computer Aided Design
CMG	Control Moment Gyro
DMA	Direct Memory Access
DEC	Declination
ECI	Earth Centred Inertial
FPGA	Field Programmable Gate Array
FIFO	First In, First Out
FOV	Field Of View
HDL	Hardware Description Language
HDMI	High Definition Multimedia Interface
ILA	Integrated Logic Analyser
LIS	Lost-In-Space
LVDS	Low Voltage Differential Signalling
PSF	Point Spread Function
QUEST	Quaternion Estimation Algorithm
RAM	Random Access Memory
RA	Right-Ascension
SAO	Smithsonian Astrophysical Observatory
SoC	System on a Chip
SPI	Serial Peripheral Interface
SNR	Signal to Noise Ratio

TTL	Transistor-Transistor Logic
TMR	Triple Modular Redundancy
UART	Universal Asynchronous Receiver Transmitter
USB	Universal Serial Bus
VHDL	Very High Speed Integrated Circuit Hardware Description Language

Chapter 1

Introduction

Satellite attitude determination plays a fundamental role in the majority of space missions. The information is vital for navigation in space, firing thrusters for manoeuvres and pointing antennas and scientific payloads to required positions. Various type of attitude-determining sensors, or combinations of them, are used onboard a spacecraft. Sun sensors, horizon sensors, magnetometers and star sensors are some of the more common ones. These instruments form part of an integrated attitude determination and control system (ADCS).

Star trackers are the most accurate instruments for attitude determination [3]. High precision scientific objectives that require fine pointing of payloads demand very accurate attitude control systems. This stresses the need for star trackers since other sensors are not capable of achieving these accuracies. Many space missions also require a certain level of agility for specialized tasks. Agile satellites capable of fast manoeuvres also considerably increases the amount of earth observation and space science mission data that can be collected, making them more efficient [19]. High-performance ADCS's especially when making use of Control Moment Gyro's (CMG's) improves the agility of a satellite but demands high update rates from attitude sensors. Striving to increase the operational envelope of a spacecraft motivates the development of accurate star trackers with fast update rates.

1.1 Problem Statement

Given a two dimensional image that contains only a small portion of the visible night sky, star trackers need to determine the attitude by performing pattern recognition algorithms on star constellations and mapping them to known star positions in a catalogue. This non-trivial problem is illustrated in figure 1.1. Large amounts of pixel data need to be processed during its operation, and the execution time of these image processing algorithms affects the update rate of the star tracker.

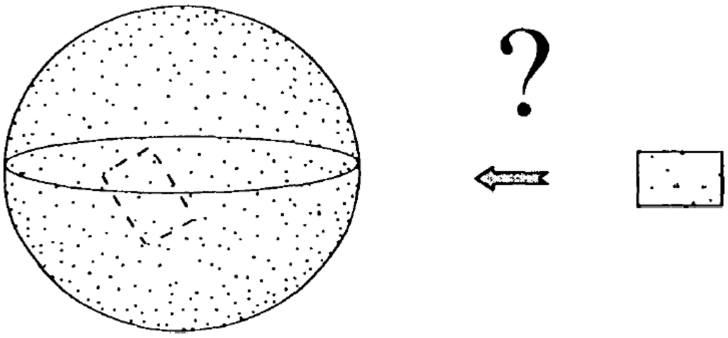


Figure 1.1: Star pattern recognition problem. A star tracker uses a photo of the sky and has to recognise star constellations. Image from Eisenman, A.R. [3].

Star trackers are made up of electronic image sensors, optics and digital hardware that process data. Figure 1.2 shows a diagram of a typical star tracker. Generally, a System-on-a-Chip (SoC) device that consists of an interconnected Field Programmable Gate Array (FGPA) and a microprocessor is used to interface with the sensor and processes the image data. A typical configuration would connect the FPGA directly to the image sensor to stream high-speed pixel data and save it in memory where the processor acess it to perform detection and matching algorithms.

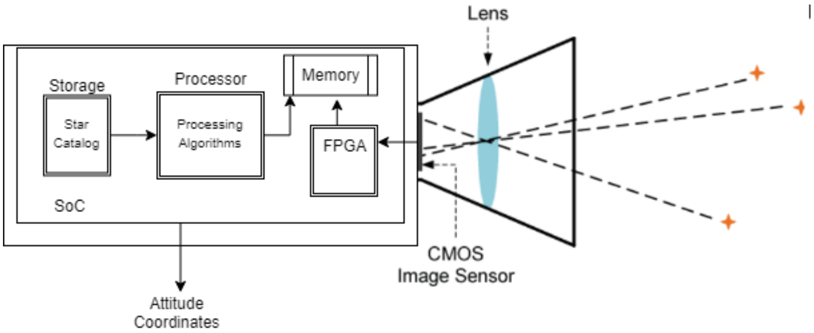


Figure 1.2: Block Diagram of a typical star tracker.

Steps in a sequential star tracker are shown in figure 1.3. Integration time (also sometimes referred to exposure time) is the time needed for the sensor to be exposed to light. For star trackers, this can be the longest process in the sequence. Data must be read-out from the sensor then centroids are extracted, positions are calibrated (lens distortion correction), catalogue matching and attitude determination are done, and finally, the results are passed to the ADCS. The processing time of this entire sequence determines the update rate of a star tracker.

It is unlikely to achieve update rates close to 10 Hz with the sequence as mentioned above. Interleaving the processes can produce faster update rates: While the star tracker processing is happening, the sensor can start with the integration of the next frame. A timing diagram is shown in figure 1.4 that demonstrates this interleaved processing.

Matching can be optimised by tracking modes and reduced catalogue searches [18]. Centroid detection can be optimised by tracking smaller regions around known stars using

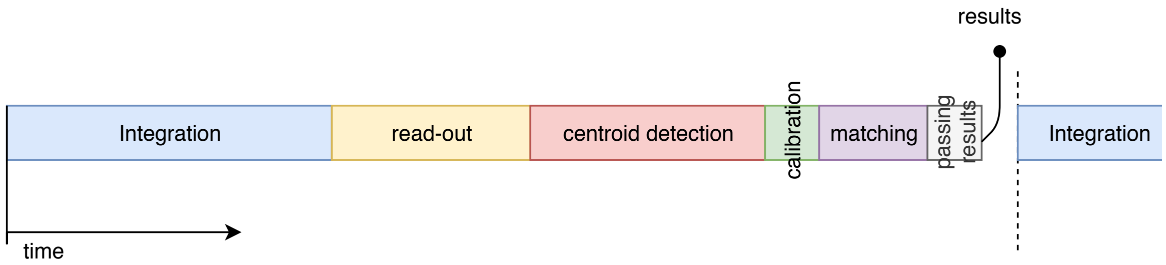


Figure 1.3: Steps in a sequential star tracker.

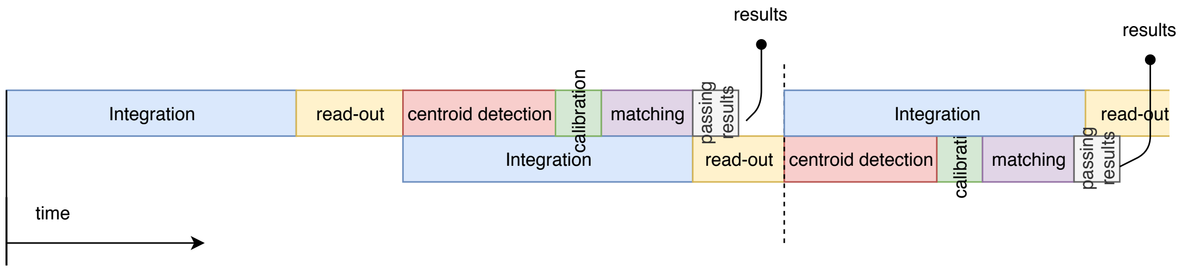


Figure 1.4: Interleaved processing of a star tracker.

priori attitude information, but even with this approach and in *lost-in-space* (LIS) situations, a large number of pixels have to be processed, resulting in a high computational burden for a conventional microprocessor.

We propose a solution where a parallel pipelined hardware architecture is designed on an FPGA to calculate the centroid locations at the same time as the pixel data is streamed from the sensor. Instead of saving the entire image in memory, only the centroid locations are passed to the processor, making it possible for the matching algorithm to start in parallel. Techniques that improve the matching algorithm speed is also implemented on the processor side of the SoC environment. This architecture significantly improves speed and makes image integration time the primary determining factor of the star tracker update rate. Figure 1.5 shows a timing diagram of the proposed solution.

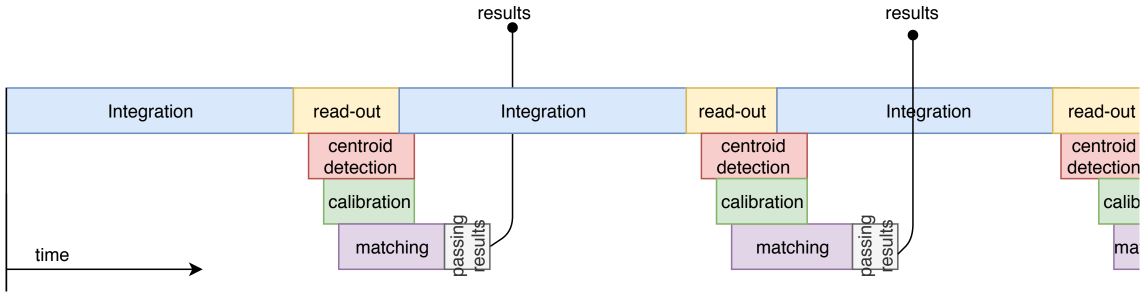


Figure 1.5: Interleaved processing with parallel hardware pipeline.

1.2 Project Objectives

Modern second-generation star trackers have accuracies in the arcsecond range, and update rates are typically 0.5 to 10Hz [28]. For high-performance ADCS's that use CMGs, an update rate of up to 10Hz is required. The hardware used to develop and test the star tracker algorithms for this project consists of a MicroZed 7020 development board that hosts a Xilinx ZYNQ SoC with some peripherals. The star images are streamed from a Gecko imager provided by Space Commercial Services (SCS). The imager acts as the camera front-end to complete the system and data is streamed from the Gecko using Low-Voltage Differential Signals (LVDS). The design objective for this project is, therefore, to optimise star detection algorithms on a SoC platform for a given sensor/optics combination that meets the following requirements:

- Accuracy error of less than 10 arcseconds across boresight.
- Update rate of at least 10Hz.
- Resource usage must not exceed available hardware.

Extensive testing is done for each sub-algorithm independently. An end-to-end test of the fast star tracker prototype demonstrates the integrity of the research.

1.3 Chapter Outline

Chapter 2 will cover the background information needed for the thesis. This includes star tracker characteristics, related star detection algorithms and basic theory on space navigation, lens distortion and FPGA's. Chapter 3 covers the star detection algorithm and its hardware implementation. Chapter 4 covers the matching (Star Identification) section of the star tracker. Lens distortion correction and attitude determination (QUEST) is covered as part of the full end-to-end system in chapter 5. The final results and analysis are discussed in the same chapter. Finally, the research conclusion is found in chapter 6.

Chapter 2

Background

A star tracker is a device that captures images of the night sky, detects possible star positions on the image, matches constellations on the image to known positions in a catalogue and determines the inertial attitude of the spacecraft.

2.1 Star Tracker Characteristics

Several parameters can characterize the performance of star trackers. A summary of selected characteristics follows:

Accuracy

Star detection algorithms and the overall design of the system affects the pointing accuracy. Liebe [20] provides an in-depth description of the accuracy performance measurement of star trackers. Accuracy measurement is essentially the angular distance error with respect to the boresight. It is common to quote accuracy in 1σ or 3σ values, and two measurements are given for commercial products - cross boresight (pitch and yaw) and about boresight (roll). Cross boresight errors are significantly less than roll errors, and modern trackers are capable of accuracies in the range of 0.1 to 20 arcseconds (cross boresight) [3]. To achieve this level of pointing accuracies, sub-pixel centroid detection is crucial. This is discussed further in chapter 3.

Update Rate

The update rate is dependant on two main factors: the exposure time of the sensor and the processing speed of the algorithms. Exposure time is highly interconnected with the rest of the optical and hardware design. Longer exposure times allow more photons on the sensor with better SNR but lowers the update rate. Choices regarding the sensitivity of the sensor, the aperture of the lens, the sky coverage and desired spacecraft dynamics all play a role in the exposure time. If a spacecraft is expected to experience high slew rates, the exposure time must be limited because stars may become smeared on the image plane, which decreases the centroid detection accuracy. Kazemi et al. [16] do present a method to extract accurate centroids during high slew rates, but it comes with a high computational cost and goes beyond the scope of this research. Various techniques to improve the processing speed of the algorithms has been considered and is the main focus of this thesis.

Physical Volume and Mass

The physical size and mass of a star tracker is an important consideration when comparing commercial products. High-performance star trackers such as the HAST by Ball Aerospace are massive instruments designed for large satellites [26]. It is safe to assume that they outperform smaller star trackers in most areas. The increasing market for microsatellites pushes designs towards smaller and lighter instruments. Research in this thesis focuses mainly on technology related to nano satellites.

Power Consumption

Similar to volume and mass, all satellites have a limited power budget that limits the hardware resources such as processors and memory that can be used. Cubesat compatible star trackers typically have a power consumption of less than 1 Watt on average. Larger star trackers can typically consume 10 W of power.

Other characteristics

Other parameters such as Field of View (FOV), sky coverage, sensor pixel resolution, SNR, catalogue size, analogue-to-digital converter (ADC) resolution, the number of stars tracked and maximum slew rate is used to describe a star tracker. These all directly affect the accuracy, speed, mass, volume and power consumption in a certain way due to the interconnected nature of the components. Depending on the design objective, preference is given to certain parameters above others.

2.2 Algorithms

Algorithms used by star trackers in literature and practice mostly follow the same flow of processes [10, 24, 32, 3, 14]. These processes are classified into four main steps. Figure 2.1 illustrates how these sub-algorithms work together in a flow chart.

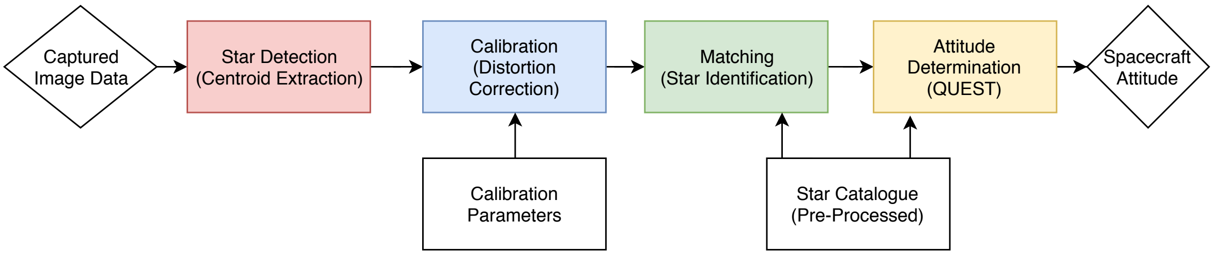

Captured Image Data
Star Detection (Centroid Extraction)
Calibration (Distortion Correction)
Matching (Star Identification)
Attitude Determination (QUEST)
Spacecraft Attitude
Calibration Parameters
Star Catalogue (Pre-Processed)

Figure 2.1: High level algorithm flow chart. The entire process can be described in four primary steps.

Star Detection

The first step is to extract the centroid positions of possible stars on the image plane. This thesis puts a strong focus on star detection as most of the optimisation is done in hardware during this step. Chapter 3 discuss star detection in detail.

Distortion Correction

After the star centroid positions are acquired, lens distortion correction is done and the image plane coordinates are converted to 3D sensor body coordinates in Cartesian form. Matching uses these vectors as input to the algorithm.

Matching

Patterns or features found in constellations are used to match each detected star to a known star position in a catalogue. This step requires more complex methods, and software-based optimisation techniques are used to improve processing speed, as discussed in chapter 4.

Attitude Determination

For the final step, these valid detected stars that are matched to known ECI coordinates that can be used to determine the spacecraft attitude. Algorithms such as Quaternion Estimation Algorithm (QUEST) are commonly used and is discussed in chapter 5.

2.3 Commercially Available Star Trackers

A list of efficient, commercially available star trackers (suitable for nano satellites) with high-performance standards follows below in table 2.1.

Manufacture	Sinclair Interplanetary	Hyperion Technologies	Blue Canyon Technologies	Space Astronomy Laboratory
Model Name	ST-16RT2	ST400	NST	ST5000
Update Rate	2 Hz	5 Hz	5 Hz	10 Hz
Accuracy (Cross-Boresight)	5 arcseconds	<10 arcseconds (3 sigma)	6 arcseconds	0.5 arcseconds
Accuracy (Roll)	55 arcseconds	<120 arcseconds (3 sigma)	40 arcseconds	17 arcseconds
Mass	0.16kg	0.28kg	0.35kg	4kg
Power	Peak 1W Avg <0.5W	Avg <0.7W	Peak <1.5W	12W
Volume	62 x 56 x 38 mm	53.8 x 53.8 x 90.35 mm	100 x 55 x 50 mm	NA

Table 2.1: Table comparing commercially available star trackers.

The instruments listed in the table shows that small form-factor star trackers can offer high performance in very efficient packages. The smallest and most lightweight, ST-16RT2 from Sinclair Interplanetary, weighs only 0.16kg and has a cross-boresight accuracy of 5 arcseconds. It is also the most power-efficient with average consumption down to 0.5W, making it ideal for Cubesats. The update rate is relatively slow (2Hz), but interestingly enough it performs a full lost-in-space (LIS) solution on each frame, leading to a zero initial acquisition time. The ST400 and NST trackers are slightly faster but also slightly larger. These update rates would also be listed for tracking mode, and both would require a LIS acquisition time of a few seconds. The most high-performance tracker in the list is the ST5000 designed by the Space Astronomy Laboratory at the University of Wisconsin. It is heavier (4kg), and less power efficient (12W) than the other trackers but is one of very few small-form star trackers with an update rate of 10Hz. They also report very high accuracies of up to 0.5 arcseconds. It is important to mention that the 10Hz update rate is only during tracking mode. The ST5000 is capable of autonomous LIS acquisition, but this process can take 1-10 seconds therefore the specified 10Hz update rate is not always true.

2.4 Related work in Star Detection

Liebe [20] gives an excellent overview of the operation and accuracy performance of star trackers. This reference covers the details on standard so-called second generation star trackers and proposes two modes of operation: 1). A fully autonomous Lost-In-Space (LIS) mode and 2.) a tracking mode where priori attitude information is used to optimise the processing and reduce redundant searching. This tracking mode is a common method used to increase the update rate of a star tracker. In tracking mode, a windowing technique is used to avoid searching through the entire image [33] when detecting centroids. Many matching optimisations in literature also require priori attitude information. Commercial star trackers would often claim fast update rates which is only true during tracking mode. LIS attitude acquisition is a slower process.

Knutson [17] worked on a fast centroiding algorithm using a tracking mode and windowed centroid extraction. Fast results are achieved but as discussed in chapter 5.7, it is still unlikely to achieve 10 Hz update rate using his techniques.

Erlank [10] developed a Cubesat-compatible star tracker using efficient low power hardware. In his thesis, he described some algorithm optimisation techniques. His end product was still relatively slow with an update rate of 1Hz. His thesis provides good algorithm descriptions which helped form some of the baseline software tests. He also covers a detailed system design in the thesis.

Lindh [21] presented a novel star detection method on an FPGA. His rotating FIFO (First In, First Out) approach was valuable to the work in this thesis. The rotating FIFO takes advantage of the fact that pixels are streamed in from left to right for each row. Blobs of pixels (possible stars) are managed by queueing and de-queuing values to a single FIFO. At the start of each new line, the blob with the lowest x-value (right-most on the image) is always at the front of the queue (FIFO). His design, however, was subject to constraints and timing issues which are improved in this research. More details are discussed in chapter 3. He did not implement matching algorithms and therefore, could not present an end-to-end system that demonstrates a certain update rate. His tests were also performed with relatively slow input data (4 frames per second).

Zhou et al. [36] published a paper in 2014 that demonstrates a fast centroid extraction algorithm on FPGA. They present a two-step method where first a maximum pixel-level centre is located through zero crossing of the first derivative in a small region. Using a fixed window weighted centroid method with the pixel-level centre as the middle of the window, the sub-pixel offsets to the sub-pixel centre can be calculated. The sub-pixel centre is then obtained by adding the offsets to the pixel-level centre. This was implemented on FPGA with test images. Their results concluded that the two-step method had roughly the same accuracy than the simpler threshold method. Only when high noise values were present in the image did their algorithm outperform. This approach seemed overly complex compared to [21] and was not further investigated.

2.5 Space Navigation and Coordinate Systems

Navigation requires a fixed reference coordinate system. In space, what is up and what is down might become less clear than on the surface of the Earth. Three different coordinate systems are used throughout this thesis and conversions between the coordinate systems are made to determine a spacecraft attitude. The most important coordinate system used for space navigation is the Earth-centred Inertial (ECI) coordinate system.

Earth-centred Inertial

Some ancient civilizations assumed that Earth was fixed and at the centre of the universe [25]. They imagined that all stars were attached to a celestial sphere, revolving about the Earth in one day. While it is known now that this is not true, the concept of a celestial sphere still holds great value as a tool for navigation in space. Because stars and other astronomical objects are at such great distances away, it becomes valid to assume that for an observer on (or close to) Earth, they all seem *equally* far away, as if fixed or projected onto the inside of a sphere with a large but unknown radius.

The ECI coordinate system is fixed in inertial space and oriented using the Earth's orbit plane. The x-axis is the Vernal Equinox and is defined in the direction of the point where the Earth's equator intersected the Earth's orbit plane at 12:00 on 1 January 2000 (J2000 ECI coordinate system). The z-axis is defined pointing from the centre of Earth towards the north celestial pole.

Because celestial objects can be assumed to be at an arbitrary unit radius away, any position on the sphere can be expressed using only two coordinates: Right Ascension and Declination. Right Ascension is the angle measured from the vernal equinox and declination is measured from the celestial equator. This ECI coordinate system is better illustrated in figure 2.2.

Image Plane Coordinate System

Image plane coordinates are two-dimensional coordinates on the image sensor surface. This is typically expressed in pixels for digital images, and the origin can vary, depending on the application. The true origin is the point where the boresight axis intersects the image plane, near the centre of the sensor. In software processing algorithms, however, it is typically defined as the first pixel in the top-left corner of the sensor.

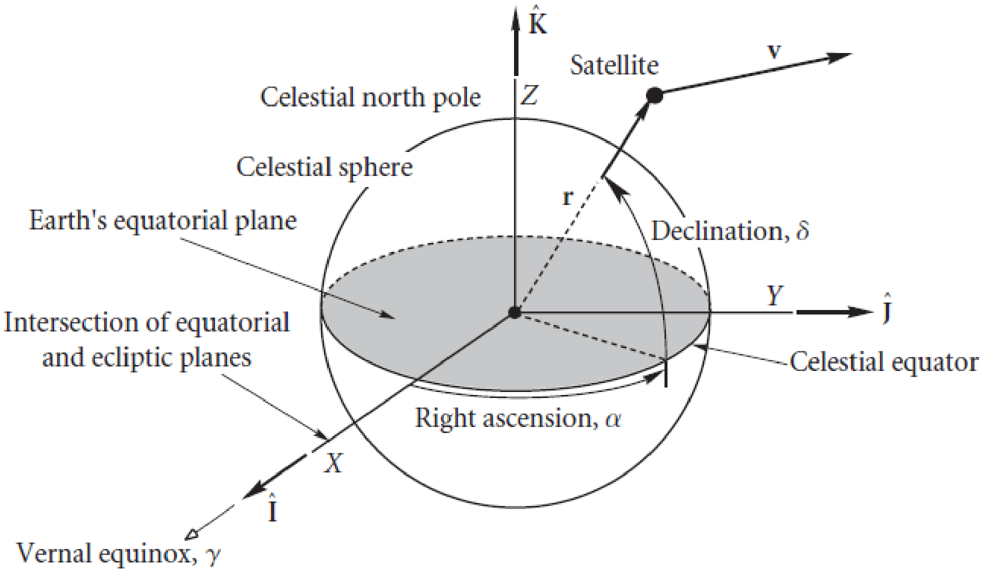


Figure 2.2: Diagram showing the ECI coordinate system. The Earth can be considered a point at the origin, and all stars are at a fixed distance away on a sphere. The location of any star can be expressed using only Right Ascension (α), and Declination (δ) as their radius from Earth (r) is arbitrarily far. Image is taken from website with unknown author[2].

Sensor Body Coordinate System

Sensor body coordinates are related to the camera viewpoint in 3D space. The origin is at the optical centre of the lens. The z-axis (or principal axis) is aligned with the camera lens boresight. The focal distance of the lens becomes the z-distance from the origin to the image plane. Image plane coordinates of stars found in the star detection algorithm is converted to these sensor body coordinates to find the angular separation between stars. These coordinates are expressed in Cartesian form. Figure 2.3 better illustrates this coordinate system.

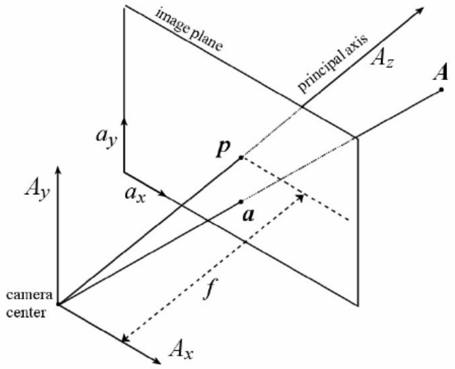


Figure 2.3: Diagram showing how the image plane is related to the sensor body coordinate system. Notice how the focal distance is the z-distance from the origin along the boresight (principle axis) to the image plane. Image taken from Liu et.al. [22]

2.6 Lens Distortion

In reality, there is no such thing as an ideal pinhole camera, and every lens causes some distortion to the image due to variations in image magnification. This optical aberration needs to be characterized and corrected in software before pixel positions can be used for matching algorithms. Lens distortion is *mostly* radially symmetric and can be classified as either barrel distortions or pincushion distortions [29]. Figure 2.4 shows these two types of radial distortions.

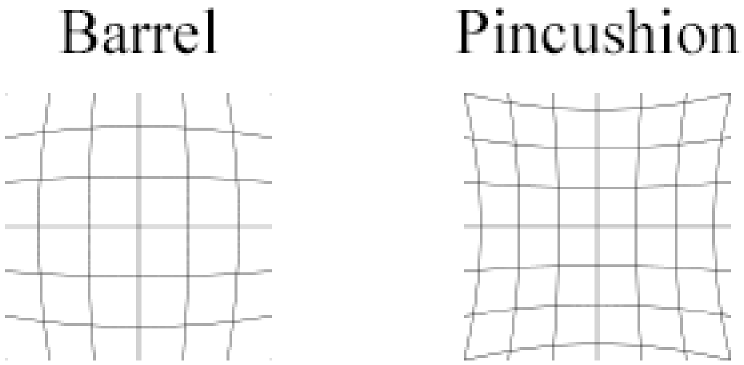


Figure 2.4: Two types of radial distortion. If a point appears closer to the optical centre than it should, it is referred to as barrel distortion whereas an outward displacement is pincushion distortion.

Brown's distortion model is a popular mathematical model used to characterize lens distortion using a series of polynomial coefficients. An in-depth evaluation of different lens distortion models can be found in a paper by Ricolfe-Viala et al. [29]. These coefficients can sometimes be supplied by the lens manufacturer, but in most cases, it is necessary to calibrate the camera and lens combination through an iterative algorithm or other technique since the sensor is not perfectly aligned to the optical axis. More details on distortion correction and calibration is found in section 5.1.

2.7 FPGA Overview

The core of this thesis focus on utilizing the parallel processing power of an FPGA to increase star detection speed significantly. FPGA's are field-programmable gate array devices containing hundreds of thousands of logic gates, flip-flops, DSP blocks and other logic units that can be configured to implement nearly any digital logic circuit. Computer-Aided Design (CAD) tools are used to map user designs to FPGA's, adding a layer of abstraction between the hardware and the designer. Designs are written in Hardware Description Language (HDL), and the CAD-flow converts the design into a stream of bits that are used to program the FPGA. The Xilinx Vivado suite is used as a CAD environment, and the flow for designing custom hardware follows below.

1. **HDL Code**. The designer writes HDL code either in VHDL or Verilog that describes the functionality of the hardware. This includes all ports, logic, local registers (signals), arithmetic blocks and other hardware. Signal assignments happen

concurrently; therefore, clock-triggered processes are written to build a synchronous system.

2. **Behavioural Simulation**. During behavioural simulation, test-benches are written, and the logical functionality of the design is simulated and tested. This is pure software simulation, and the tool does not consider any routing, timing delays or hardware constraints. The design can be verified by having the test bench check output based on known inputs or by letting the test bench write the output to files on a PC. It can also be de-bugged on a lower level by looking at a waveform diagram that shows register and wire-values at any given time. The test-benches act as stimuli, giving the designer the chance to test different input scenarios.

3. **Logic Synthesis**. This transforms the HDL description into a set of boolean gates, Flip-Flops and other FPGA platform specific hardware blocks like memory, Digital Signal Processors and more. The routing connection (netlist) between all of the logic is also synthesised.

4. **Implementation**. The synthesized design must now be mapped to specified hardware. The CAD tool optimizes the design, places the design and routes all of the connections to form a complete circuit.

5. **Bitstream Generation** The implemented design is exported as a bitstream. This bitstream is written to the FPGA which configures the entire device.

Figure 2.5 illustrates this FPGA development flow. Behavioural simulation gives the designer ability to validate HDL logic without synthesizing and implementing the hardware. The synthesis and implementation can take a significant time (10 to 30 minutes) for large designs, making hardware changes and debugging a painstakingly slow process. Test-benches are therefore written to test most cases of the HDL in simulation to avoid as much hardware debugging as possible.

Since the abstraction layer in the toolchain manages the hardware mapping, the designer must know (to a certain extent) what the HDL code is actually creating. After implementation, the designer can inspect the entire circuit diagram that has been generated by the CAD tool. Although overwhelming in size, it is often valuable to help the designer think in terms of flip-flops instead of variables. Figure 2.6 demonstrates how VHDL code is converted to a circuit by the CAD workflow. The code is a very basic synchronous adder that adds two 3-bit numbers onto a 4-bit output. Notice how the adder is implemented into lookup-tables (LUTs) instead of a classic adder logic circuit.

Timing considerations are critical when working in FPGA since the hardware is synchronous and driven by clocks. In a later section, it is shown how crossing multiple clock domains becomes complex and may lead to unexpected behaviour.

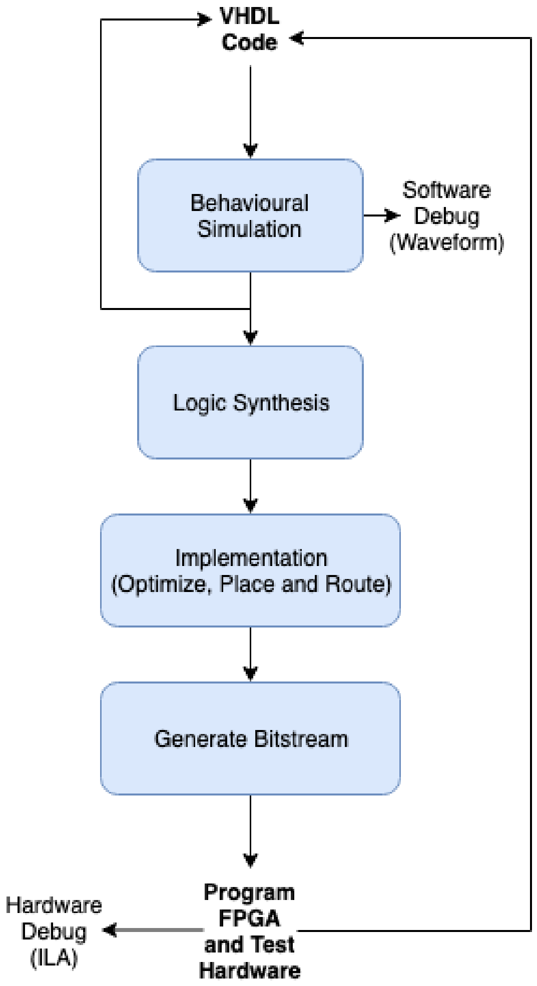
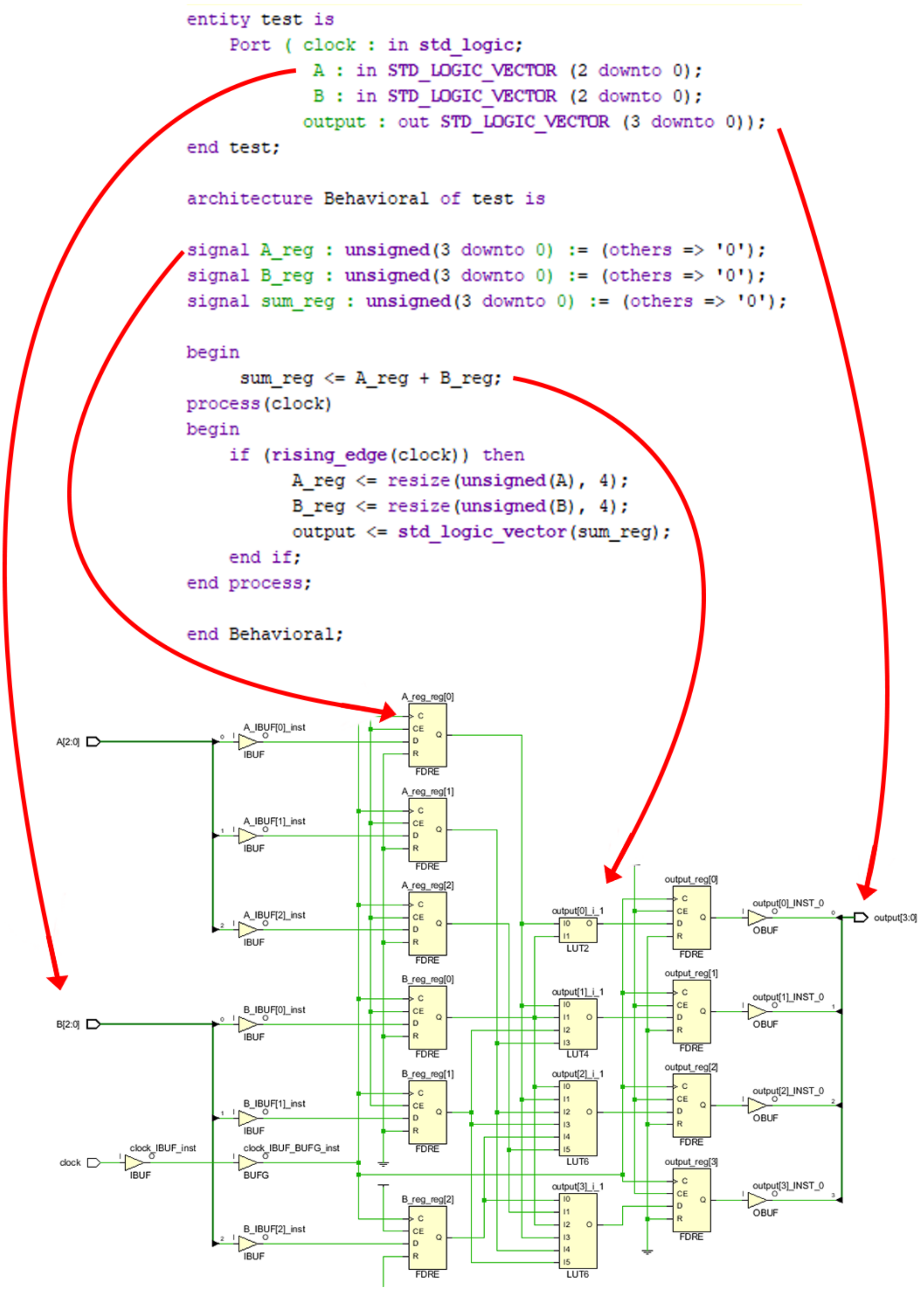

VHDL Code
Behavioural Simulation
Software Debug (Waveform)
Logic Synthesis
Implementation (Optimize, Place and Route)
Generate Bitstream
Hardware Debug (ILA)
Program FPGA and Test Hardware

Figure 2.5: FPGA development workflow.



```
entity test is
    Port ( clock : in std_logic;
           A : in STD_LOGIC_VECTOR (2 downto 0);
           B : in STD_LOGIC_VECTOR (2 downto 0);
          output : out STD_LOGIC_VECTOR (3 downto 0));
end test;

architecture Behavioral of test is

signal A_reg : unsigned(3 downto 0) := (others => '0');
signal B_reg : unsigned(3 downto 0) := (others => '0');
signal sum_reg : unsigned(3 downto 0) := (others => '0');

begin
     sum_reg <= A_reg + B_reg;
process(clock)
begin
    if (rising_edge(clock)) then
          A_reg <= resize(unsigned(A), 4);
          B_reg <= resize(unsigned(B), 4);
          output <= std_logic_vector(sum_reg);
    end if;
end process;

end Behavioral;
```

Figure 2.6: VHDL to Synthesis example. he code is a very basic synchronous adder that adds two 3-bit numbers onto a 4-bit output. Notice how the adder is implemented into lookup-tables (LUTs) instead of a classic adder logic circuit.

2.8 Hardware Overview

There are fundamental interdependencies between components of a star tracker that complicates the system design. This project focuses on algorithm optimisations and does not discuss the system design in depth. For detail design discussion, there are many resources available in literature [24, 10, 32]. Hardware used in the project consists of a MicroZed development board that hosts a Xilinx Zynq 2070 SoC device. All of the algorithms are implemented on this board. Centroid detection is done directly on the FPGA, whereas the rest of the algorithms takes place on the processor system side. The imager used for an end-to-end test is the Gecko imager from Space Commercial Services. The Gecko was used based on availability.

2.8.1 Imager Details

The Gecko imager provides 2 serial pixel channels through LVDS along with a control line. The maximum LVDS clock speed is 200MHz. It is set up to stream images of 2048x1088 pixel resolution. The exposure and gain setting is configured using Serial Peripheral Interface (SPI) commands.

2.8.2 SoC Details

The Zynq 7020 SoC block design is shown in figure 2.7. The Zynq processor has two ARM Cortex-A9 CPU's along with vast I/O peripheral options, memory, clock generating circuits, memory controllers, and AXI interface ports between the configurable FPGA circuit (referred to as the Programmable Logic (PL)) and processor (referred to as the Processing System (PS)). The PS generates the clock that drives the PL logic circuit. Data is transferred between the PL and PS using AXI ports or DMA channels.

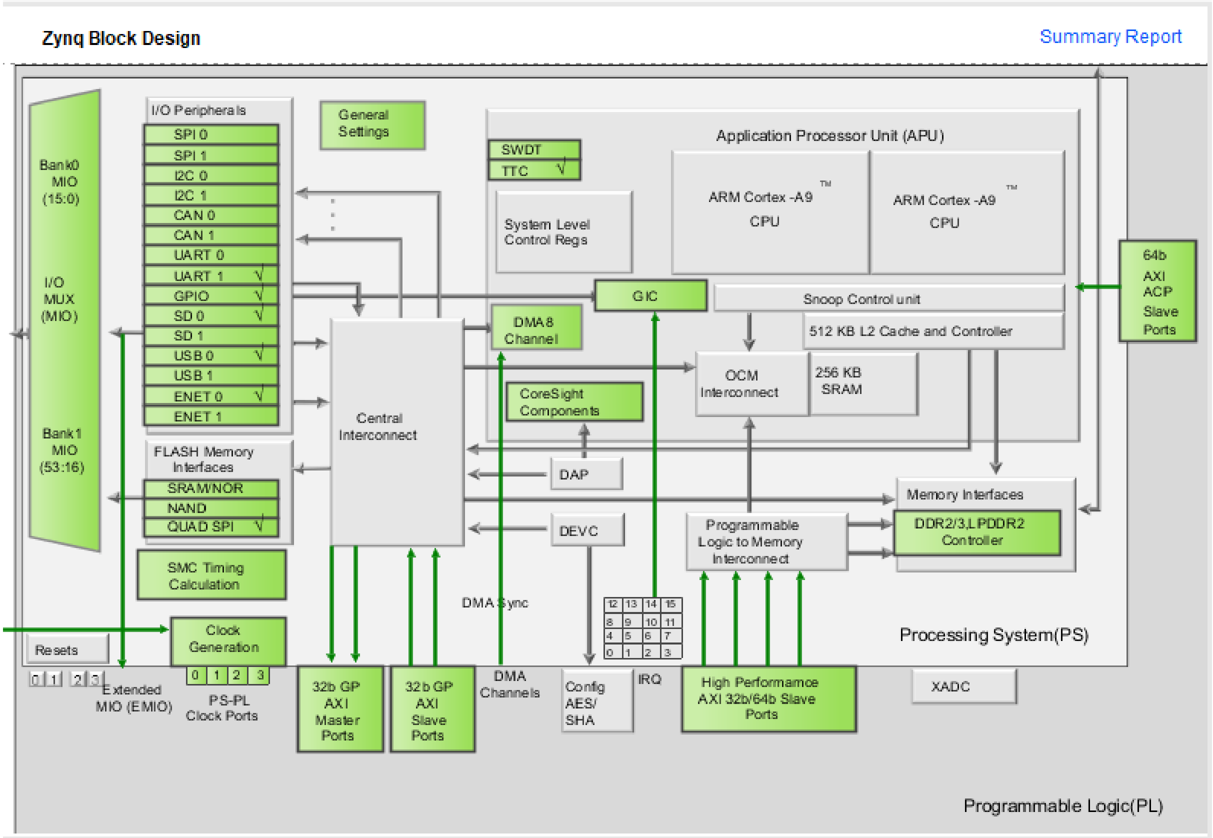


Figure 2.7: Zynq 7020 SoC Block Design.

Chapter 3

Star Detection

Star detection is the first processing step in a star tracker operation. It is during this step where hardware optimisation is done to improve speed significantly. It is vital to design a system that is capable of sub-pixel centroid detection accuracy and robust to detector noise. Any error in centroid detection is propagated down the algorithm pipeline and reduce the overall accuracy of the star tracker.

As pointed out in paragraph 2.4, many star trackers have two modes of operation: 1) LIS mode and 2) Tracking mode. The hardware optimisation technique discussed does not implement a tracking mode since the parallel detection in LIS is proved to be fast enough for a 10Hz update rate (chapter 5.7).

3.1 Stars on the image sensor

A star on an image sensor can be considered a point light source for all practical purposes. With an ideal pinhole camera, the star might appear as a single-pixel which, counter-intuitively, leads to less accurate centroid detection. In reality, an ideal pinhole camera does not exist, and image formation through a lens can be described by a Point Spread Function (PSF). Depending on the aperture size and focus setting, the degree of spreading (blurring) of the point light source changes. Spreading introduces additional information about the star as it spreads the light source over several pixels. This allows centroiding algorithms to determine the centre of a star to sub-pixel accuracy. Depending on the intensity of the light source, defocusing is done to have stars spread over 3x3 to 5x5 pixels (typically). Theory on how the PSF is formed and how light is spread through the defocussed optics can become rather complex [10], but for the purposes of a star tracker, it can be accurately approximated as a 2D Gaussian distribution. Figure 3.1 shows a zoomed-in image of a single star that is slightly defocused.

3.2 Traditional Star Detection Approach

It is a common approach for star trackers to use a centre-of-mass (or weighted average) algorithm to determine star centroid locations [20]. Literature suggests the following steps: First check each pixel in the image and compare it to a threshold. This threshold-value can be pre-defined or dynamically adjusted. If the pixel value is above the threshold, a region of interest is identified. Neighbouring pixels that are above the threshold are

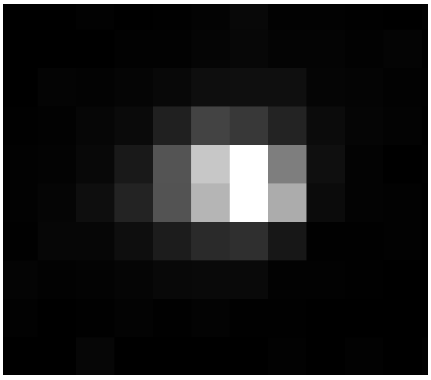


Figure 3.1: Visual example of a real star region on a star photo. The image is zoomed in to see individual pixels. Notice how the star is spread over multiple pixels due to optical defocusing. It may seem valid that a 2D Gaussian distribution can approximate the PSF.

clustered together. Once all of the neighbouring pixels are grouped, and the region is considered valid, a centre-of-mass calculation is performed on the region pixels.

In this work, the optimised algorithms follow the same logic as the traditional approach but are adapted for inline streaming and hardware-friendly calculations. Data structures, timing and data-flow, is the biggest challenges (and changes) when working in hardware. Traditional centroiding algorithms and their detail are first discussed, followed by a novel adaption in hardware.

3.3 Algorithm Details

3.3.1 Image Plane Search and Region Growing Algorithm

The first step is grouping together regions of pixels above the threshold. The Region Growing Algorithm described by Erlank [10] and used on SUNSAT [32] is used to form the baseline tests for later comparison. In short, it searches through the image and creates a list of valid regions (groups of pixels) that might be stars. These regions can be of arbitrary shape and only contain pixels above the threshold. Some conditions are introduced during the growing algorithm such as an allowed minimum and maximum pixels per region, margins of allowable gaps in between consecutive pixels in a region and an X- and Y-crop factor of the image. Too many pixels in a region might indicate invalid light sources such as planets, the moon or reflections from the sun while with too little pixels it becomes hard to distinguish between a star and noise or sensor artefacts. The crop factor helps to avoid searching for stars on the edge of the image where lens distortion is at the maximum, making centroid detection less accurate. Figure 3.2 shows a simplified example of how regions are found in a star image.

3.3.2 Centroid Calculation

As mentioned in the previous section, a centre of mass algorithm is very commonly used for centroid calculations in star trackers. It is simple to implement and not very processor-intensive compared to other methods such as Gaussian curve fitting [10]. Although less accurate than Gaussian curve fitting [18], the centre of mass equation can have centroid

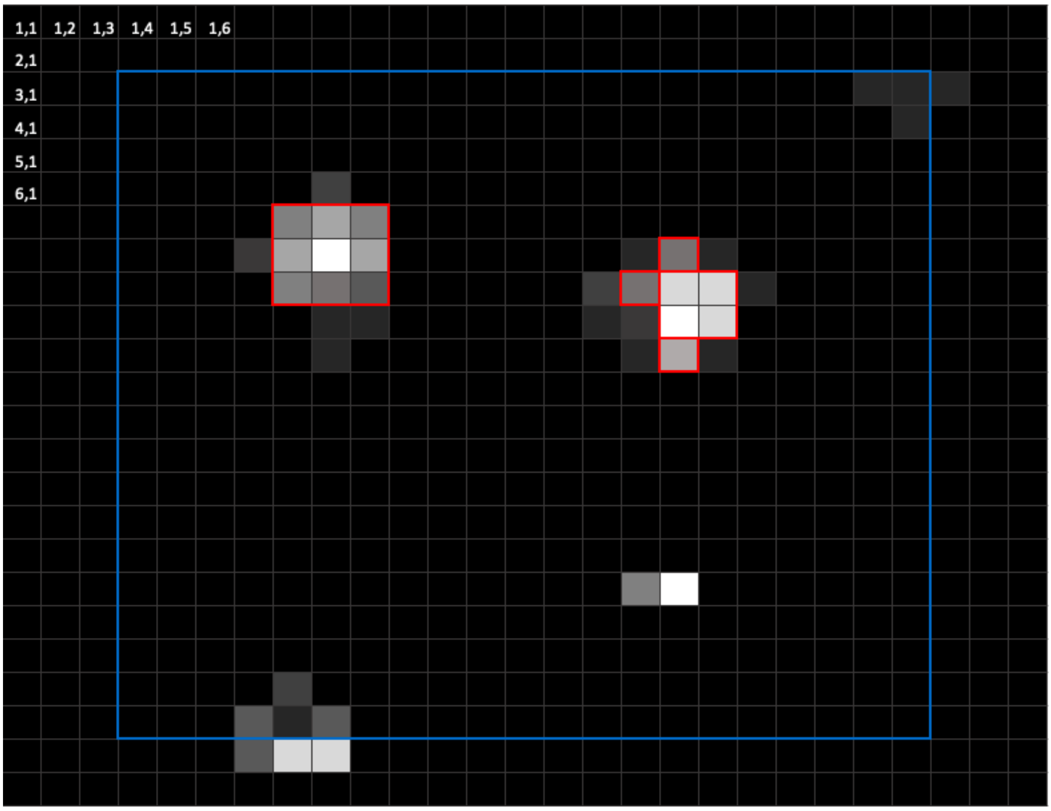


Figure 3.2: Simple illustration of a list of regions. The red borders indicate valid regions and the blue line indicated the crop factor.

accuracies within 0.2 pixels which is adequate for our purposes [32]. Equations 3.1 and 3.2 describe the centre of mass equation that is applied to each region.

$$x_{\text{centroid}} = \frac{\sum_{i=0}^{\text{totalpixels}}[(\text{pixel}[i]_x) \times (\text{pixel}[i]_{\text{intensity}} - \text{threshold})]}{\sum_{i=0}^{\text{totalpixels}}[\text{pixel}[i]_{\text{intensity}} - \text{threshold}]} \tag{3.1}$$

$$y_{\text{centroid}} = \frac{\sum_{i=0}^{\text{totalpixels}}[(\text{pixel}[i]_y) \times (\text{pixel}[i]_{\text{intensity}} - \text{threshold})]}{\sum_{i=0}^{\text{totalpixels}}[\text{pixel}[i]_{\text{intensity}} - \text{threshold}]} \tag{3.2}$$

where

$$x_{\text{centroid}} = \text{horizontal centroid position on image plane} \tag{3.3}$$

$$y_{\text{centroid}} = \text{vertical centroid position on image plane} \tag{3.4}$$

$$\text{totalpixels} = \text{total number of pixels found in the region} \tag{3.5}$$

Not all centre of mass algorithms subtracts the threshold from the accumulated values, but this helps to reduce the numerical size of values during computation.

3.4 Fast Star Detection Hardware Design

There are a few things to consider when adapting the algorithm for real time detection. First, the complete image is not available in memory for random access, making the Region Growing Algorithm [10] not possible to implement. Instead, regions has to be grown and identified as the pixels are streamed in one-by-one. The second thing to notice is that the centre of mass equation is a single divide of two summations. The numerator ($\sum_{i=0}^{\text{totalpixels}}[(\text{pixel}[i]_x) \times (\text{pixel}[i]_{\text{intensity}} - \text{threshold})]$) and the denominator ($\sum_{i=0}^{\text{totalpixels}}[\text{pixel}[i]_{\text{intensity}} - \text{threshold}]$) can be accumulated for each region as more pixels are added to it. This effectively merges the Region Growing Algorithm and (most of) the centre of mass calculation into one process. It is never necessary to save the individual pixel values of the region, only the accumulated intensities and boundary positions of the region needs to be stored in registers. The divide calculations can be done in a single step once a valid region is found.

If a pixel value is higher than the threshold, a region is initialized. For each region, a set of variables are needed. These variables are listed below and visually described in figure 3.3.

- **Start Y**. This is the starting row number of the region.
- **Min X**. This is the minimum pixel x-position of the region.
- **Max X**. The rightmost x-position of the region.
- **Number of Rows**. The number of rows that the region currently have.
- **Cumulative Intensity**. This is the denominator part of the center of mass equation and is updated as the region is expanded.
- **X-Sum**. This is the numerator part of center of mass equation for calculating the x-coordinate of the centroid.
- **Y-Sum**. This is the numerator part of center of mass equation for calculating the y-coordinate of the centroid.
- **Number of Pixels**. This is the number of pixels currently in the region. It is used to determine whether a region is valid or invalid on closure.

The rules for growing a region is simple. If a pixel with an intensity greater than the threshold is detected and the x-position is within the range of the current (`minX` - margin) and (`maxX` + margin) then it is added to the region and the relevant values are updated. If an entire row passes where no pixels were added to the region, e.g. the current row is greater than `startY` + `numberOfRows` then the region is closed. If the number of pixels in the region at this point is within the allowed range then two divide calculations are performed to determine the centroid positions. Multiple stars in a row is handled by a rotating data-flow design better explained in section 3.5.4.

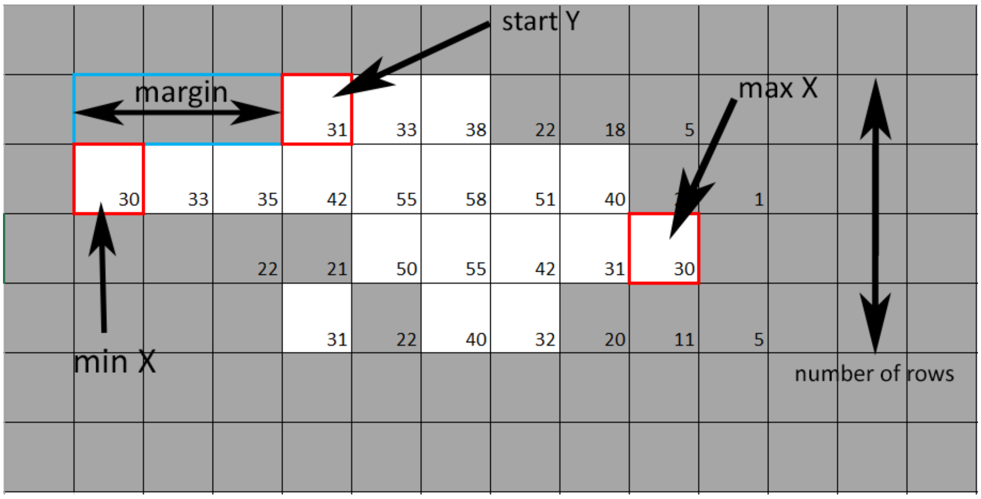


Figure 3.3: Visualization of variables in an active region. The numbers in the blocks indicates pixel intensities (not coordinates) and a threshold of 30 is assumed for illustration purposes.

3.5 Hardware Design

Translating the algorithm to a parallel pipelined hardware architecture requires thoughtful planning and good design. The star detection pipeline is split into five main processes which are better illustrated in figure 3.4. The system operates with three clock domains: Serial Clock, Pixel Clock and System Clock. The first two blocks are specific to the imager used in the star tracker. The design can, however, become generic to any sensor/imager by only changing these two blocks to match the interface and streaming protocol of the specific camera.

3.5.1 Simplified Hardware Architecture

1. **Serial to Parallel conversion**. Although technically not part of the algorithm, this block receives serial data from the sensor (at the serial clock speed), synchronises with the sensor and clocks out two channels of 8-bit parallel pixels at the pixel clock rate. It also generates the pixel clock, which is a division of the serial clock.

2. **Pixel Split and Count**. This block receives two 8-bit pixel channels coming in at pixel clock speed, counts the number of pixels and keeps track of when a new row occurs. It outputs 20 bit parallel data containing a single pixel value (bit 7 downto 0), the x-count (bit 18 downto 8) and a new line indicator (bit 19). Each pixel with its count data is pushed to a FIFO that makes crossing over to the system clock domain possible. We assume that the system clock is always faster than the pixel clock. More details around timing are discussed in section 3.5.7.

3. **Threshold and Margin Counter**. As soon as data is available on the 20 bit FIFO, this block reads the data at the system clock speed and compare the pixel value to the threshold. Depending on the pixel value, the x-count, and other parameters this block outputs message signals such as ‘valid’, ‘end-of-line’ or ‘end-of-region’ along with the pixel data to the next processing block. Not all pixels are passed on; only the ones that are used to create or update regions. More details on the design are discussed in section 3.5.3.

4. **Region Manager/Updater with rotating FIFO**. This is the most complex part of the pipeline and manages the data structures and flow of growing regions. Many iterations on the design were considered, developed and tested, and finally, a rotating FIFO approach adapted from Lindh [21] was used as it is the most dynamic and low resource solution. This is further discussed in section 3.5.4.

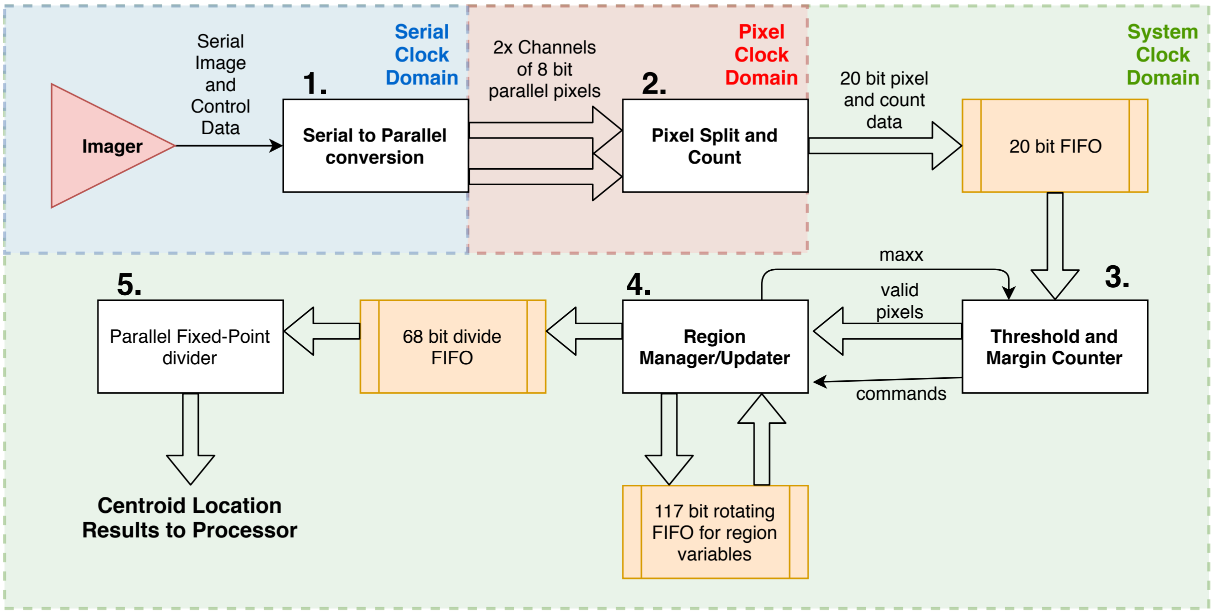

Serial Clock Domain
Pixel Clock Domain
System Clock Domain
Imager
Serial Image and Control Data
1.
Serial to Parallel conversion
2x Channels of 8 bit parallel pixels
2.
Pixel Split and Count
20 bit pixel and count data
20 bit FIFO
3.
Threshold and Margin Counter
maxx
valid pixels
commands
4.
Region Manager/Updater
117 bit rotating FIFO for region variables
68 bit divide FIFO
5.
Parallel Fixed-Point divider
Centroid Location Results to Processor

Figure 3.4: Simplified block diagram of hardware architecture. This illustrates the 5 main processes along with different clock domains.

3.5.2 Pixel Split and Count

This IP block in the hardware pipeline is responsible for receiving two channels of 8-bit pixel data from the sensor interface, counting the pixels and keeping track of when a new line occurs. The input ports include the system clock, pixel clock and parameters such as the image x- and y-dimensions as well as the x and y crop factor. These parameters are updated internally on each reset. All of the ports can be seen in figure 3.5. The data structure of the 20-bit output that is pushed to a FIFO is described in table 3.1. 11 bits are chosen for the x-count since $2^{11} = 2048$ which is the maximum x-dimension the design allows. This is a valid assumption since most star trackers have lower resolution than 2048 pixels wide. In any case, the algorithm can be easily adapted for higher resolution images.

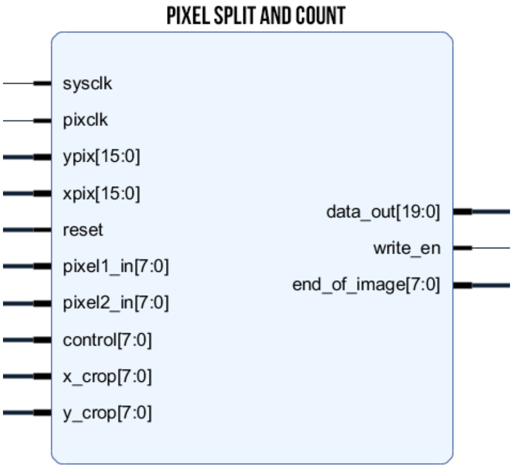


Figure 3.5: IP Block of pixel split and count.

Bit	Description	Unsigned Range
7 downto 0	pixel value	0 - 255
18 downto 8	x-count value	0 - 2048
19	new-line indicator	0 - 1

Table 3.1: 19 bit FIFO data structure

The FIFO next in line is running at the system clock speed, but the incoming pixel data are transferred at the pixel clock speed. This is a situation where care must be taken crossing from the slower (pixel clock) domain to the faster (system clock) domain. When latching data from a slow flip-flop to a faster flip-flop, violations of setup and hold time occur, making the data subject to metastability. A simple fix to the problem is to ”double-flop” the data. The output of the faster Flip-Flop is re-registered on a second Flip-Flop in the fast clock domain. This second Flip-Flop now has a stable output. The double-flopping technique is better illustrated in figure 3.6. An edge-condition of the pixel clock needs to be detected. Therefore the stable output is re-flopped to compare two registers containing the pixel clock. Below is an example of a VHDL process that shows how slow clock edges can be detected in a faster process.

```
-- VHDL Example
process (system_clock)
begin
        if rising_edge(system_clock) then
-- pixClk_reg1 is METASTABLE, pixClk_reg2 and pixClk_reg3 are STABLE
                pixClk_reg1 <= pixClk;
                pixClk_reg2 <= pixClk_reg1;
                pixClk_reg3 <= pixClk_reg2;

                if(pixClk_reg3 = '0' and pixClk_reg2 = '1') then
                        -- Positive Edge Condition
                end if;
        end if;
end process;
```

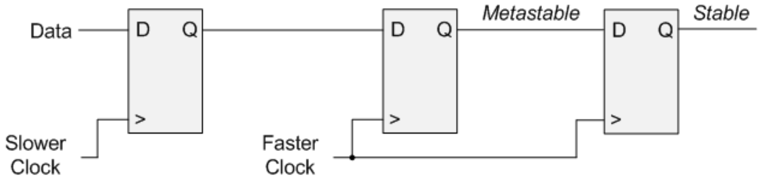


Figure 3.6: Double-flopping: Crossing from a slow to a fast clock domain.

It is crucial that the system clock is faster than the pixel clock since two pixels must be processed and pushed to the FIFO within a single pixel-clock cycle. This is due to the streaming interface of the specific imager and can be adapted for other sensor protocols. The logic of the IP block is described in a flowchart at figure 3.8. A state machine is used to control the output of odd and even pixels.

The block contains registers (internal signals) that hold the pixel-value, write-enable status, x-count and end-of-line signals. The process is triggered by the system clock, and the state machine controls the flow. To better understand the timing of the process, a timing diagram of the behavioural simulation is shown in figure 3.7. The numbered circles are explained as follows:

1. On the rising edge of the pixel clock, the incoming pixel data is updated. `pixClk_reg1` is also updated but takes effect on the next rising edge of the system clock.

2. After two system clock cycles, a stable pixel-clock edge is detected, and the state machine progresses to the next state at the next rising edge of the system clock. The input data is loaded into registers that are stable at the next clock cycle. One should notice that the double flopping adds a certain timing constraint as two overhead clock cycles are essentially wasted.

3. The state machine is now in the `WriteOdd` state and both pixel value registers are valid. The system will check the pixel position of the odd pixel and perform some logic to update the `write_en`, `x_count`, `line_count` and `end_of_line` registers accordingly. This decision-making process happens concurrently and is described in figure 3.8.

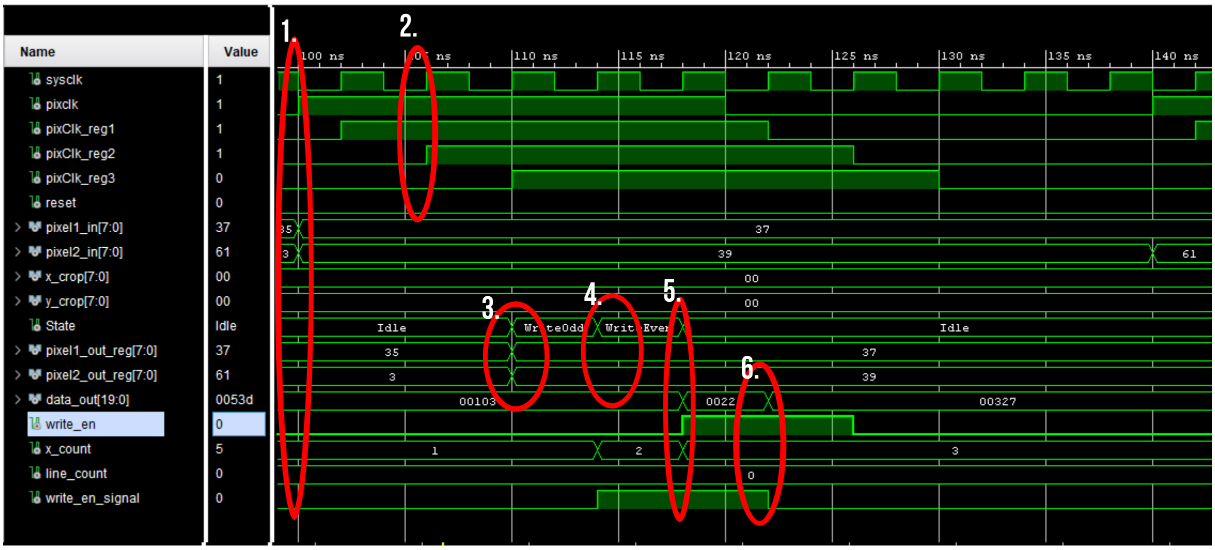


Figure 3.7: Behavioral Simulation of Pixel Split and Count IP Block.

4. The state machine progressed to `WriteEven` after one system clock cycle. Identical logic as in the last state is followed to update the control and count registers. The results and data from the *odd* pixel are now also loaded onto the output ports. The state machine goes into Idle state from here.

5. The odd pixel along with count data is valid and stable on the output ports. `write_en` is activated if the position is within crop area. The *even* pixel is loaded from registers onto the output ports.

6. The even pixel is now valid on the output along with its control and count data. The `write_en_signal` is lowered that takes effect on the output at the next rising edge of the system clock. The state machine remains in `Idle` state until the next rising edge of the Pixel Clock.

The pixels are now counted and transferred to a faster clock domain. The next block in the pipeline reads from this FIFO once data is ready.

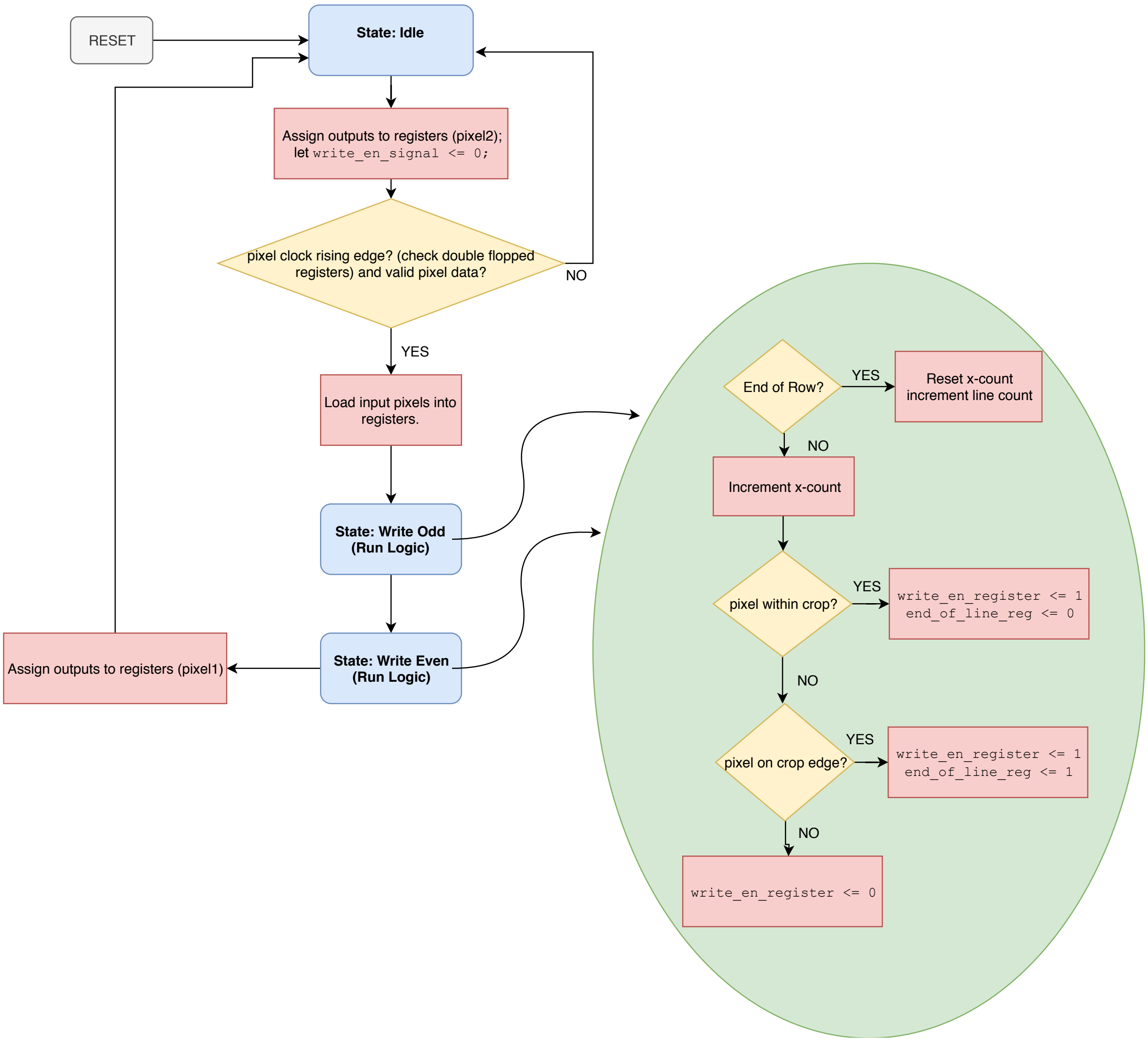

RESET
State: Idle
Assign outputs to registers (pixel2); let write_en_signal <= 0;
pixel clock rising edge? (check double flopped registers) and valid pixel data?
NO
YES
Load input pixels into registers.
State: Write Odd (Run Logic)
State: Write Even (Run Logic)
Assign outputs to registers (pixel1)
End of Row?
YES
Reset x-count increment line count
NO
Increment x-count
pixel within crop?
YES
write_en_register <= 1
end_of_line_reg <= 0
NO
pixel on crop edge?
YES
write_en_register <= 1
end_of_line_reg <= 1
NO
write_en_register <= 0

Figure 3.8: Flowchart describing the logic of the Pixel Count IP block process.

3.5.3 Threshold and Margin Counter

The threshold and margin counter IP block is essentially responsible for comparing the incoming pixel values to the threshold and only passing valid pixels to the next block that is used to create new regions or update existing regions. It reads data from the 20-bit FIFO populated by the previous block. The block operates in the system clock domain and will output data along with control signals (`regionDone`, `valid` and `end_of_line_out`) to help decision making in the next step. Figure 3.9 shows a black-box of the IP core. A state machine controls the flow of the process, and a handshake protocol is implemented for communicating synchronously with the next block in the pipeline. Handshaking is necessary because the next block can take multiple clock cycles to work through its state machine. When two pixels next to each other are passed on, data might be missed due to the next process still being busy. A `data_ready` signal is activated when valid outputs are available. Only when the next block is in an idle state, and this output data is loaded onto its registers will the `read_ack` signal be activated, letting this block know that it can load new data onto the outputs. Since this block does not have direct access to the active region registers and only passes valid pixels, it somehow needs to know when an

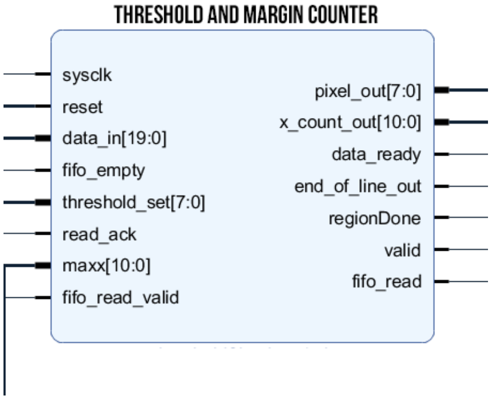


Figure 3.9: IP Block of pixel count and FIFO control.

end of a region (per line) is reached. This can be achieved by knowing the maximum X-count value of the active region. Pixels are streamed starting left to right (increasing x-count). Logic follows that when a pixel-value is below the threshold, and the x-position is greater than the maximum x-value of the region plus a margin, then the region is not grown further for the current line. This block has no knowledge of the state of regions; therefore, a `maxx` value is returned from the next block to enable this functionality. It would have been possible to integrate this entire block with the next to avoid passing register values between blocks as described, but the advantage of modular design and better insight during hardware debug makes the handshaking complications worthwhile.

To understand the flow of the state machine, some of the signals in this block is explained. Signals are essentially local registers to the block that are used in similar to how variables are used in software code. The major difference is that all signals are concurrently updated on a clock-edge because they are realised as Flip-Flops during synthesis and implementation. The important signals include:

- `maxx_buf`. This stores the value of `maxx` input.
- `newRegion`. This indicates whether a new active region is loaded/created or not and is used to determine when to close a region.
- `has_read`. This signal is used to keep track of the handshake between the block and the next block. It goes low when `read_ack` is activated while in Idle.
- `fifo_read_signal`. This signal is a status indicator that keeps track of the handshaking.

On each rising edge of the system clock, the `maxx_buf` is updated. When this buffer changes a value, it indicates that a new active region is loaded. The `newRegion` signal is updated accordingly and is used to know when to close a region. The rest of the functionality all happens in a state machine described below in figure 3.10. The state machine makes timing complications easier to handle as it splits the process into three states, happening in three clock cycle steps:

1. **Idle**. This is the default state, and the block stays in this state until data is available on the FIFO and the next block has already read the current output data. If this

is true, it loads data from the FIFO into its buffers and moves to the next state at the next rising edge of the system clock. Handshaking with the next block is also handled in this state.

2. **Check Pixel**. At this state, the data is already loaded from the FIFO, and the `fifo_read` output is lowered to avoid reading from the FIFO twice. Values in the buffers are checked, and the control outputs are updated accordingly. If no action is taken with the current pixel, the machine returns to the idle state. If an action is taken, such as creating a new region, growing an active region or closing an active region, the machine goes to the next state.

3. **Process Output**. At this state, the buffer data is loaded onto its outputs, and the `data_ready` line is made high to start a handshake procedure with the next block. The machine also returns to the Idle state at the next rising edge.

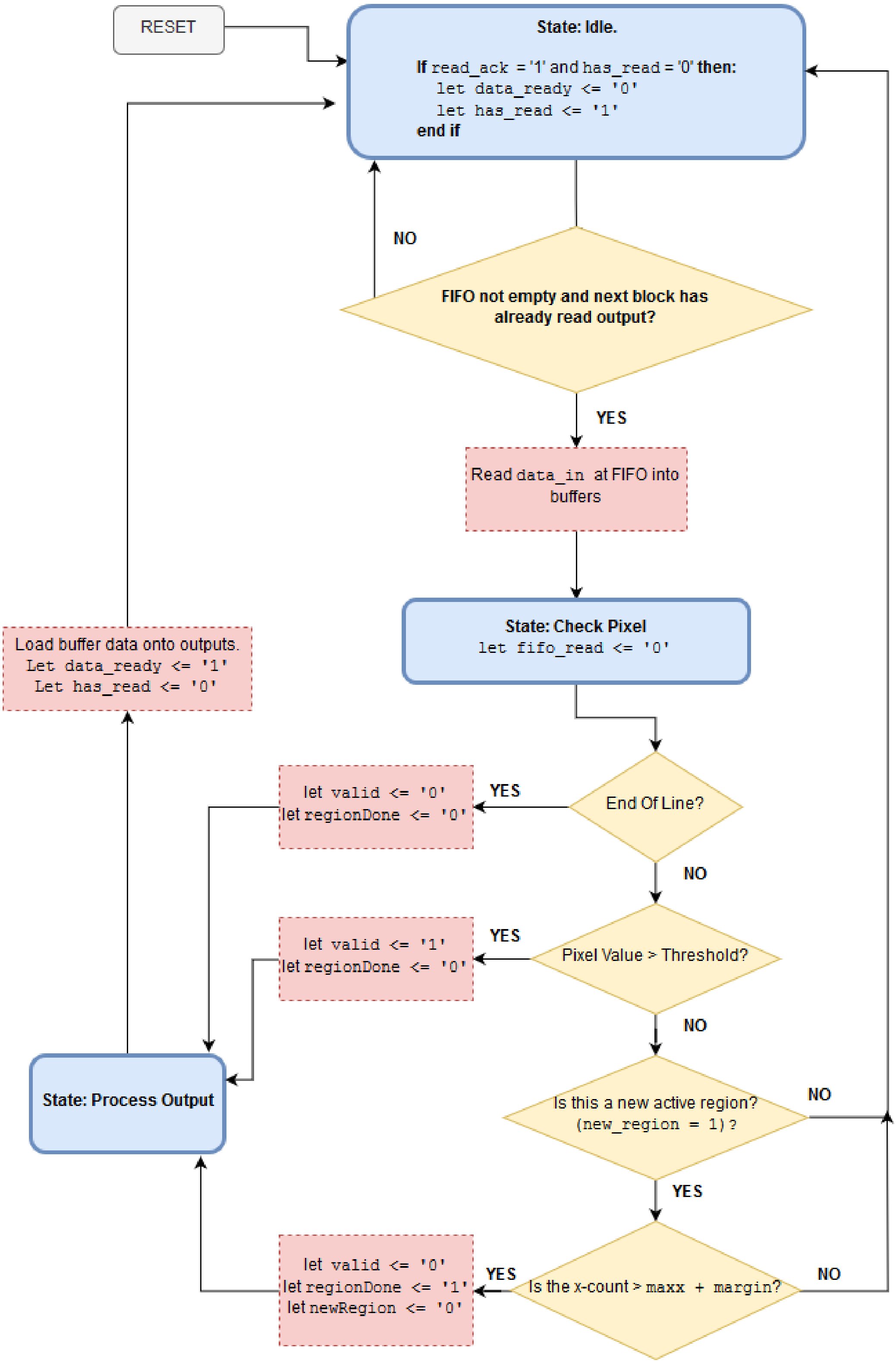


Figure 3.10: Logical state diagram of Threshold block. The blue blocks indicate states; the yellow diamonds indicate decision-making units and the red blocks indicate actions taken at each state or after each decision.

3.5.4 Region Manager with Rotating FIFO

The heart of the centroid detection pipeline lies with this block. A state machine controls the flow of logic, and a rotating FIFO is used to store and update ongoing regions dynamically. Figure 3.11 shows the black box with all of the inputs and outputs.

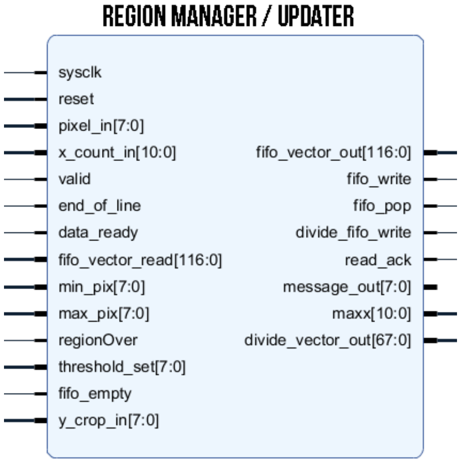


Figure 3.11: IP Block of Region Manager.

The concept of the rotating FIFO adapted from Lindh [21], is to take advantage of the fact that pixels are streamed in from left to right for each row. By de-queuing and queuing growing regions to a single FIFO, the left-most region always appears first in the queue (FIFO) at the start of each line. Lindh's design had some constraints and issues: the X-centre position of each line in a region is calculated at every region line-end using a serial divider. This takes many clock cycles and causes data to be missed if regions form close to each other. The centroid location is calculated using the mean value ($\frac{\sum X_{centre}}{\text{number of rows}}$). The reason for this is that he approaches each line completely independent and only updates the region variables should the X-centre of the new detected line be within a margin of an existing region.

The adapted design stores all of the variables mentioned in section 3.4 of the *active* region in local registers which is used to do logical decision making and centre of mass calculations. This consolidates the process and allows for more control when growing regions compared to Lindh. Once a new active region must be created or loaded, these variables are enqueued to a 117-bit FIFO to be used and updated later. If the region comes to an end (no more pixels are added to it), then relevant variables are passed on to a different 68-bit FIFO that initiates the divide calculations. A state diagram is shown in figure 3.12. Because of the complexity of the decision making within each state, not all of the detail is presented in the figure. There are seven states used in the machine to create, update and close regions. The system remains in Idle until data-ready is high. From here it progresses to various states depending on the incoming commands, the count values and many other variables. A First-Word-Fall-Through FIFO is used in the design that makes it possible to read data from the FIFO without de-queueing it. A row-by-row description of FIFO events follows on page 45.

Table 3.2 shows the data structure of the rotating FIFO. 16 bits are chosen for the cumulative intensity ($\sum_{i=0}^{\text{totalpixels}}[\text{pixel}[i]_{\text{intensity}} - \text{threshold}]$) since $2^{16}/255 = 257$ meaning

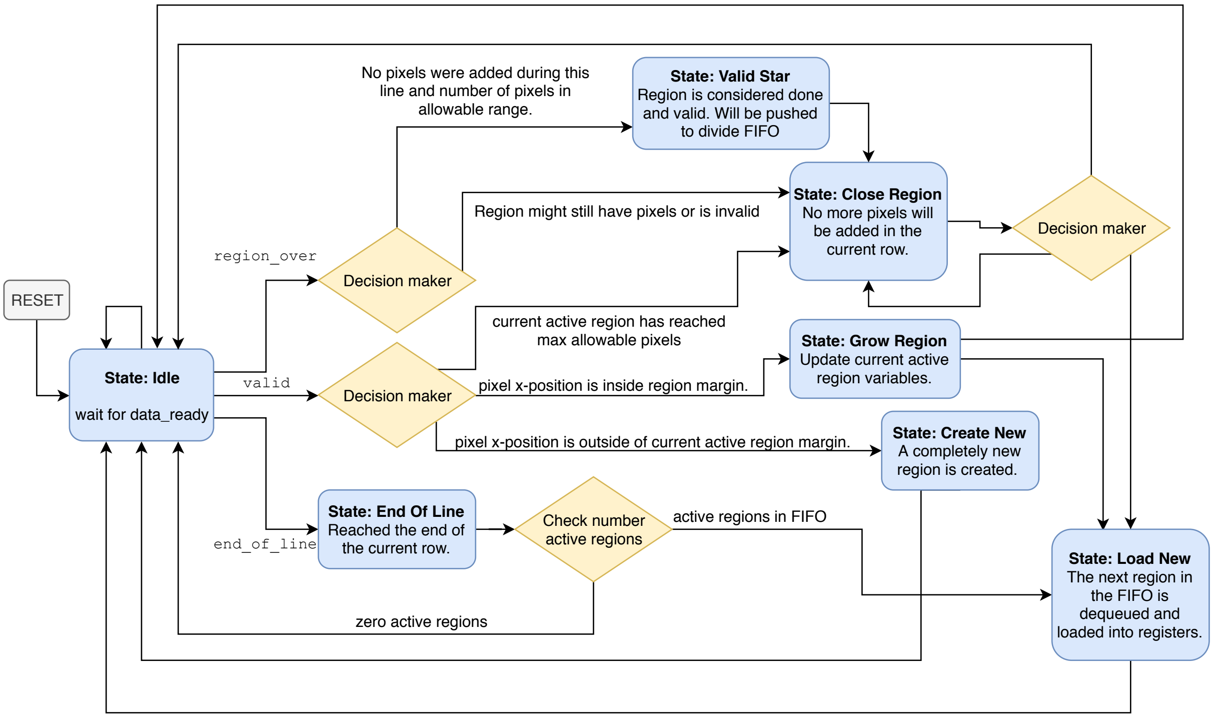

No pixels were added during this line and number of pixels in allowable range.
State: Valid Star
Region is considered done and valid. Will be pushed to divide FIFO
Region might still have pixels or is invalid
State: Close Region
No more pixels will be added in the current row.
Decision maker
region_over
Decision maker
RESET
current active region has reached max allowable pixels
State: Grow Region
Update current active region variables.
State: Idle
wait for data_ready
valid
Decision maker
pixel x-position is inside region margin.
pixel x-position is outside of current active region margin.
State: Create New
A completely new region is created.
State: End Of Line
Reached the end of the current row.
end_of_line
Check number active regions
active regions in FIFO
State: Load New
The next region in the FIFO is dequeued and loaded into registers.
zero active regions

Figure 3.12: State Machine diagram showing the logical flow of the Region Manager IP flow. Row-by-Row description is explained on page 45.

Bits	Description
10 downto 0	Min X
21 downto 11	Max X
32 downto 22	Start Y
40 downto 33	Line Count
48 downto 41	Number of Pixels
74 downto 49	$\sum_{i=0}^{\text{totalpixels}}[(\text{pixel}[i]_x) \times (\text{pixel}[i]_{\text{intensity}} - \text{threshold})]$
100 downto 75	$\sum_{i=0}^{\text{totalpixels}}[(\text{pixel}[i]_y) \times (\text{pixel}[i]_{\text{intensity}} - \text{threshold})]$
116 downto 101	$\sum_{i=0}^{\text{totalpixels}}[\text{pixel}[i]_{\text{intensity}} - \text{threshold}]$

Table 3.2: Data Structure of rotating FIFO of active regions.

that in a worst case where all of the pixels in a region is completely saturated, overflow will only occur after 257 pixels. This is sufficient as such large regions are considered invalid. The threshold value is not used in this calculation since it is variable that can be set to a very low value depending on the nature of the image. The X-Sum and Y-Sum ($\sum_{i=0}^{\text{totalpixels}}[(\text{pixel}[i]_y) \times (\text{pixel}[i]_{\text{intensity}} - \text{threshold})]$) are chosen as 26 bits. In a worst case scenario where large regions with many saturated pixels are found at the rightmost edge of the image, overflow will occur at:

$$\text{Max pixels before overflow} = \frac{2^{26}}{255 \times 2000} \tag{3.6}$$

$$\approx 131 \text{ pixels.} \tag{3.7}$$

Although 131 pixels does not seem much, it is unlikely that all of the pixels in a region is completely saturated and at the edge of the image. In many cases, 131 pixels would be considered too much for a valid region. A check for overflow is built into the state machine that flags a region as invalid when the registers are close to overflow. Once again, the threshold value is not used in this computation; therefore, in practice, it should allow for even more pixels. For safety, it is not recommended to let the maximum pixels per region parameter be more than 200.

A row-by-row description of FIFO events using a small example grid (figure 3.13) follow below.

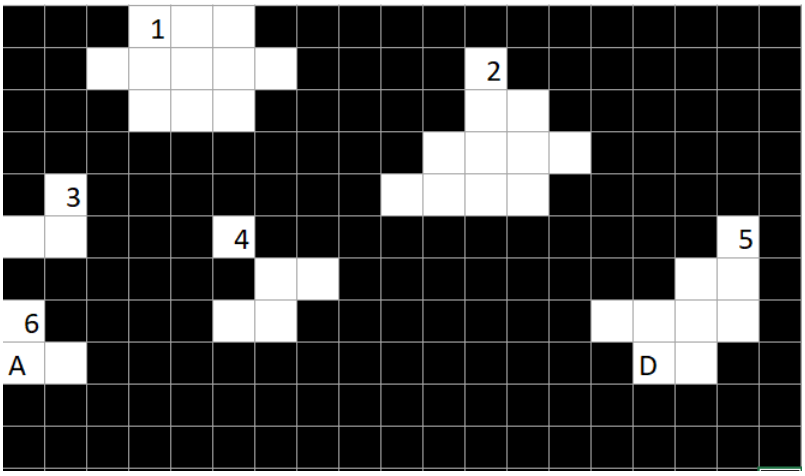
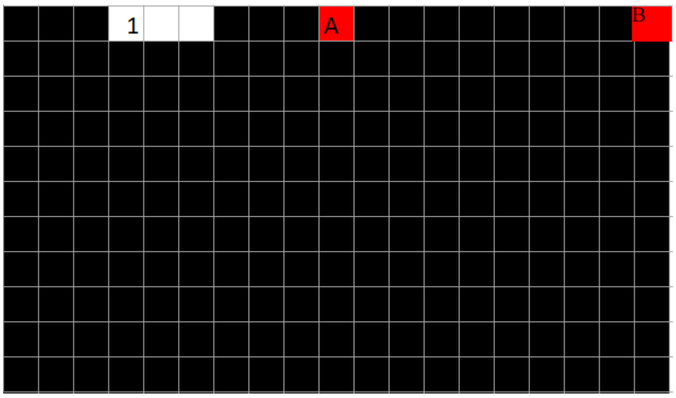


Figure 3.13: Simple example grid to explain row-by-row FIFO events in section 3.5.4

Row-By-Row description.

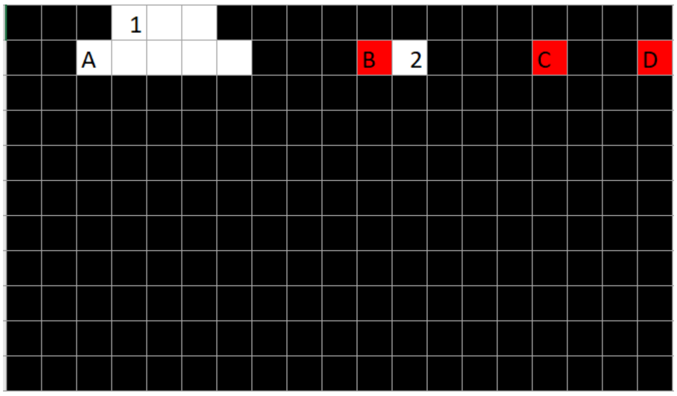
A row-by-row description of example events is presented. Each line has markers that are used to describe relevant events in tables. White blocks indicate pixels above the threshold and red blocks are event indications.



Row 1

(Row 1)

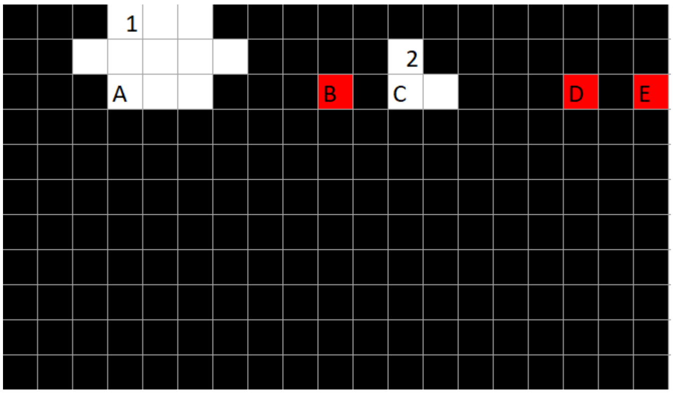
Marker	**Description**	**FIFO**	**Active Region**
1	Region 1 discovered and created. Next two pixels will be added to region since they are within the margin.	→ [] →	None
A	The x-position is outside of the margin, region 1 is enqueued.	→ [] →	1
B	End of line reached, region 1 is loaded from FIFO but not de-queued.	→ [1] →	1



Row 2

(Row 2)

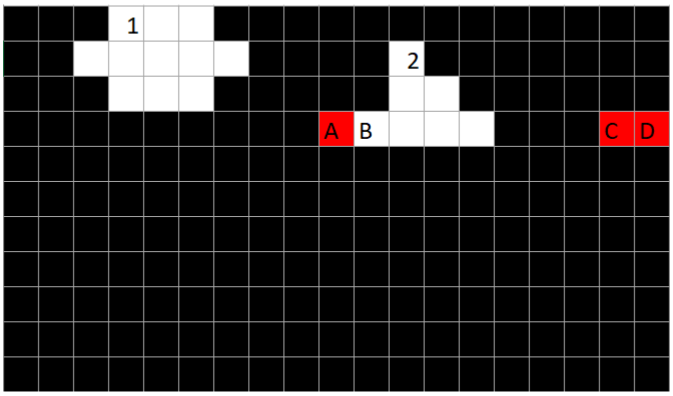
Marker	**Description**	**FIFO**	**Active Region**
A	Pixel position within margin of region 1. Pixels added to region.	→ [1] →	1
B	The x-position is outside of the margin for region 1, the updated values for region 1 is enqueued and old region values are de-queued.	→ [1] →	1
2	Region 2 is discovered and created. Pixel added to region.	→ [1*] →	None
C	X-position is outside of the margin for region 2, region 2 is enqueued. First in FIFO (Region 1) has min-x less than region 2 max-x, therefore not loaded.	→ [1*] →	2
D	End of Line Reached. Region 1 is loaded but not de-queued from the FIFO.	→ [2, 1*] →	None



Row 3

(Row 3)

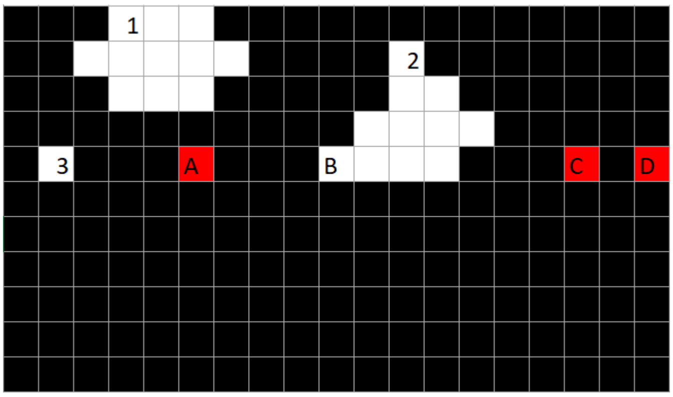
Marker	Description	FIFO	Active Region
A	Pixel position in within margin of region 1. Pixels added to region.	$\rightarrow [2, 1^*] \rightarrow$	1^*
B	The x-position is outside of the margin for region 1, updated region 1 is enqueued and old region 1 is de-queued. Region 2 is loaded from the FIFO on the next clock cycle since its min-x is greater than region 1 max-x.	$\rightarrow [2, 1^*] \rightarrow$	1^*
C	Pixel position in within margin of region 2. Pixels added to region.	$\rightarrow [1^{**}, 2] \rightarrow$	2
D	X-position is outside of the margin for region 2, updated region 2 is enqueued and old region 2 de-queued. First is FIFO (Region 1) min-x is less than region 2 max-x, therefore not loaded.	$\rightarrow [1^{**}, 2] \rightarrow$	2
E	End of Line Reached. Region 1 is loaded from FIFO but not de-queued.	$\rightarrow [2^*, 1^{**}] \rightarrow$	None



Row 4

(Row 4)

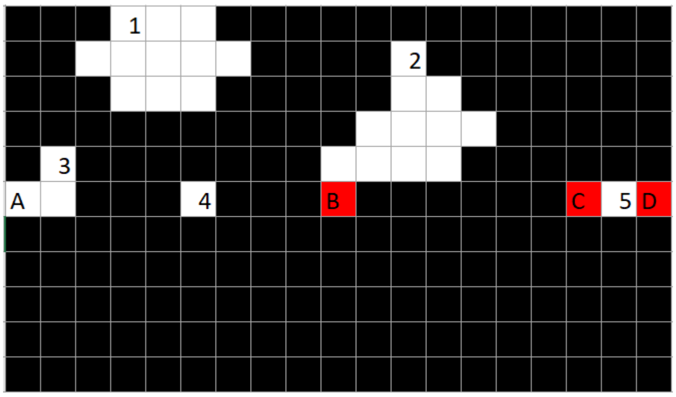
Marker	**Description**	**FIFO**	**Active Region**
A	The x-position is outside of the margin for region 1. Since no pixels were added in this row, the region is closed and de-queued. The values will now be pushed to the divide FIFO. Region 2 is loaded from the FIFO at the next clock cycle but not de-queued.	→ $[2^*, 1^{**}]$ →	1^{**}
B	More pixels added to region 2.	→ $[2^*]$ →	2^*
C	X-position is outside of the margin for region 2, updated region 2 is enqueued and old de-queued.	→ $[2^*]$ →	2^*
D	End of line reached. Region 2 is loaded from FIFO.	→ $[2^{**}]$ →	None



Row 5

(Row 5)

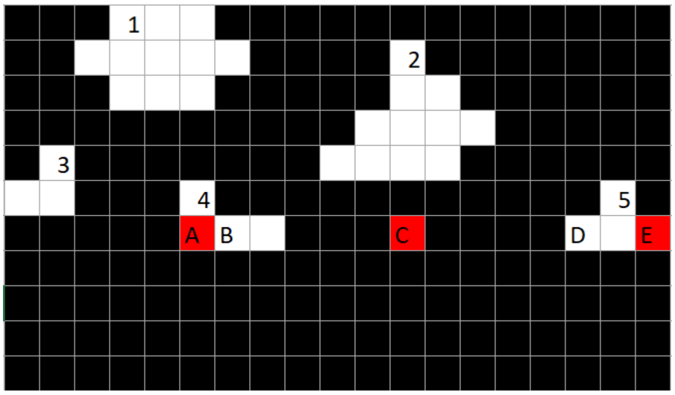
Marker	**Description**	**FIFO**	**Active Region**
3	New valid pixel found with x-position less than min-x of active region (Region 2). Region 3 is discovered and created.	$\rightarrow [2^{**}] \rightarrow$	2^{**}
A	X-position is outside of the margin for region 3, region 3 is enqueued. Region 2 is loaded from FIFO.	$\rightarrow [2^{**}] \rightarrow$	3
B	More pixels are added to region 2.	$\rightarrow [3, 2^{**}] \rightarrow$	2^{**}
C	X-position is outside of the margin for region 2, updated region 2 is enqueued and old dequeued. Region 3 is not loaded from FIFO since its max-x is less than current x position.	$\rightarrow [3, 2^{**}] \rightarrow$	2^{**}
D	End of line reached. Region 3 is loaded from FIFO.	$\rightarrow [2^{***}, 3] \rightarrow$	None



Row 6

(Row 6)

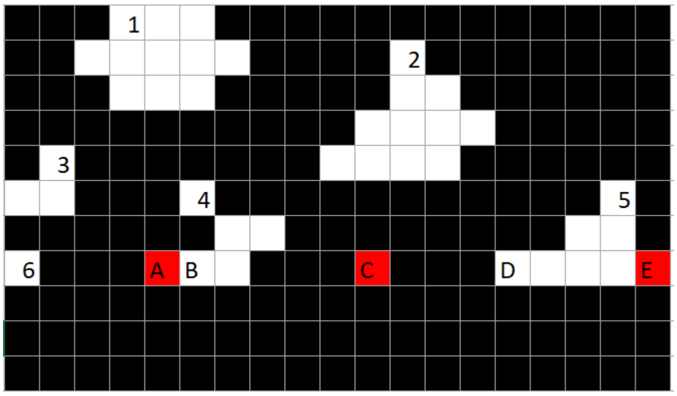
Marker	Description	FIFO	Active Region
A	More pixels added to region 3.	→ $[2^{***}, 3]$ →	3
4	X-position is outside of the margin for region 3, updated region 3 is enqueued and old de-queued. Region 4 is discovered and created.	→ $[2^{***}, 3]$ →	3
B	X-position is outside of the margin for region 4, region 4 is enqueued and Region 2 is loaded from FIFO.	→ $[3^{*}, 2^{***}]$ →	4
C	X-position is outside of the margin for region 2 but no pixels were added in this row. Region is closed and de-queued. Values are sent to divide FIFO for further calculation. Region 3 is NOT loaded since max-x is less than current x.	→ $[4, 3^{*}, 2^{***}]$ →	2^{***}
5	Region 5 is discovered and created.	→ $[4, 3^{*}]$ →	None
D	End of row reached. Region 5 is enqueued and region 3 is loaded from FIFO.	→ $[4, 3^{*}]$ →	5



Row 7

(Row 7)

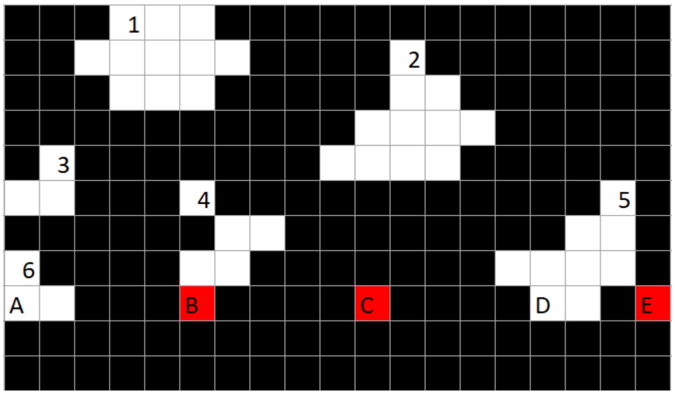
Marker	Description	FIFO	Active Region
A	X-position is outside of the margin for region 3. Region 3 is closed and de-queued. It is not valid and not sent to divide FIFO. Region 4 is loaded from FIFO.	$\rightarrow [5, 4, 3^*] \rightarrow$	3^*
B	More pixels added to region 4.	$\rightarrow [5, 4] \rightarrow$	4
C	X-position is outside of the margin for region 4, updated region 4 is enqueued old region 4 is de-queued. Region 5 is loaded from the FIFO at the next clock cycle.	$\rightarrow [5, 4] \rightarrow$	4
D	More pixels added to region 5.	$\rightarrow [4^*, 5] \rightarrow$	5
E	End of row reached. updated region 5 is enqueued and old region 5 de-queued. Region 4 is loaded from FIFO.	$\rightarrow [4^*, 5] \rightarrow$	5



Row 8

(Row 8)

Marker	**Description**	**FIFO**	**Active Region**
6	Region 6 discovered and created.	$\rightarrow [5^*, 4^*] \rightarrow$	4^*
A	X-position is outside of the margin for region 6. Region 6 is enqueued. Region 4 is loaded from the FIFO.	$\rightarrow [5^*, 4^*] \rightarrow$	6
B	More pixels added to region 4.	$\rightarrow [6, 5^*, 4^*] \rightarrow$	4^*
C	X-position is outside of margin for region 4. Updated region 4 enqueued and old region 4 de-queued. Region 5 is loaded at next clock cycle.	$\rightarrow [6, 5^*, 4^*] \rightarrow$	4^*
D	More pixels added to region 5.	$\rightarrow [4^{**}, 6, 5^*] \rightarrow$	5^*
E	Updated region 5 enqueued and old region 5 de-queued. Region 6 loaded from FIFO at next clock cycle.	$\rightarrow [4^{**}, 6, 5^*] \rightarrow$	5^*



Row 9

(Row 9)

Marker	**Description**	**FIFO**	**Active Region**
A	More pixels added to region 6.	$\rightarrow [5^{**}, 4^{**}, 6] \rightarrow$	6
B	X-position is outside of the margin for region 6. Updated region 6 is enqueued and old de-queued. Region 4 is loaded from the FIFO at the next clock cycle.	$\rightarrow [5^{**}, 4^{**}, 6] \rightarrow$	6
C	X-position is outside of the margin for region 4. No more pixels were found in this row. Region 4 is now closed and de-queued. Region 5 loaded from FIFO at the next clock cycle.	$\rightarrow [6^{*}, 5^{**}, 4^{**}] \rightarrow$	4^{**}
D	More pixels added to region 5.	$\rightarrow [6^{*}, 5^{**}] \rightarrow$	5^{**}
E	Updated region 5 enqueued and old region 5 de-queued. Region 6 loaded from FIFO.	$\rightarrow [6^{*}, 5^{**}] \rightarrow$	5^{**}

Taking a look at row nine, one can see that the margin might lead to undesirable region shapes. Region 4, for instance, has a pixel with no direct neighbours. Many region growing algorithms discard such a pixel to increase accuracy. Another thing to notice is when two regions form very close to each other (such as in this row-by-row example), and the margin is fairly large, some overlapping might occur, and two irregularly shaped regions can form in close proximity. In contrast, when a large region occurs with pixel values lower than the threshold in between and the margin size is small, then two stars can be detected whereas, in reality, only a single star was present. In real star photos, the stars are usually not this close to each other and it should not become a problem with clear star photos. Given the nature of a star PSF, gaps of pixels lower than the threshold should not be a problem. When large blobs of high-intensity pixels are found in an image, such as the moon, then the algorithm will most likely discard the region due to crossing the maximum pixel count. Even if these false stars are detected, they should be flagged invalid during matching.

3.5.5 Divider Block

The final step in the star detection process is to perform the division explained in equations 3.1 and 3.2. A sub-circuit is designed to control the divide logic flow and calculation. The circuit can be seen in figure 3.14. A custom IP block (1) waits until there are data available on the divide-FIFO. When data is ready, it initialises two fixed-point dividers (2 and 3). Each divider receives an unsigned dividend of 26 bits and a divisor of 16 bits (variables explained in table 3.3). The output is a fixed point value with a fractional width of 8 bits. Although 8-bit fraction resolution may seem unnecessary ($\frac{1}{256}$), using fewer bits for the fraction resulted in computational errors and inaccurate results. The output of the dividers are 40 bits, but the top 8 bits are discarded since the decimal part of the result is never larger than the image-width pixel resolution of 2048 (11 bits). 32-bit results are used as the AXI interface works with 32-bit data transactions.

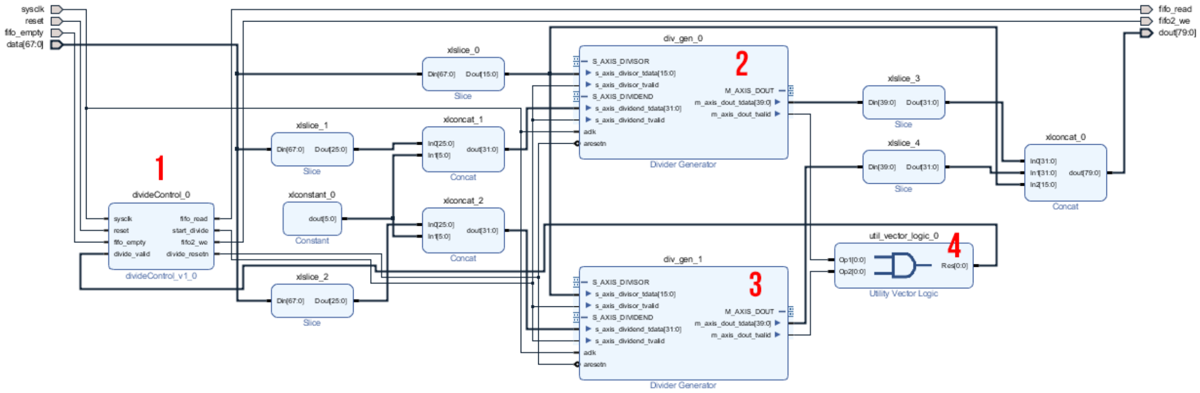


Figure 3.14: Divisor block sub-circuit.

When both dividers are done (4), a signal is activated, and an interrupt is generated to the PS. These fixed-point dividers have a significant latency of 36 cycles. A FIFO buffer before the divider block must be used to avoid timing errors. Table 3.3 shows the data structure of the divide FIFO.

Bits	**Description**	**Unsigned Range**
15 downto 0	$\sum_{i=0}^{\text{totalpixels}}(\text{pixel}[i]_{\text{intensity}} - \text{threshold})$	0 - 65 536
41 downto 16	$\sum_{i=0}^{\text{totalpixels}}[(\text{pixel}[i]_x) \times (\text{pixel}[i]_{\text{intensity}} - \text{threshold})]$	0 - 67 108 864
67 downto 42	$\sum_{i=0}^{\text{totalpixels}}[(\text{pixel}[i]_y) \times (\text{pixel}[i]_{\text{intensity}} - \text{threshold})]$	0 - 67 108 864

Table 3.3: Data Structure of divide FIFO.

3.5.6 AXI Interface and Full System

The entire system is configured and connected in the Vivado Suite using block diagrams. The final design is shown in figure 3.15. Notice that the design consists out of more blocks than mentioned in figure 3.4. Many of these are AXI interconnect blocks that are used to send and receive data between the PL and PS. The ZYNQ block (1.) can also be seen and generates the system clock used to drive the PL. It also acts as the AXI Master. Other IP blocks are Xilinx-generated and include the input buffers, processor-system resets, FIFO blocks and AXI GPIO's needed to run the system. Each numbered block in the figure is explained:

1. ZYNQ Processor System.
2. Input buffers. Receives the LVDS signals from the imager and sends it as single values to the next block.
3. Serial to Parallel conversion.
4. Pixel Split and count block.
5. 20 bit FIFO to cross over to system clock domain.
6. Threshold and Margin count block.
7. Regional Manager.
8. 117 bit rotating FIFO.
9. 86 bit Divide FIFO buffer.
10. Divider Block sub-circuit in a wrapper.
11. AXI interconnect to pass centroid results to PS.
12. AXI GPIO that generates interrupt at PS when new valid results are found or end of the frame is reached.
13. AXI interconnect that passes parameter values (Threshold, Max Pixels per region, Min Pixels per region, X-resolution, Y-resolution, X-Crop, Y-Crop) to the PL and updates on each reset.
14. AXI GPIO used to send a reset signal from PS to PL.

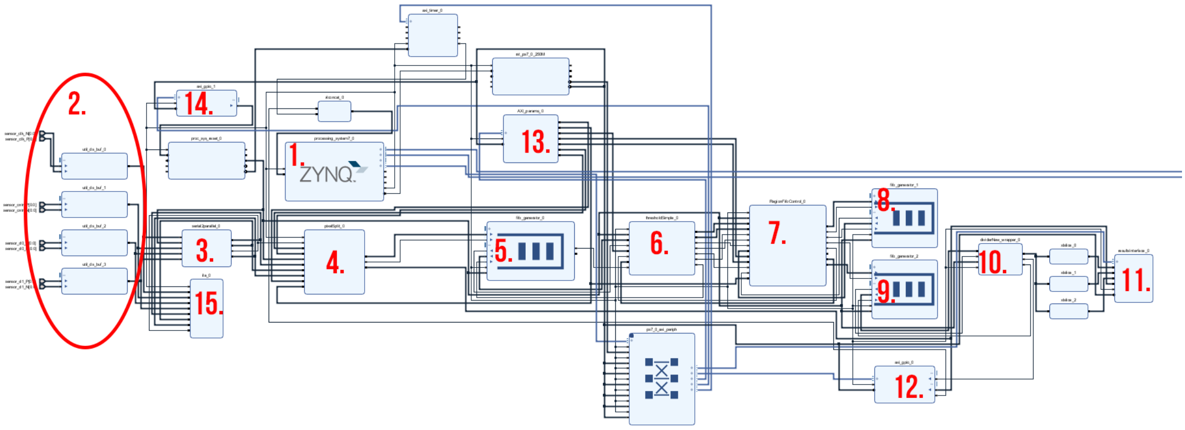


Figure 3.15: The block diagram of the entire system. The numbered blocks are explained in the section.

3.5.7 Timing Overview

The processing pipeline is a synchronous system driven by clocks. Because external input from the imager operates at a different rate than the system, multiple clock domains are used. The three clock domains used on the PL is:

1. Serial Clock. This is the LVDS clock received from the imager. At maximum speed configuration, it transfers two channels of LVDS data at 200MHz.

2. Pixel Clock. This is a divided clock generated by the serial to parallel conversion block. The clock speed is always equal to Serial Clock / 8 and is synchronised to the serial image data.

3. System Clock. This is the fast clock domain at which most of the PL logic operates. The clock is generated by the ZYNQ processor system and can be configured to a maximum of 250MHz.

The pixel split and count block determine timing constraints for the system clock. The rest of the system have buffers or handshaking protocols between blocks that avoids data corruption. The latency of the pixel split block determines the minimum system clock speed.

After the first pixel clock rising edge, three system clock cycles need to pass before a stable edge is detected, and the state machine progress to reading data. One clock cycle is used to process the odd pixel. At the next clock cycle, the even pixel is processed while the odd pixel is transferred. Once cycle later the even pixel is transferred while the machine goes back into Idle. Only at the next clock cycle can a rising edge of the pixel clock be sampled. This adds up to an interval of 7 system clock cycles between a pixel clock. The minimum system clock speed can, therefore, be calculated:

$$\begin{aligned}\text{Minimum System Clock} &= \frac{\text{Serial LVDS Clock}}{8} \times 7 && (3.8)\\ &= \frac{200\text{ MHz}}{8} \times 7 && (3.9)\\ &= 175\text{ MHz}. && (3.10)\end{aligned}$$

The system PL clock is configured to run at 200 MHz. The CPU is configured to run at its maximum speed of 666 MHz. The distortion correction and matching algorithms that take place on the PS side operates at this clock speed of 666 MHz.

3.6 Test Setup

A distinct flow of tests and simulations are created, starting with the least unknown variables to create base tests, up to the point where the entire design can be validated with confidence. The centroid detection and matching are validated separately at first since they can be tested independent. The entire end-to-end system test is the final validation and can be seen in chapter 5. The centroid detection algorithm required the most significant flow of tests due to hardware implementation on FPGA. Figure 3.16 shows the flow of the tests and each case is described below.

1. First, the centroid detection algorithm is written in Python and tested on a PC using the simulated star image. Results can be validated by directly comparing it to known centroid positions of the generated image. The matching algorithm based on the Region Growing Algorithm by Erlank [10] is written in Python and used as a base test. A scripting language gives the added benefit of rapid development and quick visual plots. These Python algorithms now become the baseline test for later validation. This all happens within a PC environment.

2. The centroid detection algorithm is re-designed for hardware and written in VHDL. The simulated star-map image is streamed directly when the behavioural simulation is done within the Vivado suite on a PC. The results are compared to known values, and the design can be debugged and validated all within the PC environment.

3. A real star image is introduced to the algorithm. This demonstrates how the system handles noise and false stars. Since no information about the centroid locations is available, the only way to validate the VHDL code is to compare the centroid results to the results of the Python algorithm (baseline test).

4. Behavioural simulation is now over, and the FPGA implementation needs to be tested. Image data cannot directly be streamed from the PC, so it was decided to build an *emulated* sensor on a separate microprocessor that transfers data with the same protocol and connection as the Gecko imager. This enables a controlled test using simulated images directly on the final hardware design of the FPGA. The speed of the microprocessor is very slow compared to the Gecko, but the functionality of logic can still be accurately tested. This is the longest debugging process since *any* change in the VHDL code requires re-synthesis, re-implementation and re-writing of the bitstream to the FPGA - a process that can take up to 20 minutes each time.

5. The same test is now performed but using a real captured image. Centroid results from the FPGA output are compared to the output of the Python script using the same image.

6. Finally, the Gecko imager can be connected to the FPGA and tested in full speed data transfer. The Gecko imager is placed in a 'star-tube', and a still is captured to the PC. The still is used to run through the Python script. The sensor is connected

to the FPGA and turned on. If the centroid results are the same, then the full centroid detection algorithm proved to be accurate.

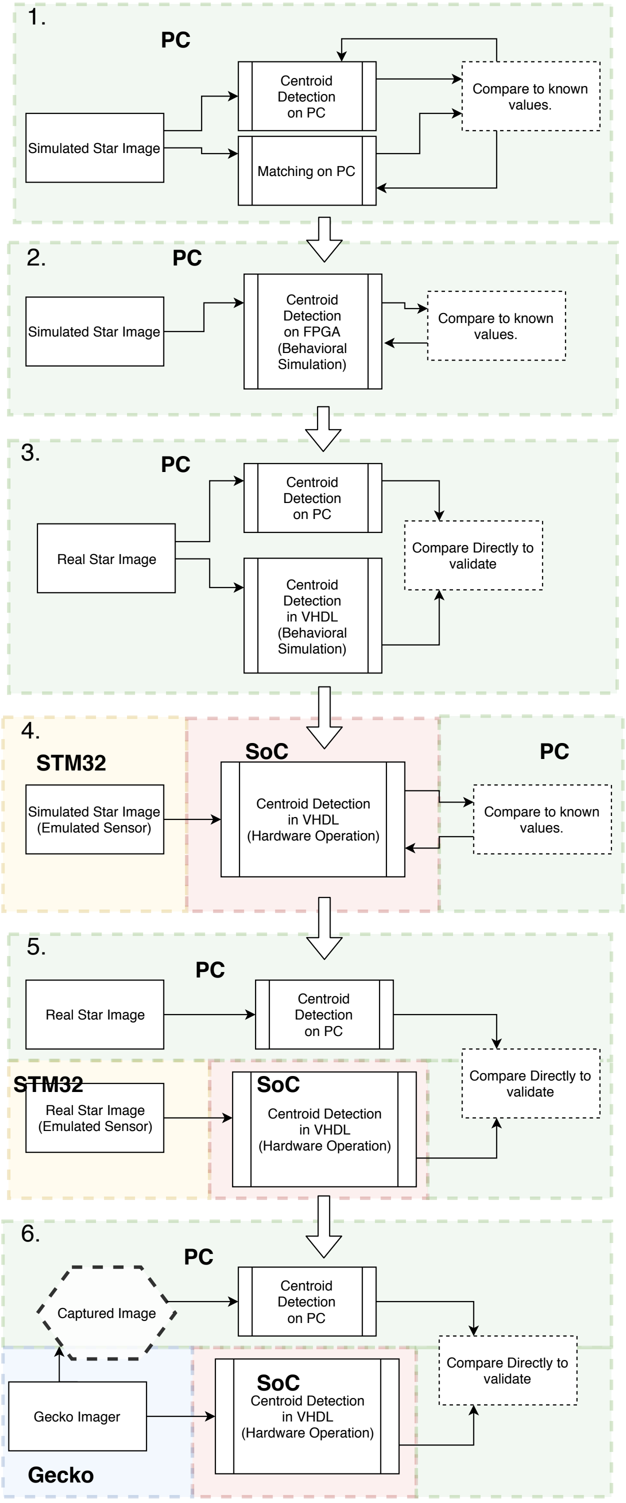

1.
PC
Simulated Star Image
Centroid Detection on PC
Matching on PC
Compare to known values.
2.
PC
Simulated Star Image
Centroid Detection on FPGA (Behavioral Simulation)
Compare to known values.
3.
PC
Real Star Image
Centroid Detection on PC
Centroid Detection in VHDL (Behavioral Simulation)
Compare Directly to validate
4.
STM32
Simulated Star Image (Emulated Sensor)
SoC
Centroid Detection in VHDL (Hardware Operation)
PC
Compare to known values.
5.
PC
Real Star Image
Centroid Detection on PC
Compare Directly to validate
STM32
Real Star Image (Emulated Sensor)
SoC
Centroid Detection in VHDL (Hardware Operation)
6.
PC
Captured Image
Centroid Detection on PC
Compare Directly to validate
Gecko Imager
SoC
Centroid Detection in VHDL (Hardware Operation)
Gecko

Figure 3.16: Centroid Detection validation flow.

3.7 Star-Map Image Simulation

A star-map image is key to testing the validity and reliability of centroid detection and matching algorithms of a star tracker. Real-time tests under the starry sky is not easy to realise due to many unknown variables early in the development stage. Ideally, an image with known star centroid positions, ID's and attitude coordinates, as if taken from the star camera must be used to validate the functionality and accuracy of the design. This can be achieved by star-map image simulation. The working process of star-map simulation is, to a certain extent, the exact inverse of attitude determination. First, the orientation of the star tracker's boresight is chosen arbitrarily. The scope of the simulation, according to the chosen RA and DEC and the FOV , is confirmed. Stars that fall into the window is extracted from a catalogue database and projected onto a 2D image plane. Stars are generated at the image plane coordinates to simulate a de-focussed lens. Noise is finally added to the image, and a simulated star photo is generated. This process is described in figure 3.17.

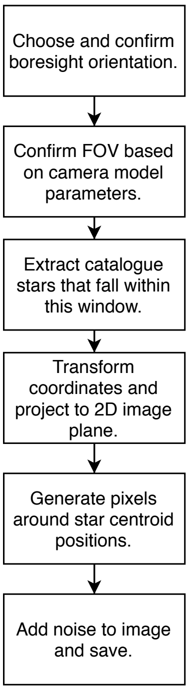

Choose and confirm boresight orientation.
Confirm FOV based on camera model parameters.
Extract catalogue stars that fall within this window.
Transform coordinates and project to 2D image plane.
Generate pixels around star centroid positions.
Add noise to image and save.

Figure 3.17: Star-map image simulation process. This is a similar process followed in other literature [10, 13, 17].

3.7.1 Coordinate Transformation

There are two transformations necessary in the process: First, a rotational transform from the catalogue star ECI coordinates to the star trackers sensor body coordinate system. This rotational transformation method is adapted from a paper by Qian et al. [13]. Second, a projection transformation is done to project the star camera coordinates onto the 2D image plane [7]. The ECI coordinates of a star can be expressed as (α, δ) for RA and DEC. The star tracker directional vector (sensor body vector) can be expressed as

(X, Y, Z) and the image plane coordinates can be expressed in the continuous domain as (x, y). Figure 3.18 shows how these coordinate systems are related.

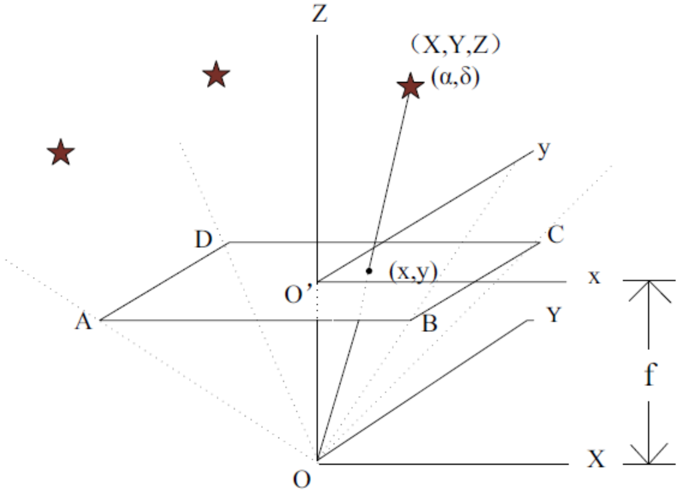


Figure 3.18: Coordinate model of star tracker imager. (X, Y, Z) axis is the star camera sensor body coordinates and (x, y) are the image plane coordinates. Image taken from Qian et. al. [13]

Considering the distance between the earth and other fixed stars, it is assumed that the camera boresight is at the centre of the earth thus any translational transformation between ECI coordinates and star camera coordinates can be ignored. A rotation-matrix, **M**, is expressed as follows:

$$[X, Y, Z]^T = \mathbf{M}[U, V, W]^T, \tag{3.11}$$

where $[U, V, W]^T$ is the ECI celestial coordinates in Cartesian form. Using the principle of 3-1-3 Euler coordinate rotation [9], **M** can be expressed as follows:

$$M = \begin{bmatrix} \cos\psi & \sin\psi & 0 \\ -\sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & \sin\theta \\ 0 & -\sin\theta & \cos\theta \end{bmatrix} \cdot \begin{bmatrix} \cos\varphi & \sin\varphi & 0 \\ -\sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 1 \end{bmatrix}, \tag{3.12}$$

where ψ, θ, φ are the 3D rotation angles. By inspecting figure 3.19, where (α_0, δ_0) is the boresight orientation, the rotation angles θ and φ can be related to the boresight as

$$\theta = 90^\circ - \delta_0 \tag{3.13}$$

$$\varphi = 90^\circ + \alpha_0. \tag{3.14}$$

The ECI direction vector of a fixed star, $[U, V, W]^T$ can be also be expressed in RA and DEC as:

$$\begin{bmatrix} U \\ V \\ W \end{bmatrix} = \begin{bmatrix} \cos\alpha\cos\delta \\ \sin\alpha\cos\delta \\ \sin\delta \end{bmatrix}. \tag{3.15}$$

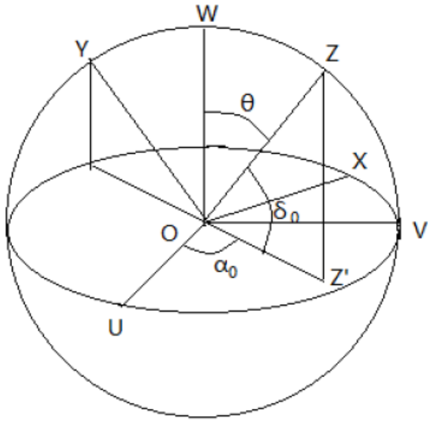


Figure 3.19: Celestial coordinate system to star trackers coordinates. (α_0, δ_0) is the boresight orientation and (U, V, W) is the ECI axis. Image taken from Qian et. al. [13]

By combining equations 3.12, 3.13 and 3.15, a single expression for $[X, Y, Z]^T$ is found:

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} \cos\alpha\cos\delta \\ \sin\alpha\cos\delta \\ \sin\delta \end{bmatrix} \cdot \begin{bmatrix} \cos\psi & \sin\psi & 0 \\ -\sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & \sin\delta_0 & \cos\delta_0 \\ 0 & -\cos\delta_0 & \sin\delta_0 \end{bmatrix} \cdot \begin{bmatrix} -\sin\alpha_0 & \cos\alpha_0 & 0 \\ -\cos\alpha_0 & -\sin\alpha_0 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \tag{3.16}$$

where (α, δ) is the coordinate of a star in the ECI system, (α_0, δ_0) is the chosen boresight orientation and ψ is the rotation angle *around* the boresight (around the Z-axis of star camera coordinates).

The next step is to project these $[X, Y, Z]^T$ vectors onto the image plane. This is done with camera perspective projection. Details on this can be found in lecture notes by Collins [7]:

$$\begin{bmatrix} x \\ y \end{bmatrix} = \frac{f}{Z} \begin{bmatrix} X \\ Y \end{bmatrix}, \tag{3.17}$$

where f is the focal length in millimetre.
Finally, the pixel coordinates can be found by scaling the image plane coordinates, flipping the axis and translating to the top right corner as described in figure 3.20. This can mathematically be described:

$$x\prime = -\frac{x}{S_x} + o_x, \tag{3.18}$$

$$y\prime = -\frac{y}{S_y} + o_y, \tag{3.19}$$

where S_x, S_y is the pixel resolution divided by the sensor size and o_x, o_y is the principle point, or simply the centre of the sensor in pixel units.

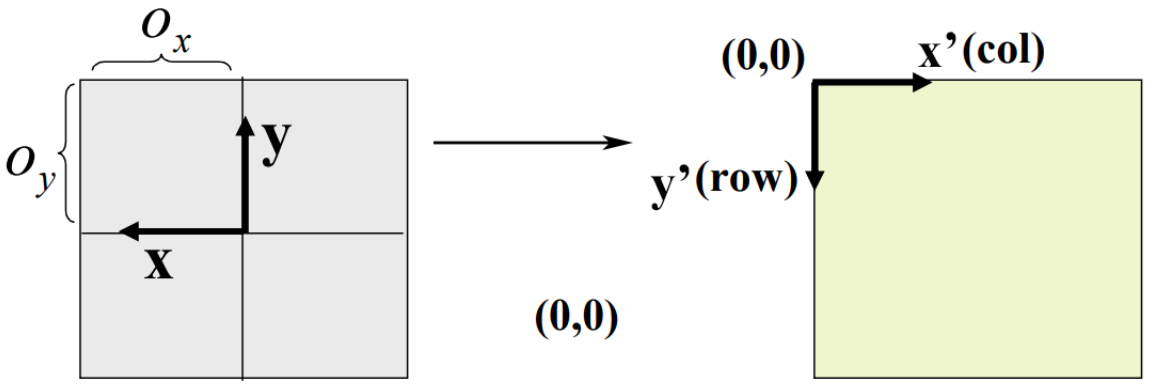


Figure 3.20: Image plane to pixels. Image taken from lecture notes [7].

3.7.2 Star Generation

To generate the actual pixel values of the simulated star, a Gaussian point spread function (PSF) is used. This will approximate the lens blur found in real star images. For each star location, a baseline grid size is chosen. 5x5 pixels is a good estimate. The Gaussian function of the simulated star PSF, adapted from a Thesis by Knutson [17] is used and described:

$$G(X) = e^{-J} \tag{3.20}$$

$$J = \frac{1}{2}(X - \overline{X})^T \Sigma^{-1} (X - \overline{X}), \tag{3.21}$$

where $G(X)$ is the Gaussian function solved at every discrete point of the star grid, $X = \begin{bmatrix} x_i \\ y_i \end{bmatrix}$;

$\overline{X}$ is the sub-pixel centroid coordinates of the star: $\overline{X} = \begin{bmatrix} x_c \\ y_c \end{bmatrix}$;

Σ is defined as: $\Sigma = \begin{bmatrix} \sigma & 0 \\ 0 & \sigma \end{bmatrix}$, where σ is the standard deviation that can be calculated as $\sigma = \frac{\text{FWHM}}{2\sqrt{2\log(2)}}$, and FWHM is the Full Width Half Maximum of the PSF.

3.7.3 Adding Noise

Noise is added to emulate noise often found with digital sensors. A simple approximation is to chose random samples from a Gaussian distribution and scale it down to a maximum noise floor. These values are added to the image after it has been generated.

3.8 Emulated Hardware Sensor

To test the hardware operation of the FPGA design, a simulated image needs to be transferred from an external device. An *emulated* sensor is created using an STM32 microcontroller that streams image data from a Universal Serial Bus (USB) flash drive over Transistor-Transistor Logic (TTL) lines. It was created to emulate the transmission protocol of the Gecko Imager. The main difference is the speed - the maximum speed of the serial data transmission is in the Kilohertz range, almost a thousand times slower than the Gecko. The STM32 board only has TTL output signals, and the MicroZed input pins are all LVDS. A separate LVDS driver (SN75LVDS387 from Texas Instruments) is needed for this configuration to work.

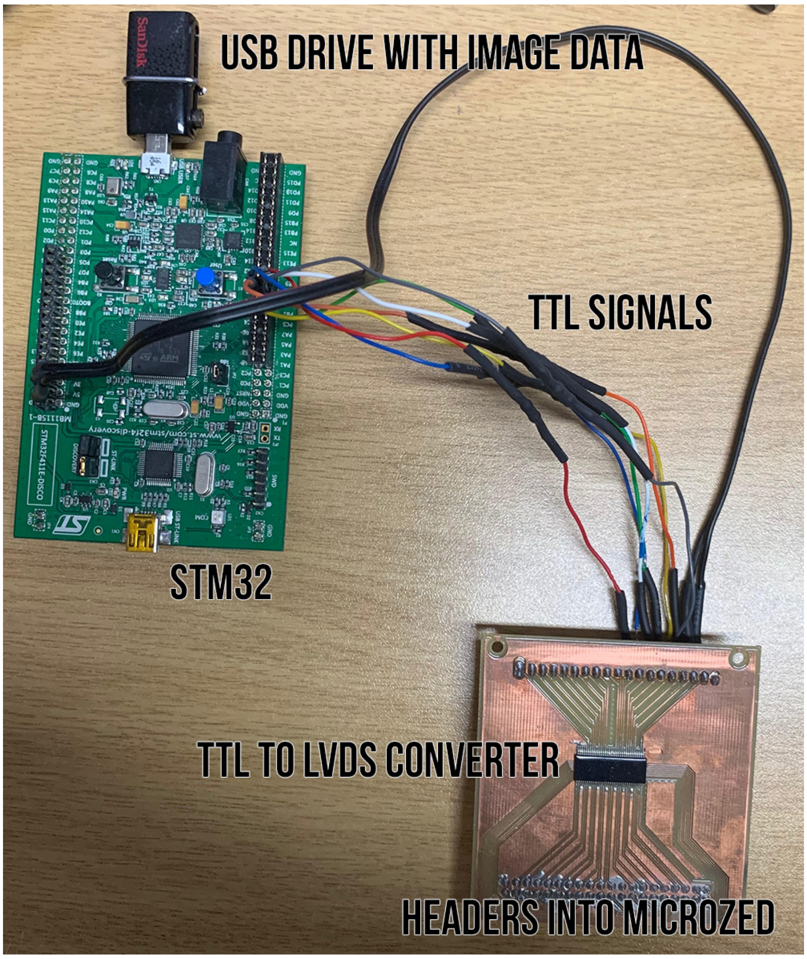


Figure 3.21: Image plane to pixels.

The embedded design follows the following approach:

1. On startup, let the output on the TTL lines be the same as the Gecko Idle state protocol. The clock is set up using a timer interrupt configured to the maximum speed of the microprocessor.

2. When a button is pressed to start streaming - load the first row of the image data into a buffer.

3. Load the bits of the odd and even pixel onto the GPIO pins, update the serial clock level and the control signal according to the protocol of the Gecko imager data transfer.

4. When the end of the first half of row is reached, load the first half of the *next* row into the buffer. Continue.

5. When the end of the second half of a row is reached, load the second half of the next row into the buffer at the correct offset and continue.

This setup provides a controlled test, using simulated and real star images. The data rate is very slow, but the functioning logic of the FPGA design can be evaluated in hardware.

3.9 Centroid Detection Results

The first test results are shown using a simulated image with known centroid locations. The generated image used can be seen in figure 3.22. A table comparing the ground truth coordinates to the results from the Python script and FPGA is shown in table 3.4.

3.9.1 Generated Image

Figure 3.22: Generated image for first tests.

Although following the same algorithm approach, there is a small difference between the results from the FPGA compared to the Python script. This has to do with small nuances in the algorithm and the 32-bit floating-point division compared to fixed-point division on the FPGA. The results are still very similar and close to the ground truth.

Generated X	Generated Y	FPGA X	FPGA Y	Python X	Python Y
1081.6861	876.699	1081.6328	876.7578	1081.6486	876.6951
1252.1765	795.861	1252.168	795.8594	1252.1709	795.8619
496.3066	748.7191	496.3594	748.7695	496.2978	748.7327
1424.668	293.7148	1424.66	293.6914	1424.6615	293.6939
1393.0927	78.1034	1393.0547	78.0625	1393.0556	78.0652

Table 3.4: VHDL (Hardware) and Python centroiding results of simulated star-field image.

The average errors for FPGA results compared to the ground truth generated locations are:

- FPGA Error X: 0.0272 (pixels)
- FPGA Error Y: 0.0597 (pixels)

This is very accurate and shows that the algorithm does work in principle. It is important to note that this is an ideal generated image with very low noise. Real star images should have different results.

3.9.2 Real Star Image

The next important test is to use a real star image that was captured with a real camera. The purpose of the test is to see how the centroid detection works on irregularly shaped stars and when noise gets introduced. As the real centroid locations are unknown, the Python algorithm acts as the baseline test used to compare results. The image is streamed onto the FPGA using the emulated sensor. Figure 3.23 shows the star image along with detected centroid locations from the FPGA. Full comparison of results can be seen in table 3.5.

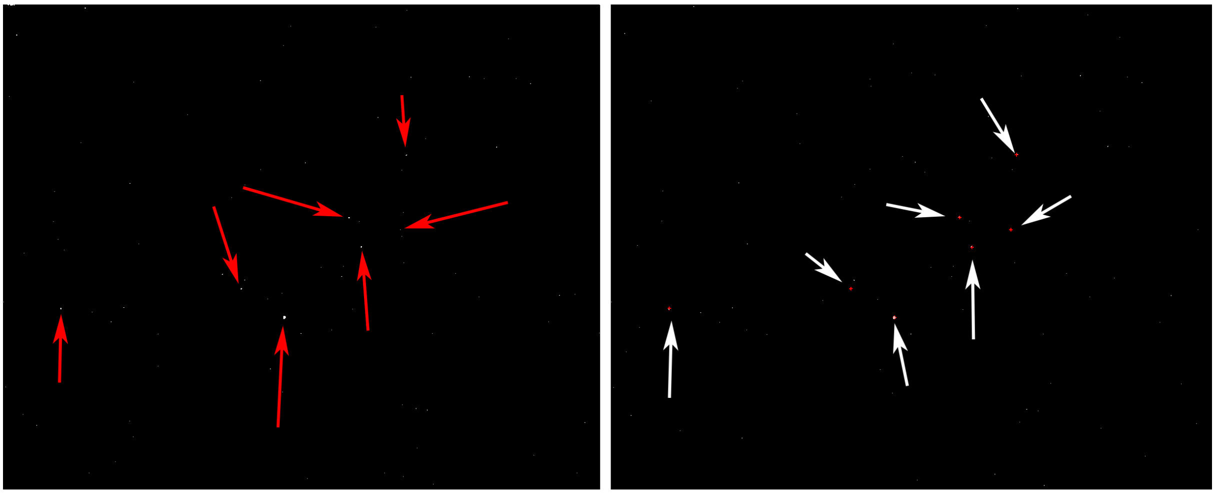


Figure 3.23: Real Star image (on the left) with the detected centroids on FGPA (on the right).

This image was not taken with the Gecko imager in this star tracker but still acts as a suitable evaluation method for the FPGA algorithm.

VHDL X	Python X	Error X (pixels)	VHDL Y	Python Y	Error Y (pixels)
864.734375	864.73718	0.002805	321.453125	321.45513	0.002005
742.175781	742.1789	0.003119	455.679688	455.68349	0.003802
851.476562	851.47899	0.002428	481.578125	481.57983	0.001705
768.074219	768.07645	0.002231	518.292969	518.29664	0.003671
510.832031	510.83556	0.003529	607.375	607.37778	0.00278
124.046875	124.05075	0.003875	649.871094	649.87218	0.001086
603.246094	603.22887	0.017224	668.949219	668.95413	0.004911

Table 3.5: Real Star Image Centroid Comparison between Python script and VHDL results.

The average error (the difference between Python and FPGA results) on this image is:

- Error X: 0.00503 (pixels)
- Error Y: 0.00285 (pixels)

This shows that the FPGA centroid extraction algorithm behaves very similar to a validated detection algorithm, and the problems pointed out in the row-by-row example does not typically appear in clear star photos. The true effect of centroid detection accuracy is seen after the matching and attitude determination algorithms.

3.9.3 Effect of Noise

An evaluation of the effect of noise on centroid detection results is made using simulated star photos. The noise floor level is increased while the intensity of the star PSF is kept constant at 100. This resulted in an expected value of 74 for the signal (star), given the specific PSF. By inspecting the plot at figure 3.24 it is clear that at an expected noise level of around 50 or more, the centroiding error becomes larger than 1 and too inaccurate to use. These errors are the difference between ground truth generated star positions and detected results in Python.

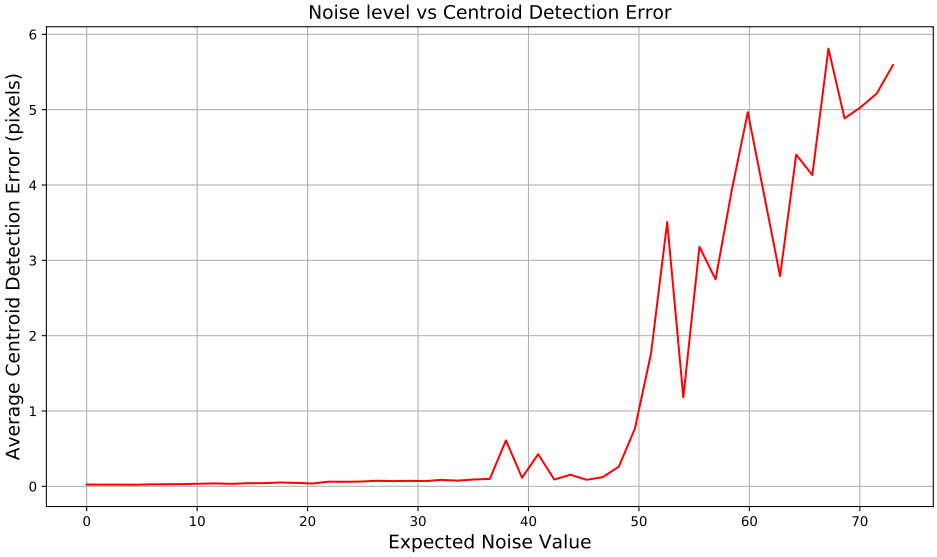


Figure 3.24: Plot of increasing noise intensity vs centroid detection error

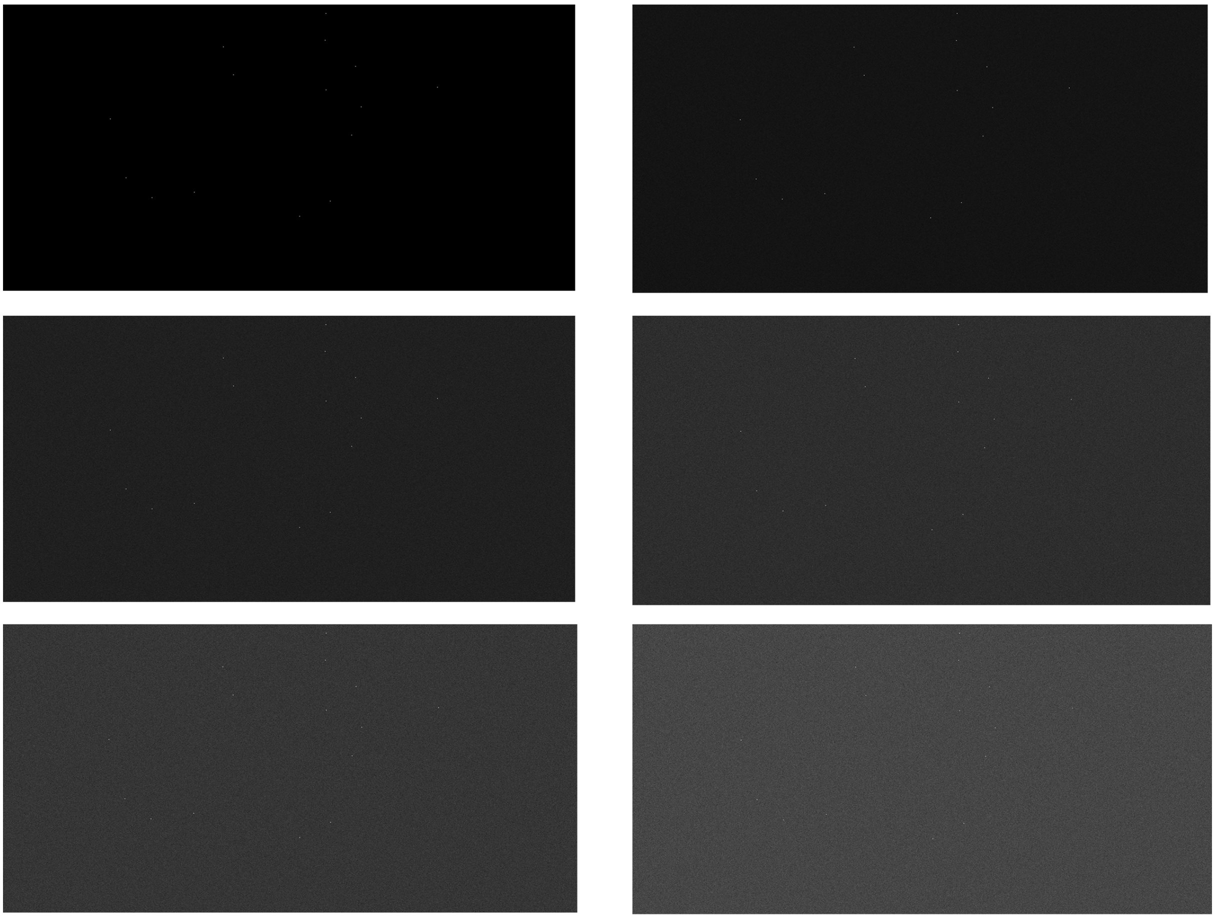


Figure 3.25: Increasing simulated noise on a star image. Top left has no noise, while bottom right has an expected noise intensity of 70.

Chapter 4

Matching Algorithm

Star matching is the most complex algorithm in the star-tracker pipeline. The end-goal is to match each detected star on the image to a catalogue star using the available information and patterns found in the distribution of centroids. Should no match (or low confidence in a match) be found, the centroid is identified as a false star. The input is a list of Cartesian unit vectors in sensor body coordinates found from the previous step, and the output is a list of corresponding star-identities that have known ECI coordinates.

4.1 Star Matching Background

Many suitable matching algorithms have been developed since the first primitive CCD based star trackers appeared [5]. The most considerable differences in these algorithms are how (and what) features are extracted from the star information. Depending on the algorithm and strategy, the star-catalogue is also pre-processed and structured differently. There are many variations and optimisation techniques for matching algorithms found in the literature. Still, most matching algorithms can be classified as implementations of either subgraph matching or pattern matching [10].

Subgraph matching algorithms such as the Triangle Algorithm [27] use features such as the angular distances between known stars to build up an undirected graph, G. The detected stars on the sensor are used to form a second undirected graph, G_s. The goal then is to find a subgraph of G that is isomorphic to G_s [18].

Pattern Matching algorithms such as the Grid Algorithm does not use features such as inter-star angles or distances. Instead, the algorithm tries to associate a unique fingerprint with each star by placing a loose grid over the star and checking for nearby stars in each of the grids cells. Erlank [10] covers useful comparisons between matching algorithms in his thesis.

4.2 Geometric Voting Algorithm

It is desired to have a matching algorithm that is fast and robust in LIS mode and is relatively simple to implement. It is also useful if the algorithm can be performed in parallel with detection, meaning that most of the processing can be done before the end of the frame is reached. Kolomenkin [18] published the Geometric Voting Algorithm in 2008 as a modification to the Search-Less algorithm. His paper provides proper pseudocode that helped to minimize implementation time. Further optimisation is performed on his algorithm by pre-processing and structuring the star catalogue in such a way that almost zero search time is necessary.

The Geometric Voting algorithm can also be classified as part of the subgraph family, but according to Kolomenkin, it is faster and more robust [18]. It consists of a voting scheme built on pairs of stars. In short: A pre-processed lookup-table is created that consists of pairs of catalogue stars and their distances. During runtime, the angular distances between detected centroids are calculated. Catalogue pairs with similar distances vote for all detected image pairs. The catalogue ID that received the most votes for each detected star is usually the correct match.

4.2.1 Algorithm Details

Pre-processing of the catalogue is crucial for the performance of the Geometric Voting Algorithm. There are over 3000 stars with magnitude 5.5 or less in the Smithsonian Astrophysical Observatory (SAO) Star Catalog. Creating a lookup-table with all of these pairs result in more than 9 million entries, requiring large memory units and can take significant time searching. Kolomenkin's solution is only to include star-pairs that have an angular distance less than the FOV of the camera. This dramatically decreases the size of the catalogue. The pre-processing step is described in the pseudocode below:

```
let $N$ = number of stars in catalogue
let $D$ = angular FOV distance
for $i = 0$ to $i = N$ do
    for $j = i + 1$ to $j = N - 1$ do
        compute angular distance, $D_{ij}$
        if $D_{ij} < D$ then
            append $i, j, D_{ij}$ to the lookup table
        end
    end
end
Sort table according to distance.
```

Algorithm 1: Lookup table pre-processing

During runtime, the angular distances between all detected star-pairs are calculated. Each catalogue star pair with an angular distance close enough to the detected star pair receives a vote. This step requires searching through the pre-processed catalogue. The next section proposes an optimised structure.

4.3 Catalogue Structure Optimisation

The motivation is to reduce catalogue search time to a minimum. By pre-processing the distance list so that the index of the array *is* the angular distance, then the star ID's can be accessed at a known memory location instead of performing a search. This technique requires scaling and quantization of the angular distances to fit the distances into a fixed-size array. The angular distance between two vectors is calculated by taking the cosine of the dot product,

$$\text{Distance} = \cos(\vec{a} \cdot \vec{b}). \tag{4.1}$$

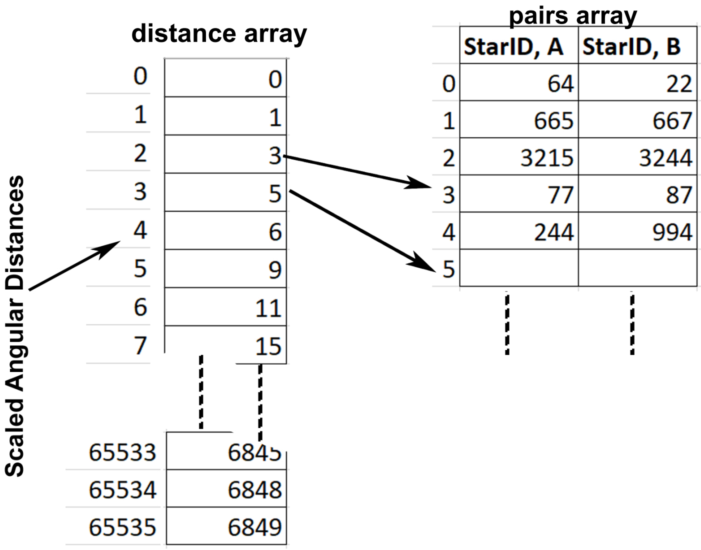
Since this result will be compared to a detected star angular distance, the cosine operation can be skipped to reduce processing time. A range used to scale and quantize all distances must be calculated. The maximum value occurs when two centroids are very close to each other. This value converges to 1 as the vectors are getting closer to each other. The minimum value of the dot product calculation occurs when two stars are far away from each other. Based on the technique described by Kolomenkin, the maximum distance is the diagonal FOV of the star camera. The total range of the dot product will, therefore, be 1 minus the diagonal FOV. The multiplication factor to scale all values into 16-bit resolution is calculated:

$$\text{Multiplication Factor} = \frac{2^{16}}{1 - \text{Diagonal FOV}} \tag{4.2}$$

Instead of saving the actual distance, D_{ij} between stars $\vec{S}_i$ and $\vec{S}_j$, a new distance value $\acute{D}_{ij}$ is calculated:

$$\acute{D}_{ij} = (|(\vec{S}_i \cdot \vec{S}_j)| - \text{Diagonal FOV}) \times \text{Multiplication Factor} \tag{4.3}$$

For pairs that are as far as the diagonal FOV, the distance is 0, and for pairs very close to each other, the distance gets close to 2^{16}. These values are rounded to the nearest integer to be used as an index. The problem with this quantization technique is that pairs of stars with similar distances D_{ij} might have the same distance, $\acute{D}_{ij}$, after scaling. A solution is to create a second array that holds the star pairs ID's while the distance array only contains the index of the first star-pair with that distance. If more than one pair is found with the same scaled distance, then they are listed underneath each other. Figure 4.1 better illustrates this structure.



0	0
1	1
2	3
3	5
4	6
5	9
6	11
7	15
65533	6845
65534	6848
65535	6849

	StarID, A	StarID, B
0	64	22
1	665	667
2	3215	3244
3	77	87
4	244	994
5		

Figure 4.1: Distance and index catalogue structure. The angular distances are the indexes for one array that contains index numbers to a second array that stores the star pairs.

4.4 Algorithm Details.

The algorithm can be split into separate parts. Each time a new centroid is detected, an interrupt gets generated on the PS. Distortion correction is done, and the centroid is saved to a struct. The data structure of each detected star is shown in table 4.1.

Variable	Type	Description
x	float	Detected X-coordinate
y	float	Detected Y-coordinate
I	float	Accumulated Intensity
ux	double	Unit Vector X after correction
uy	double	Unit Vector Y after correction
uz	double	Unit Vector Z after correction
votes	uint16[CATALOGUE_SIZE]	Voting List of all catalogue stars
likelyID	uint16	Likely Catalogue ID
validationVotes	uint8	Number of Validation votes for likely ID
valid	uint8	Flag for valid star.

Table 4.1: Data Structure of Centroids in processor system.

4.4.1 First Voting Round

When there are two or more centroids detected within the current frame, the first round of voting starts. Algorithm 2 provides pseudo-code that describes the first round of voting.

```
let $N$ = number of detected centroids
let margin = defined error margin
if $N \geq 2$ then
    for $i = 0$ to $i = N - 1$ do
        compute the scaled distance between star $N$ and each detected star,
          $\acute{D}_{iN} = (|(\vec{s}_{i-detected} \cdot \vec{s}_{N-detected})| - \text{Diagonal FOV}) \times \text{Multiplication Factor}$
        store value in temp distance array, $R_{iN}$, to be used later.
        let lowerIndex be value in distance catalogue at index [$\acute{D}_{iN}$ - margin]
        let upperIndex be value in distance catalogue at index [$\acute{D}_{iN}$ + margin]
        for $j = lowerIndex$ to $j = upperIndex$ do
            let starA = value in pairs-array at pairs[j][0]
            let starB = value in pairs-array at pairs[j][1]
            vote for starA and starB in the voting lists of centroids $N$ and $i$
        end
    end
end
```

Algorithm 2: First round of voting. This happens as each new centroid is detected

When the end of the frame is reached, two more steps are needed to finish the matching algorithms:

1. Count the voting lists for each detected star.

2. Do a second round of voting to validate the likely ID's of each detected star.

4.4.2 Counting Votes

At the end of the frame, the algorithm loops through the voting list for each detected star and stores the ID of the likely star (most votes) into the struct:

```
let $N$ = number of detected centroids
for $i = 0$ to $i = N - 1$ do
    for $j = 0$ to $j = CATALOGUE\_SIZE - 1$ do
        Find the index with highest value in votelist for Star $i$;
    end
    Store index number, $j$, in likelyID variable.
end
```

Algorithm 3: Counting Votes

4.4.3 Validating ID's

In most cases, the likely ID at this point is the correct ID for the star. There are certain events where false stars are detected. These stars also receive votes, but assigning an ID to a false star would cause problems during attitude determination. It is, therefore, necessary to validate the identification. Validation is done by computing the distances between all likely catalogue stars and comparing it to the distances found between corresponding detected stars. If the difference is less than an error margin, then the star receives a validation vote.

```
let $N$ = number of detected centroids
$R$ is the distance array created earlier in the first round of voting.
let $\epsilon$ be the allowable error.
for $i = 0$ to $i = (N - 1)$ do
    for $j = i + 1$ to $j = (N - 1)$ do
        Compute the distance between catalogue stars based on likely ID of
          detected stars,
        $\acute{D}_{ij} = (|(\vec{S}_{i-likely} \cdot \vec{S}_{j-likely})| - \text{Diagonal FOV}) \times \text{Multiplication Factor}$
        if $|R_{ij} - \acute{D}_{ij}| < \epsilon$ then
            Add a validation vote for detected stars $S_i$ and $S_j$;
        end
    end
end
```

Algorithm 4: Validation Round of Voting.

Valid stars are clustered together, meaning that they would receive a similar amount of validation votes. If the number of validation votes for a star is close to the maximal number of votes among all detected stars, then the star identification is considered valid. This validation process eliminates false matches.

4.5 Star Catalogue

As mentioned, the SAO catalogue is used for reference [1]. This catalogue contains a total of 258997 stars. Due to a short exposure time of the sensor required for a 10Hz update rate, only stars with visual magnitude less than 5 is considered, which decreases the number of stars to only 1583.

4.6 Tests and Results

Matching is first evaluated using simulated star images where the star ID's are known. This is an easy test since simulated star images by design does not contain false stars. The algorithm is tested in a Python scripting language for proof of concept and visual plots. An arbitrary RA and DEC are chosen for the generated image, and a plot is made that shows red crosses at the correct centroid locations along with the catalogue ID. This ground truth image is shown in figure 4.2.

4.6.1 Simulated Star Image

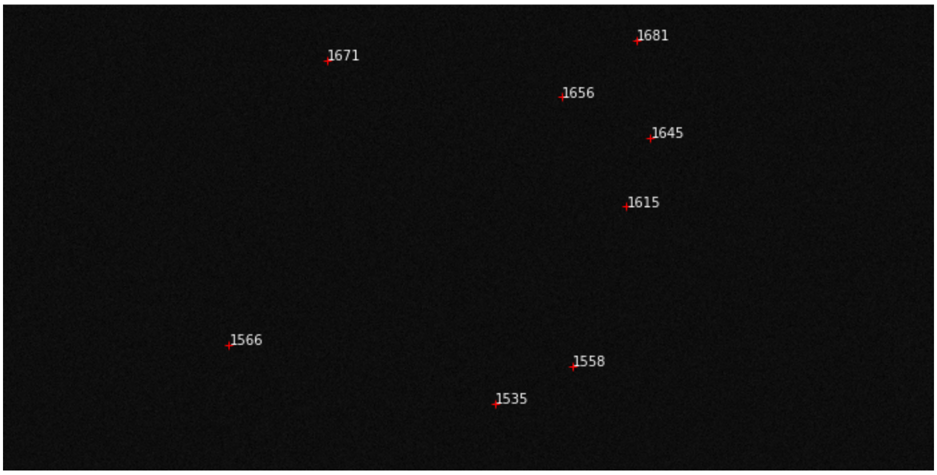


Figure 4.2: Ground truth generated star image with real ID's and centroid locations.

The centroid detection and matching algorithms are run on a PC in a python script, and the detected centroid locations along with the matching results are shown in figure 4.3. Notice that the results are identical to the ground truth, demonstrating an accurate algorithm.

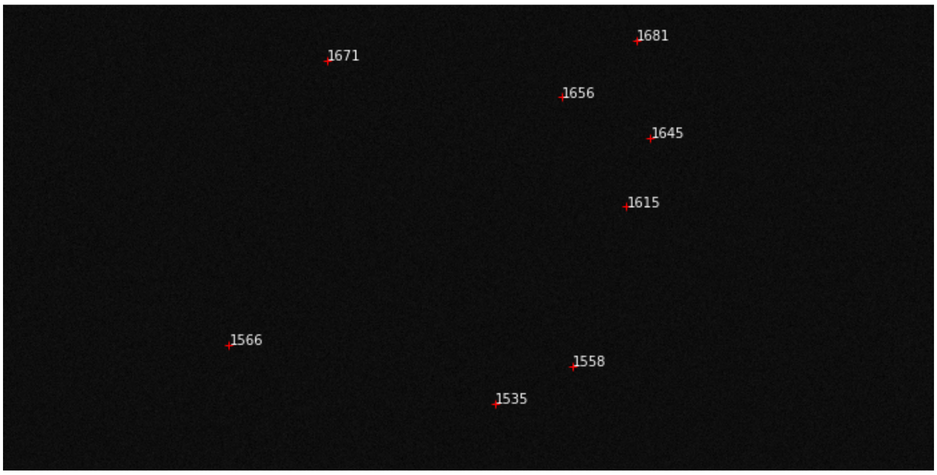
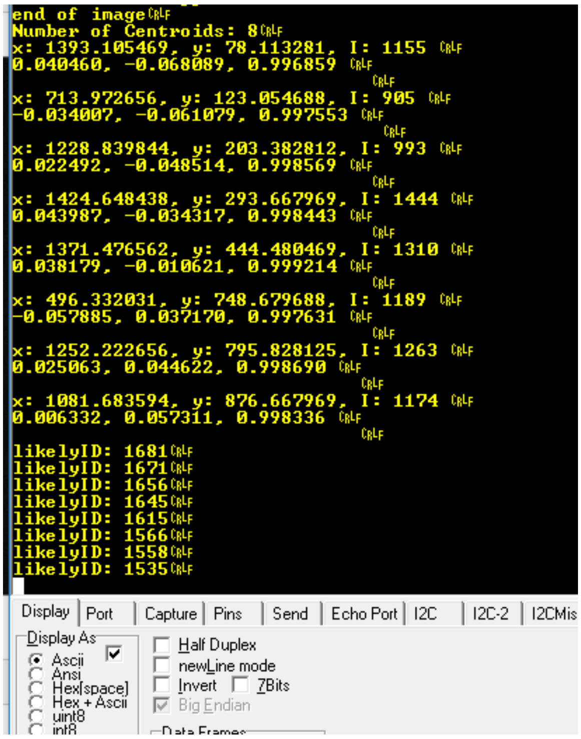


Figure 4.3: Results from Python matching and detection algorithm. Notice that the results are identical to figure 4.2.

The same image is loaded onto a flash-drive and tested on the SoC using the emulated sensor. The image is streamed through the FPGA and centroids are extracted in real-time. The matching algorithm is written in C and is done on the PS side. Results are passed back to the PC via UART. Figure 4.4 shows a screenshot of UART output. Notice that the ID's of the detected stars are the same as the ground truth. These tests were performed earlier in development, and the star catalogue contained more stars (Vmag < 5.5) which explains the different ids from later tests.



```
end of imageCRLF
Number of Centroids: 8CRLF
x: 1393.105469, y: 78.113281, I: 1155 CRLF
0.040460, -0.068089, 0.996859 CRLF
                                 CRLF
x: 713.972656, y: 123.054688, I: 905 CRLF
-0.034007, -0.061079, 0.997553 CRLF
                                  CRLF
x: 1228.839844, y: 203.382812, I: 993 CRLF
0.022492, -0.048514, 0.998569 CRLF
                                 CRLF
x: 1424.648438, y: 293.667969, I: 1444 CRLF
0.043987, -0.034317, 0.998443 CRLF
                                 CRLF
x: 1371.476562, y: 444.480469, I: 1310 CRLF
0.038179, -0.010621, 0.999214 CRLF
                                 CRLF
x: 496.332031, y: 748.679688, I: 1189 CRLF
-0.057885, 0.037170, 0.997631 CRLF
                                 CRLF
x: 1252.222656, y: 795.828125, I: 1263 CRLF
0.025063, 0.044622, 0.998690 CRLF
                                CRLF
x: 1081.683594, y: 876.667969, I: 1174 CRLF
0.006332, 0.057311, 0.998336 CRLF
                                CRLF
likelyID: 1681CRLF
likelyID: 1671CRLF
likelyID: 1656CRLF
likelyID: 1645CRLF
likelyID: 1615CRLF
likelyID: 1566CRLF
likelyID: 1558CRLF
likelyID: 1535CRLF
```

Display As ☑
● Ascii
○ Ansi
○ Hex[space]
○ Hex + Ascii
○ uint8
○ int8

□ Half Duplex
□ newLine mode
□ Invert □ 7Bits
☑ Big Endian

Figure 4.4: Results from SoC using the emulated sensor. This is a screenshot from UART output.

4.6.2 Real Star Image

The matching algorithm needs to be validated using real star images. A similar test is done, except that the real image is loaded onto the flash drive of the emulated sensor. This test is also important for verifying the distortion correction algorithm explained in section 5.1 of the next chapter. An image with known calibration parameters is passed through the entire system. The corrected centroid results along with corresponding ID's are saved to a PC and plotted on top of the image, as shown in figure 4.5. Since this is a photograph of Crux constellation, the real ID's can be found manually in the catalogue. The RA and DEC of Alpha Crux, Beta Crux, Delta Crux and other bright stars are known and used to search for their ID's in the reduced catalogue. The figure shows the original image with the star names below.

The positions and catalogue IDs of the stars can be seen in table 4.2 below.

	RA	**DEC**	**starlist ID**
Acrux	186.6497833	-63.09902778	169
Beta	191.9303875	-59.68870556	246
Delta	183.7861458	-58.74883611	276
Gamma	187.7914708	-57.11321111	296
Epsilon	185.3397917	-60.40123889	234

Table 4.2: Positions and star IDs of Crux constellation according to the catalogue.

These results conclude that the matching algorithm works as expected. The next chapter demonstrates more tests using images from the Gecko.

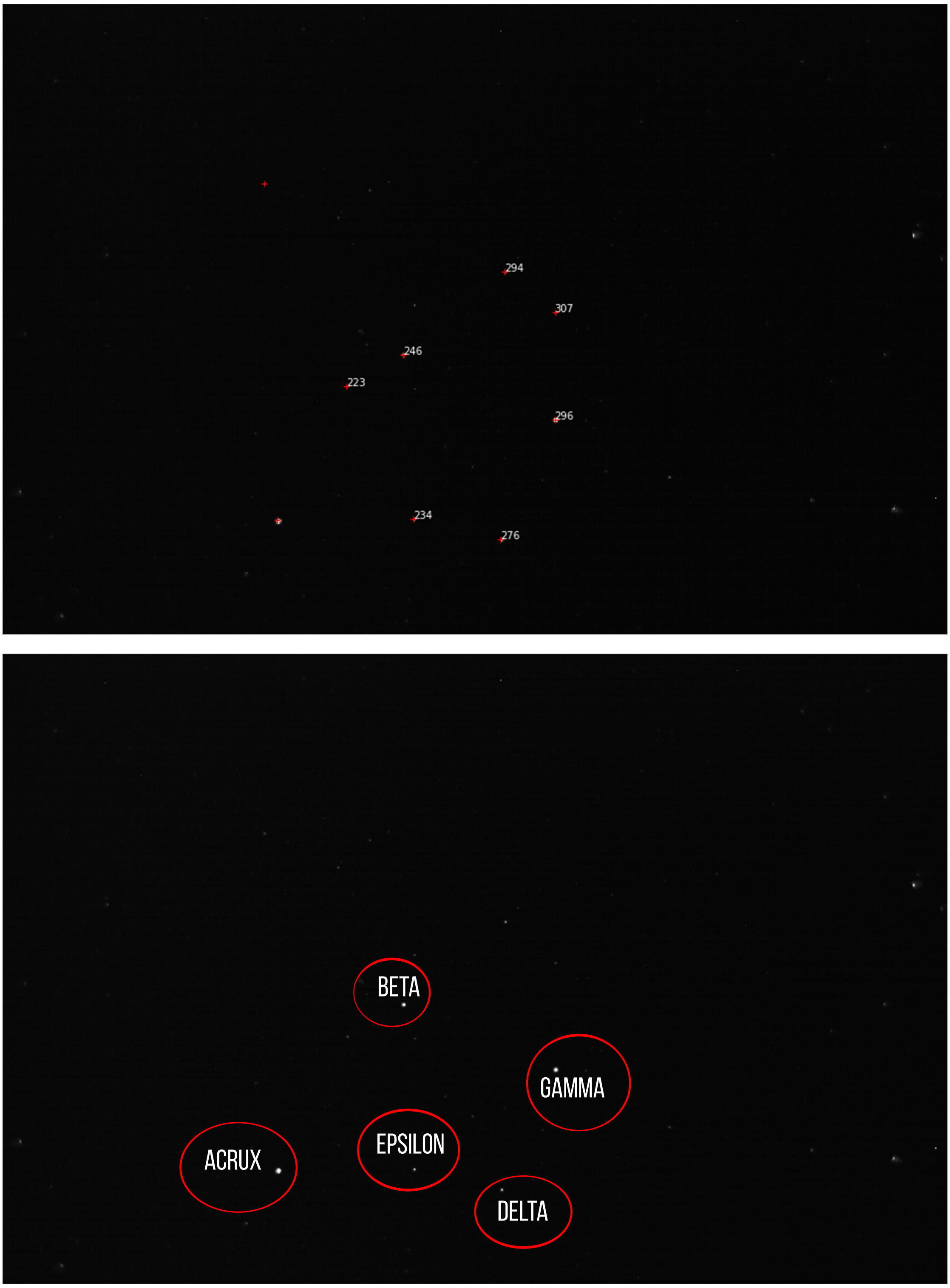


Figure 4.5: Matching results using real images.

Chapter 5

End-To-End Verification

The final end-to-end test involves pointing the imager connected to the SoC at the night sky and receiving an attitude output at the desired update rate. There are two important algorithms needed to present an end-to-end system. First, lens distortion correction must be done on real star images before identification (matching) can start. This is discussed in section 5.1. After matching, the attitude determination algorithm is used to calculate the inertial orientation of the star tracker. This algorithm is briefly explained in section 5.3.

Development Setup

Figure 5.1 shows the hardware setup for prototyping and debugging. The image data is streamed from the sensor via LVDS signals directly onto the PL. The entire centroid detection algorithm is designed on the PL and detected centroid locations are passed onto the PS with an interrupt via an AXI memory-mapped interface. Data is passed from the PS to a PC via UART to show results. During hardware debug, a JTAG interface is connected to the PL side, making it possible to probe into the HDL block design using an ILA. The JTAG interface is also used to program the bitstream onto the FPGA during development.

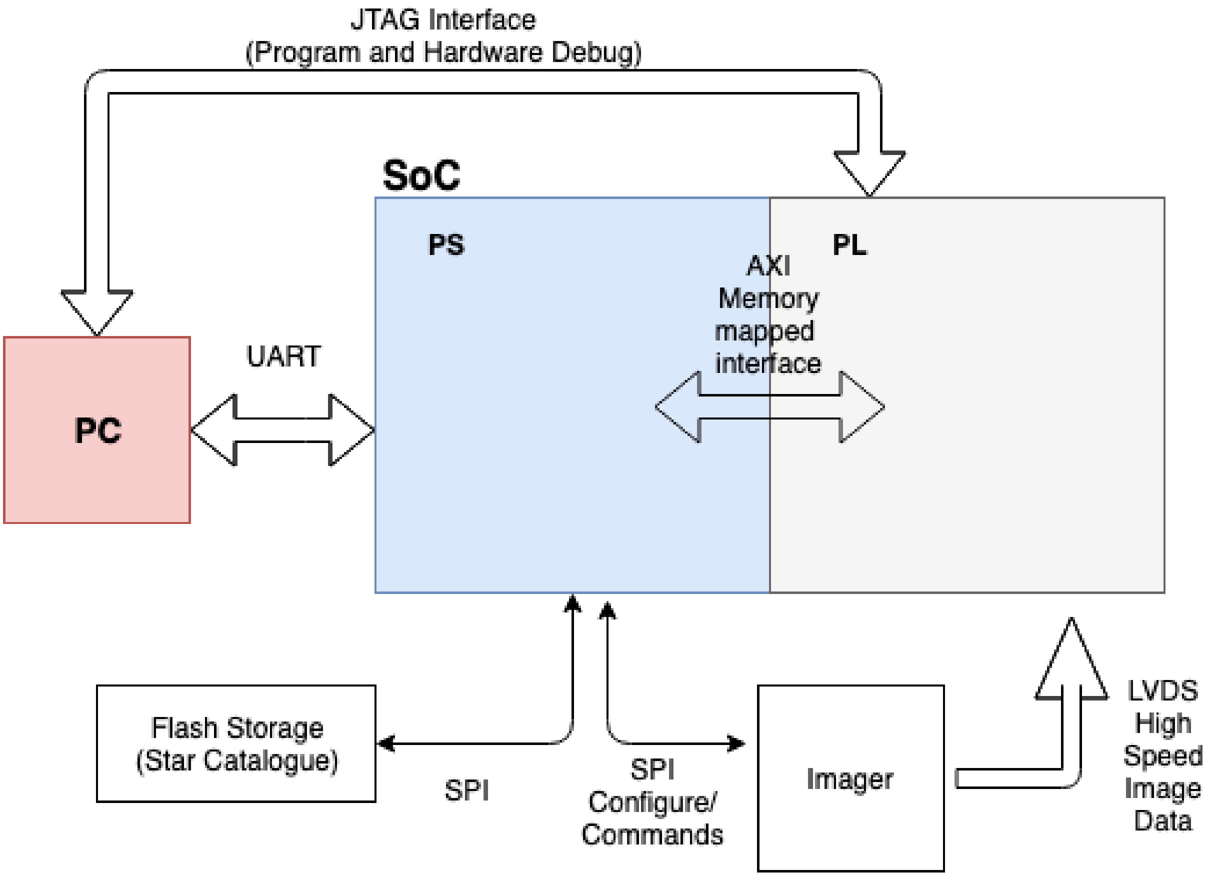


Figure 5.1: **Hardware Block Diagram**. This is the hardware setup for prototyping and debugging the star tracker. The PL side of the SoC does all of the sensor interfacing and centroid detection while the matching algorithms take place on the PS side.

5.1 Lens Distortion Correction and Vector Conversion

A lens distortion-correction function explained by Jacobs [14] is used during calibration. This function ignores the effects of tangential distortion but describes the radial distortion, coma and field curvature. Jacobs' thesis [14] contains more detail regarding the technique. The correction function requires the following variables:

- x_{ref}. This is the optical centre X in pixels.
- y_{ref}. This is the optical centre Y in pixels.
- f. Focal length in pixels.
- k_1 to k_3. Radial coefficients 1 to 3.

For each x- and y-coordinate of detected centroids, the following equations shows how radial distortion is corrected. First the optical centre is subtracted from each coordinate:

$$x' = x - x_{ref} \tag{5.1}$$

$$y' = y - y_{ref}, \tag{5.2}$$

The radial distance, r, is calculated:

$$r = \sqrt{x'^2 + y'^2}. \tag{5.3}$$

The radial distance is corrected using the function explained by Jacobs:

$$r_{\text{corrected}} = r + k_1 r + k_2 r^2 + k_3 r^3. \tag{5.4}$$

The corrected x- and y-coordinates are found:

$$x_{\text{corrected}} = \frac{r_{\text{corrected}} \times x'}{r} \tag{5.5}$$

$$y_{\text{corrected}} = \frac{r_{\text{corrected}} \times y'}{r} \tag{5.6}$$

A pointing vector to the star in image plane coordinates are returned that will be used to calculate distances between stars during matching algorithms:

$$\vec{s} = \begin{bmatrix} x_{\text{corrected}} \\ y_{\text{corrected}} \\ f \end{bmatrix} \tag{5.7}$$

The vector can be normalized to avoid overflow when performing dot-product operations during matching:

$$\vec{s}_{\text{normalized}} = \frac{\begin{bmatrix} x_{\text{corrected}} \\ y_{\text{corrected}} \\ f \end{bmatrix}}{||\vec{s}||} \tag{5.8}$$

This entire process is done on the PS side in C-code. When a new valid centroid position is detected, an interrupt is generated, and a new struct is initialised. The uncorrected x-and y-values are saved to the struct after which the distortion correction and vector conversion are applied and saved.

5.2 Calibration

The calibration procedure requires photographs of the night sky using the exact imager and optical setup as in the final star tracker. The Gecko imager provided by SCS is fitted with a small 25mm, f1.4 CCTV lens. The camera setup is taken outside to capture calibration photos. The calibration algorithm written by Dr Visagie and based on the thesis by Jacobs [14] involves manually identifying a few of the stars in the image to initialise the iterative optimisation technique. It is therefore important to capture a known constellation. The crux constellation is captured, shown below in figure 5.2. These calibration images were captured with longer exposure time to allow more stars and therefore, better calibration results. Various photographs are taken at different positions that are used to calculate the calibration parameters. The lens used on the camera is not of the highest quality and precision. Radial lens-blur becomes very visible at the edges of the image, resulting in a tilt-lens type of look. This blur is due to the fact that the sensor form factor is not properly matched with the optics. The useful field-of-view of the optics does not extend to the entire camera sensor area. The 25mm focal length also leads to a relatively small FOV. It is recommended to use a higher precision lens with a wider focal length and slightly larger aperture for better star tracker results.

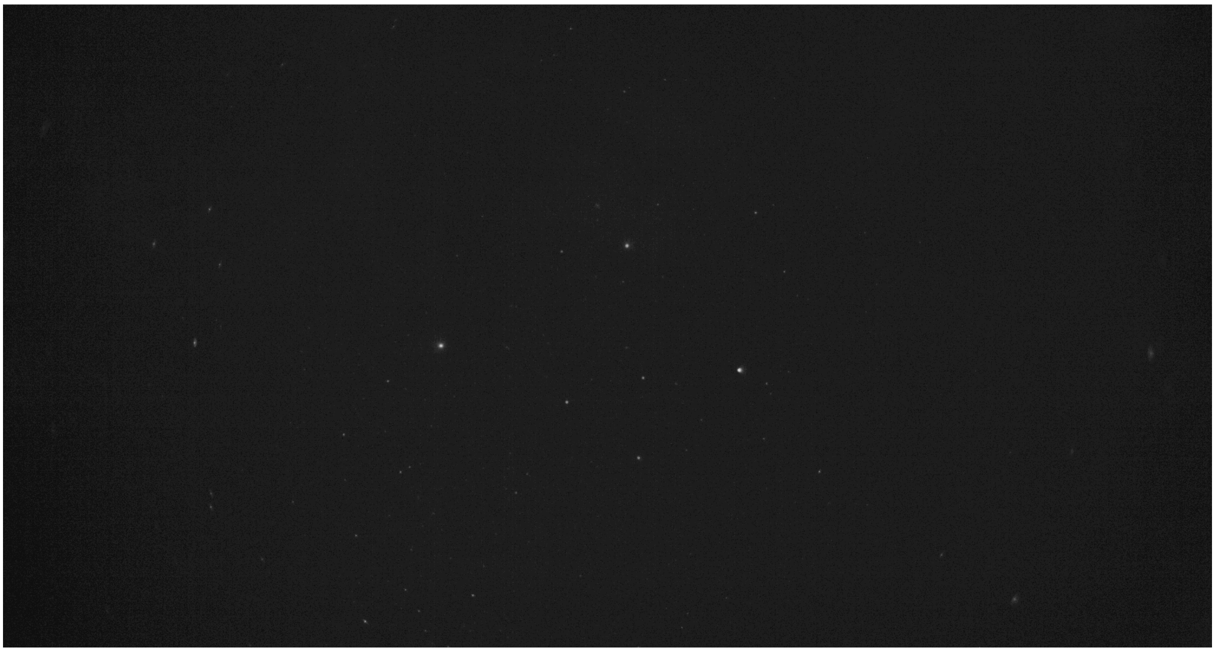


Figure 5.2: Example of calibration image captured. A longer exposure is used to allow more light and more stars on the image that will result in better calibration results.

Another important thing to mention is that the Gecko imager used in the prototype hosts a CMV2000 sensor with an RGB filter. This produces a Bayer pattern if not compensated

for in post-processing. Ideally, a monochrome sensor with a very high SNR is desired to capture clear star images at short exposure times.
Due to long wait times for a new sensor and lens, it was decided to continue using the Gecko imager setup to demonstrate an end-to-end system. The calibration parameters calculated is shown below:

- Focal Length (in pixels): 4824.5 (25 mm lens)
- Optical Centre X: 911.5 (pixels)
- Optical Centre Y: 576.7 (pixels)
- Radial Distortion coefficient, k1: -0.0013579185
- Radial Distortion coefficient, k2: 0.0000017422
- Radial Distortion coefficient, k3: 0.0000000146

5.2.1 Tests and Results

The calibration parameters are tested independently using a python script and a test photograph taken with the same camera setup. The test images were taken with shorter exposure times (less than 100ms) to achieve the desired 10Hz update rate. Significantly fewer stars are visible in the test image due to a shorter exposure. The sensor gain was also increased, which leads to a higher noise floor. Previously validated detection and matching algorithms were used alongside these calibration parameters on the test images. Figure 5.3 shows the detected star ID's of a test image.

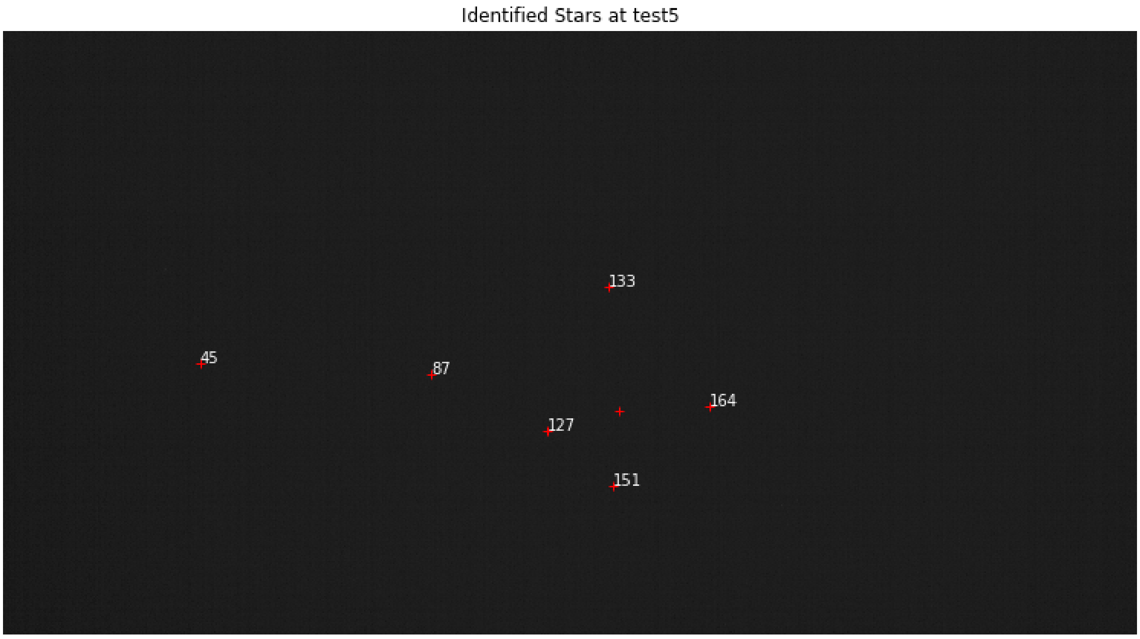


Figure 5.3: Test results using calibration parameters and previously validated detection and matching algorithms. Much shorter exposure time is used on the test image; therefore, fewer stars are visible.

	RA	DEC	starlist ID
Acrux	186.6497833	-63.09902778	87
Beta	191.9303875	-59.68870556	133
Delta	183.7861458	-58.74883611	151
Gamma	187.7914708	-57.11321111	164
Epsilon	185.3397917	-60.40123889	127

Table 5.1: Positions and star IDs of Crux constellation according to the updated catalogue.

Upon inspecting table 5.1, it can be seen that most of the correct ID's are identified. This confirms that the calibration parameters are close enough for matching to work accurately. The ID's are different from table 4.2 after the catalogue is updated to only include stars brighter than visual magnitude 5. The accuracy of the calibration parameters affects the sensor-body vectors that may affect the final attitude determination results, as explained in the next section.

5.3 Attitude Determination

Attitude determination is the final step in the star tracker process. In the context of this work, it refers to determining the attitude of the spacecraft, quantified by some rotation away from a reference, based on measurements from an image. The QUEST (Quaternion Estimator) algorithm is used and explained in this section.

5.3.1 Background

QUEST is a *point* method algorithm, meaning that it uses unit vector measurements from a single image and does not take prior attitude or dynamic information into account, therefore providing a LIS solution. QUEST, along with almost all point method algorithms are based on a classic problem by Grace Wahba [11]. The problem has to find the orthogonal matrix that minimises the least-squares estimate of attitude by the loss function:

$$J(\mathbf{A}) = \frac{1}{2}\sum_{k=1}^{N} \omega_k |\mathbf{v}_{kb} - \mathbf{A}\mathbf{v}_{ki}|^2, \tag{5.9}$$

where $\mathbf{v}_{kb}$ is the kth calculated (detected) vector in sensor body coordinates, $\mathbf{v}_{ki}$ is the matched reference vector from the catalogue in inertial coordinates, ω_k is an optional positive weight that depends on the confidence measurement and $\mathbf{A}$ is the matrix which transforms the reference vectors into the sensor body frame - essentially the rotation matrix that needs to be solved. Due to noise and detection errors, the matrix will not perfectly map $\mathbf{v}_{kb}$ and $\mathbf{v}_{ki}$, however, it is considered the optimal solution by this loss function.

Davenport's q-Method

Paul Davenport provided the first usable solution to Wahba's problem (equation 5.9) [31]. The q-method restates the loss function using quaternions in a way that it becomes an eigenvalue problem - finding the largest eigenvalue and corresponding eigenvector of a matrix. The derivation of the q-method is not presented here, but it can be found in [6]. The biggest problem with the q-method is the high computational cost required to solve the eigenvalue problem [31]. On conventional microprocessors in an embedded system environment, it is too big of a computational burden for fast star trackers.

5.3.2 QUEST

The QUEST algorithm was developed by Shuster as an approximation to the q-method to bypass the expensive eigenvalue problem [10]. The full derivation can be found in [30] but essentially it ends in a problem of solving simultaneous linear equations using Gaussian elimination. The final form is expressed in equation 5.10 and involves solving for **p**:

$$[(\lambda_{opt} + \sigma)\mathbf{1} - \mathbf{S}]\mathbf{p} = \mathbf{Z}, \tag{5.10}$$

where

$$\begin{aligned}
\mathbf{p} &= \text{Rodriguez elements, this needs to be solved for} \\
\mathbf{B} &= \sum_{k=1}^{N} \omega_k \mathbf{v}_{kb} \mathbf{v}_{kb}^T \\
\lambda_{opt} &= \sum \omega_k \\
\sigma &= tr(\mathbf{B}) : \text{The sum of elements on the diagonal} \\
\mathbf{S} &= \mathbf{B} + \mathbf{B}^T \\
\mathbf{Z} &= [B_{23} - B_{32} \; B_{31} - B_{13} \; B_{12} - B_{21}]
\end{aligned}$$

The attitude quaternion, $\bar{q}$, can be calculated using the following equation:

$$\bar{q} = \frac{1}{\sqrt{1 + \mathbf{p}^T\mathbf{p}}} \begin{bmatrix} \mathbf{p} \\ 1 \end{bmatrix}, \tag{5.11}$$

where

$$\bar{q} = \begin{bmatrix} q_1 \\ q_2 \\ q_3 \\ q_4 \end{bmatrix} = q_1\mathbf{i} + q_2\mathbf{j} + q_3\mathbf{k} + q_4. \tag{5.12}$$

Shuster claims that QUEST is up to 1000 times faster than the q-method [30]. It has become a popular algorithm for three-axis-attitude estimation. There is many information around QUEST in literature which makes it fast to implement. Modern attitude determination algorithms have been developed but do not show such a significant improvement, and makes QUEST still relevant and appropriate as a choice [10]. A complete comparison of attitude determination algorithms can be found in a paper by Mortari and Markley [23].

5.3.3 Tests and Results

QUEST is validated by using simulated images (section 4.6.1) where the ground truth attitude is known. A batch of images is generated starting at Right Ascension 0 to 180 degrees and Declination from -90 to 90 degrees. 10-degree step size is taken, and a total of 324 images are generated. These images are passed through validated detection and matching algorithms. QUEST then use these results as input. The average error in arcseconds over all of these images are found:

- **Average Right Ascension Error:** 1.06344 arcseconds.
- **Average Declination Error:** 0.70418 arcseconds.

Figure 5.4 shows a plot of the RA and DEC error in each image

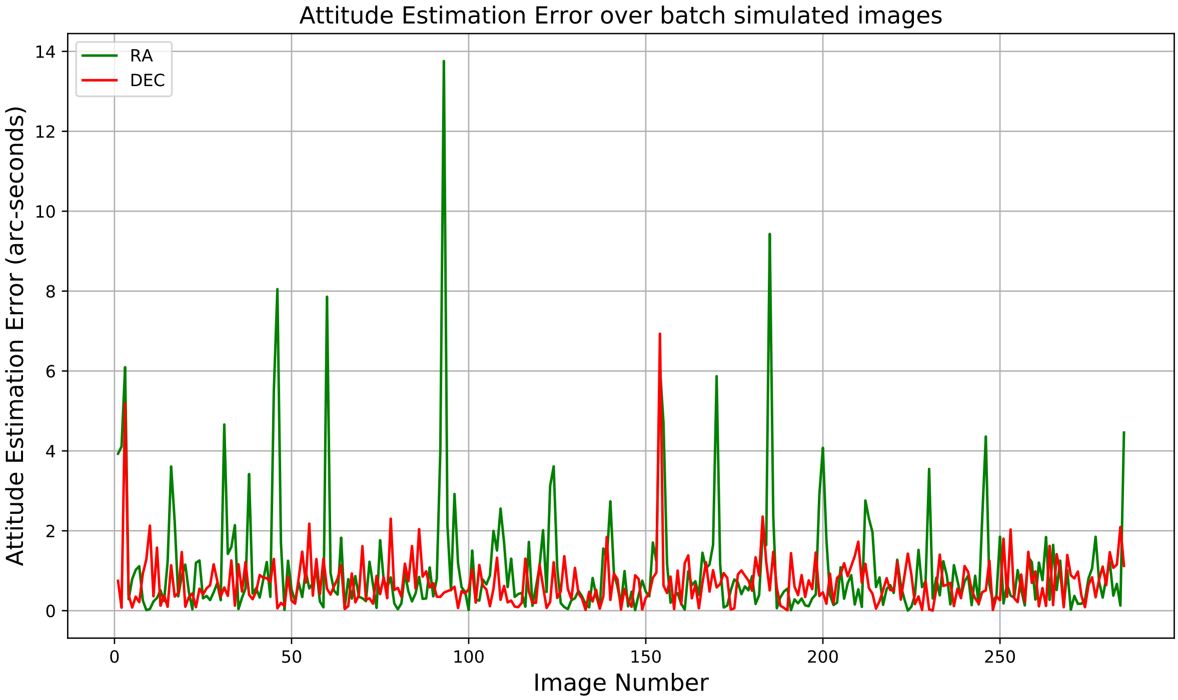


Figure 5.4: Plot of RA and DEC errors (arc-second) for each simulated image.

These results demonstrate a high-accuracy attitude determination algorithm. It is important to note that these tests are made with simulated images and real star photos have less accurate results due to noise on the image leading to less accurate centroid detection, errors in calibration and other external factors.

5.4 Laboratory Tests

Before the night sky tests are done with the end-to-end system, a final laboratory test is done to validate the complete star tracker and all of the algorithms on the SoC device. Test images (similar to figure 3.16) is tested offline using Python scripts and then loaded onto the sensor emulator to test the star tracker. This setup streams the image at a slow speed but verifies star detection using a real image taken with the Gecko at the same exposure and gain settings as would be used during the operation. Distortion correction, matching and QUEST is also validated on the SoC. The detected centroids before distortion correction is shown in table 5.2 below.

VHDL X	VHDL Y	Python X	Python Y	Error X	Error Y
1093.566406	461.117188	1093.567839	461.120603	0.0014332	0.003415015
355.886719	599.941406	355.8888889	599.9444444	0.002169889	0.003038444
772.929688	619.488281	772.9304636	619.4917219	0.000775576	0.003440854
1276.460938	677.824219	1276.463973	677.8276094	0.00303506	0.003390428
1113.164062	685.441406	1113.166667	685.4444444	0.00260467	0.003038444
982.914062	722	982.9178082	722	0.003746219	0
1101.097656	820.363281	1101.1	820.3666667	0.002344	0.003385667

Table 5.2: Final Lab Test Centroid Comparison

For these results, a threshold value of 51 was used. A margin of 4 pixels also leads to the most accurate results for the test images. These results confirm that the centroid detection works as expected using images from the Gecko with an exposure time less than 100ms.

The attitude output results are close to identical and displayed in table 5.3 below. These tests confirm that the entire flow of algorithms works as expected on the star tracker. Although the ground truth attitude of the test image is unknown, it can be evaluated by inspection: It is known that the location of the Crux constellation lies at 187,5 RA and -60 DEC. Given the attitude output in table 5.3 and the fact that Crux lies close to the centre of this test image, the results are deemed valid. This rough validation along with the extensive tests results using simulated images (section 5.3.3) provides enough confidence to assume a full working system at this slow speed.

	SoC (Degrees)	PC/Python (Degrees)	Difference (Arcseconds)
RA	188.441653827	188.441711122	0.2062
DEC	-61.5947964415	-61.5948537373	0.2062

Table 5.3: Attitude Output Results from SoC full system compared to off-line PC results

5.5 Imager Setup and Interface

5.5.1 Imager Test Stack

SCS provided Imager Test Software (ITSW) along with an Imager Test Stack (ITS) that is used to test the Gecko. With the Test Stack connected to a PC via Ethernet, it is possible to change register values on the Gecko and display and capture a live stream of raw image data. Figure 5.5 shows this test setup. The ITS enabled testing various exposure and gain settings and capturing images that are used for calibration and testing (section 5.2). The exposure setting on the imager is configured to have an integration time of roughly 99 ms. This is the maximum exposure time allowed to achieve the desired 10Hz update rate. More details on the final timing profile can be seen in section 5.8. With this exposure setting fixed, the gain of the sensor must be increased to allow for enough stars to be visible. An increased ADC gain lifts the noise level in the image and results in less accurate centroid detection, as described in section 3.9.3. Sensors with lower noise at higher gain settings will have better results in a star tracker.

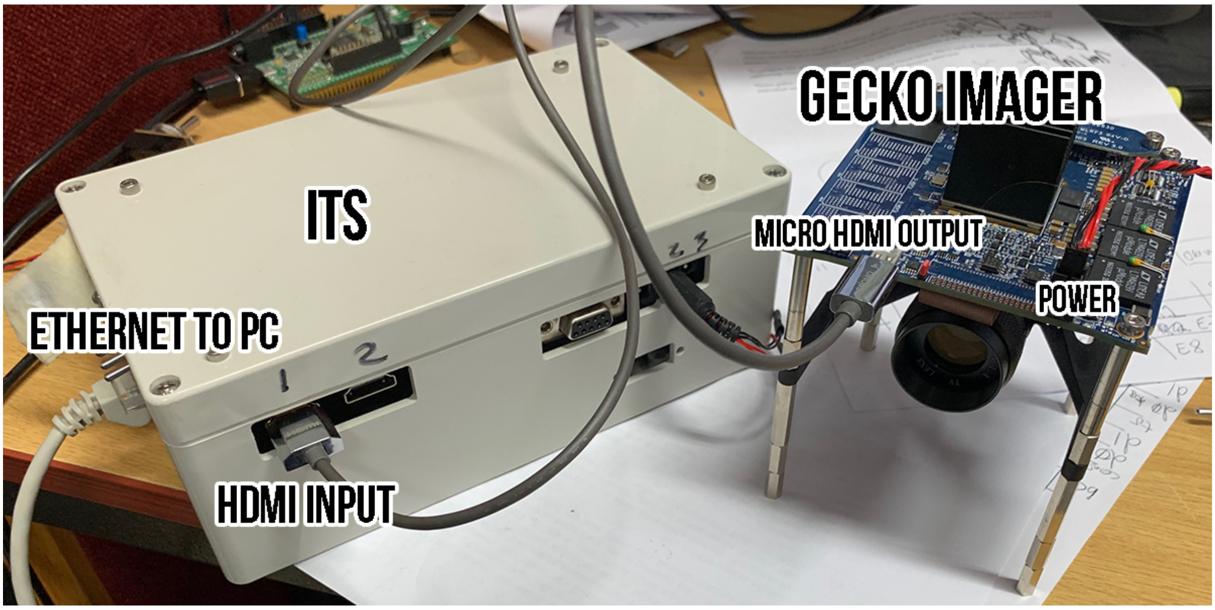


Figure 5.5: Imager Test Stack (ITS) and Gecko setup configuration.

5.5.2 Imager Interface on the FPGA

Designing an interface for a high-speed LVDS peripheral in VHDL is a complete challenge on its own, faced with numerous timing issues and bit alignment errors. Grob [12] wrote an entire article on the design of high-speed LVDS interfaces in VHDL. The Gecko has four LVDS output lines: Clock, Control, Data0 and Data1. When the imager is turned on, a training pattern is present on the outputs. This pattern is necessary to synchronise with the receiver side. When streaming starts, the control signal change to 0x83, Data0 transports the odd pixel while Data1 transports the even pixel. Figure 5.6 shows this word-level protocol. The Gecko control signal does not indicate when the end of a frame is reached, therefore pixels are counted at the pixel split block (section 3.5.2), and a flag is activated from the PL once the end of the frame is reached.

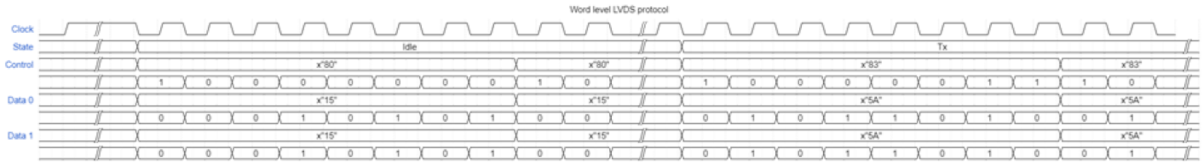


Figure 5.6: Gecko word-level LVDS protocol.

5.5.3 Bayer Filter on RGB sensor

As the Gecko imager was never intended to be used for a star tracker, the sensor has a Bayer filter to capture colour images. A Bayer filter is a colour filter array placed in front of the sensor which makes each pixel more (or less) sensitive to certain wavelengths light (red, green or blue). Figure 5.7 shows an illustration of a Bayer filter. The raw output of an image taken such a filter contains a *Bayer* pattern when viewed as monochrome values. De-mosaicing algorithms interpolate these values using various techniques to obtain a colour image. For a star image, only monochrome pixel values are needed, and

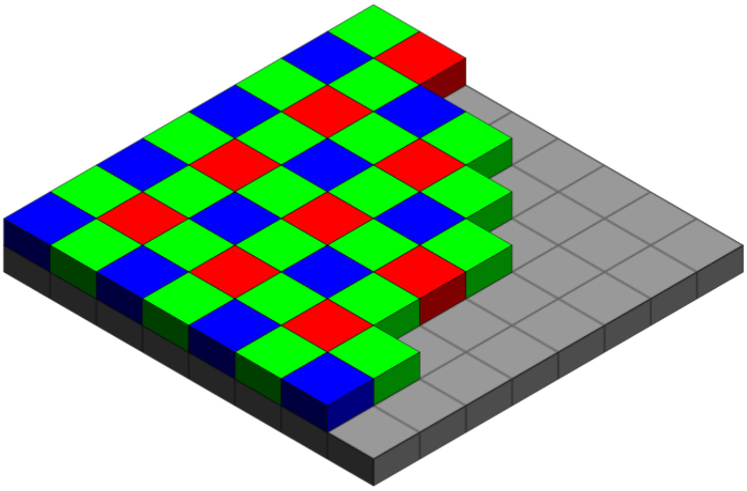


Figure 5.7: Bayer example ons RGB image sensors. Image by Colin M.L. Burnett.

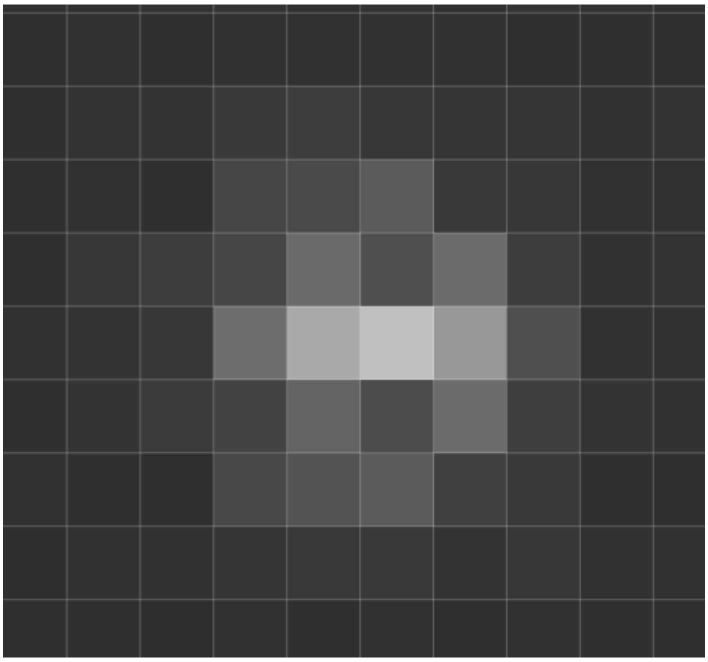


Figure 5.8: Raw zoomed-in pixels of a star captured with the Gecko. Notice how the Bayer pattern is visible in the PSF.

there is no demosaicing performed. Figure 5.8 shows a zoomed-in raw image of a star captured with the Gecko imager. Notice how the Bayer filter creates an undesired pattern in the PSF of the star. It now looks less like a 2D Gaussian distribution when compared to figure 3.1, and will most likely lead to less accurate centroid detection. It is highly recommended to use a sensor without such a filter.

5.6 Zynq PS Flow of Events

The entire flow of events is controlled from the Zynq PS. Similar to conventional embedded system designs, initialisation is done on start-up after which the processor goes into the main loop. The start-up sequence is shown as a flow diagram in figure 5.9. Green blocks indicate PS specific events; red blocks indicate events related to the PL where AXI communication is done, and blue blocks indicate sensor control commands that happen via SPI.

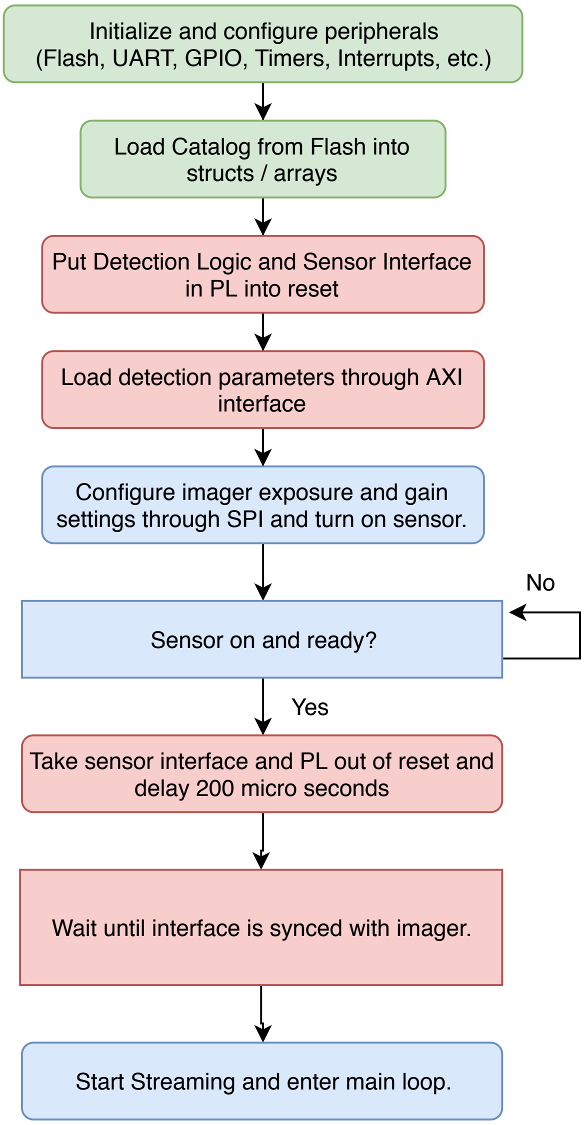

Initialize and configure peripherals (Flash, UART, GPIO, Timers, Interrupts, etc.)
Load Catalog from Flash into structs / arrays
Put Detection Logic and Sensor Interface in PL into reset
Load detection parameters through AXI interface
Configure imager exposure and gain settings through SPI and turn on sensor.
Sensor on and ready?
No
Yes
Take sensor interface and PL out of reset and delay 200 micro seconds
Wait until interface is synced with imager.
Start Streaming and enter main loop.

Figure 5.9: Start-up initialization sequence on the Zynq PS before entering the main loop.

Once the initialisation sequence is complete, the PS enters the main loop. For the test setup, a limited number of frames is chosen after which the entire process stops. The reason for this is that after each frame, the results (RA, DEC, Number of Stars, First 10 detected centroids) are saved as a log-entry in RAM. After the desired number of frames is reached, the log data is sent to the PC via UART. For flight-ready star trackers, the system typically runs indefinitely, and results are sent after each frame. The flow of the main loop is shown in figure 5.10 and briefly explained below.

When entering the main loop, the image exposure, read-out and detection happen independently (in parallel) to the PS. The end-of-frame indicator is checked by reading a register on the PL. Synchronisation errors are also checked, should the PL lose sync with the sensor. In such a case, the Gecko and detection algorithm is reset and re-synced. When a centroid is detected, the PL causes an interrupt to the PS that activates a flag. When a new centroid is detected, and the number of stars for the current frame is less than the maximum allowed, the centroid is processed, distortion correction is done, and the results are saved to a struct. If the number of detected stars at this point is greater or equal to 2, then the first round of voting is done (Matching). When the end of the image is reached, streaming is stopped, and the PL logic (centroid detection) is put into reset. The image interface is not put into reset as it is not necessary to re-synchronise since the Gecko transfers a training pattern while in idle. Votes for each star is counted, and validation (second round of voting) is performed. If more than three valid stars are identified, then QUEST is performed, the results are saved to a log, the data structures are reset on the PS, and the PL is taken out of reset, followed by a short delay. The sensor streaming is started, and the PS goes back to the start of the loop.

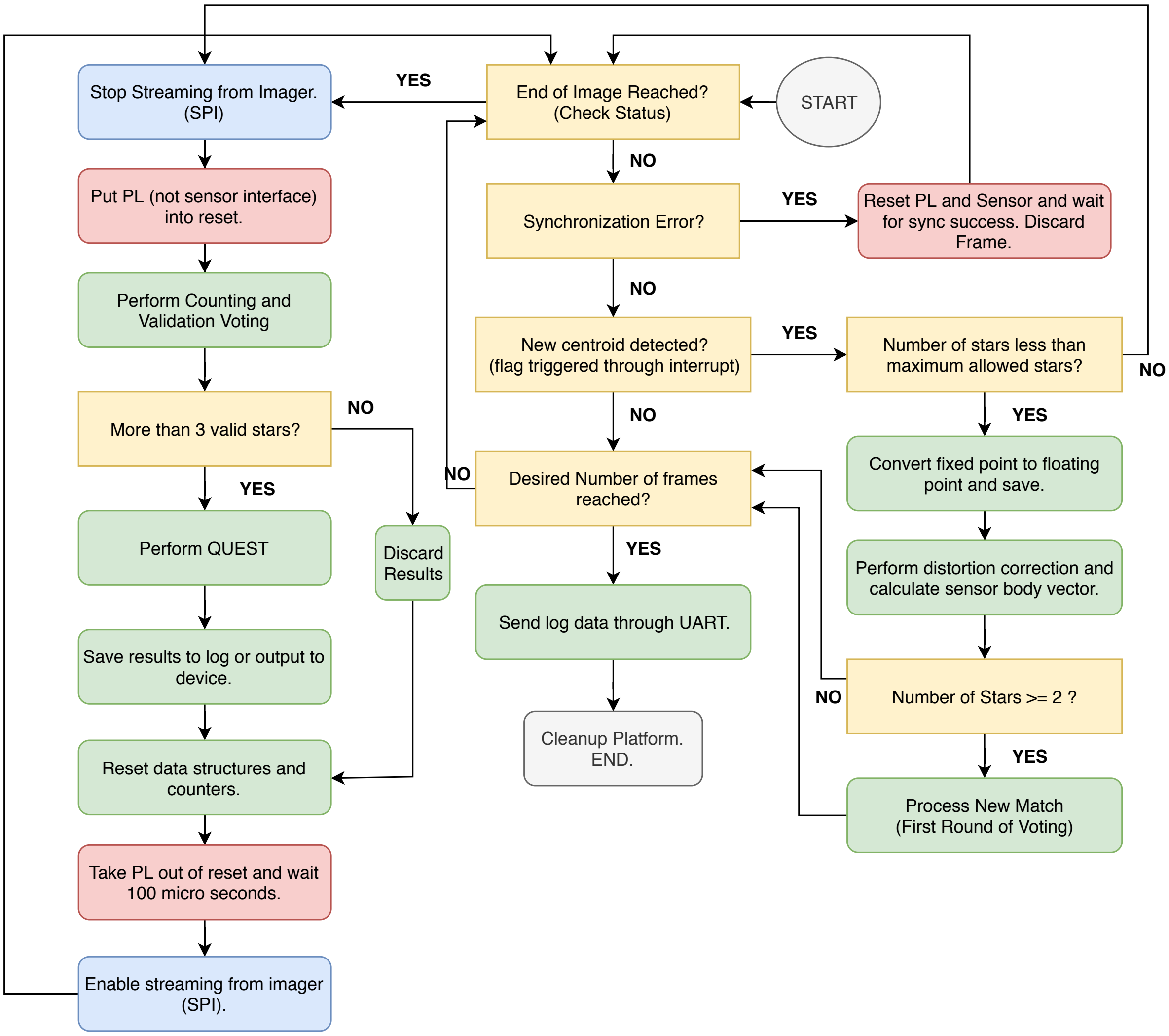


Figure 5.10: Flow diagram of main loop in Zynq PS. Yellow blocks indicate decision making, green blocks indicate PS specific functions or events, red blocks indicate PL communication or commands via AXI interface, and blue blocks indicate imager commands via SPI.

5.7 Final Results

The purpose of this test is to demonstrate that the complete system, operating under the night-sky, can output correct quaternions at a rate of 10Hz. The imager and development board is set up on the roof of the engineering building. This test setup is shown in a photograph in figure 5.11. Because of the inadequacies of the camera that were discussed earlier, it is anticipated that the results that are obtained in this test will not reflect high pointing accuracy. The attitude determination accuracy of the system is not verified in this test, but rather the 10Hz update rate. The camera is pointed towards the Crux constellation because this constellation contains enough bright stars that should be identifiable with the existing optics and gain and exposure settings. An exposure time of 99 ms was used, and the star tracker was run for 2000 frames at 10 Hz (200 seconds). After 2000 frames, the results are sent to the PC via UART containing the RA, DEC, number of stars detected, centroid locations and matched IDs.

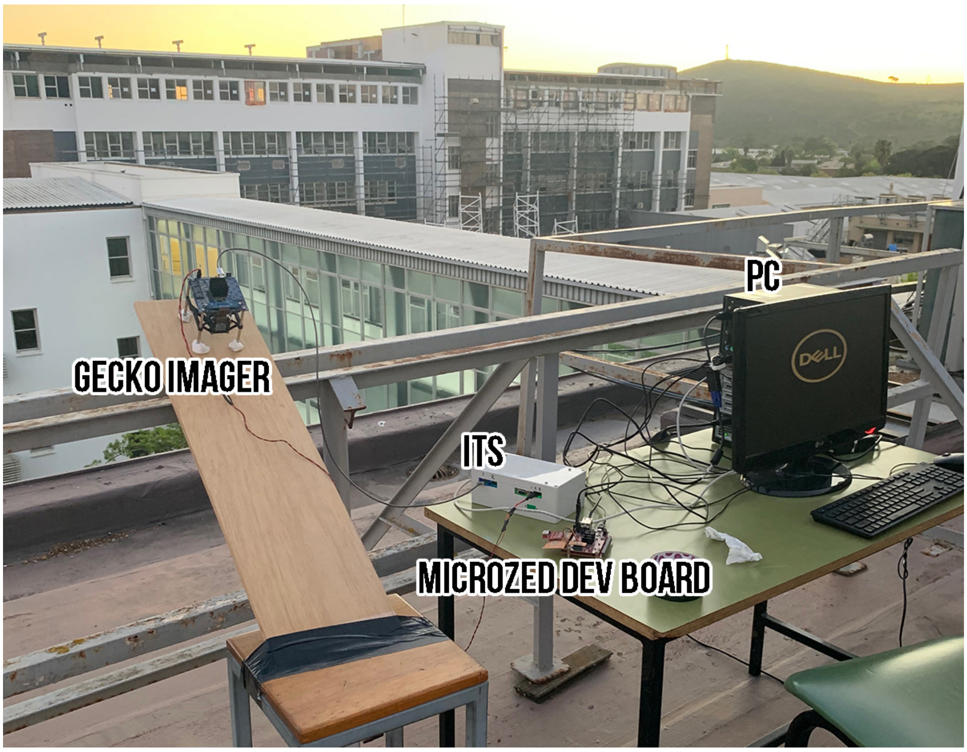


Figure 5.11: Outdoor Test Setup for end-to-end test.

The raw results indicated that the majority of frames had successful star identification and QUEST output; however, a few incorrect RA and DEC results were also obtained for some frames. Upon investigation, it was found that these errors can also be attributed to the unsuitable camera for capturing star images at such a high rate. The high ADC gain, necessary to capture stars, resulted in a high noise floor. The increased noise affects centroid accuracy. The Bayer filter results in a further distorted centroid location. If the centroid error is too high, the matching algorithm fails to correctly identify the catalogue stars because the angular separation falls outside of the allowed margin. The low number of detected stars also affects the performance of matching validation. The centroid detection, however, proved to work more consistently with sub-pixel accuracy, even at such high noise levels and with the undesired Bayer pattern.

5.7.1 Field Test Results

Ignoring frames that had failed in attitude determination due to various reasons explained, the RA output of valid frames are plotted against the frame number. Notice that a clear slope is formed. We expect this slope to correspond with the angular rotation speed of the earth. The slope is calculated as 0.0041751 degrees per second over the 200-second interval. The angular rotational speed of the earth is calculated for one sidereal day:

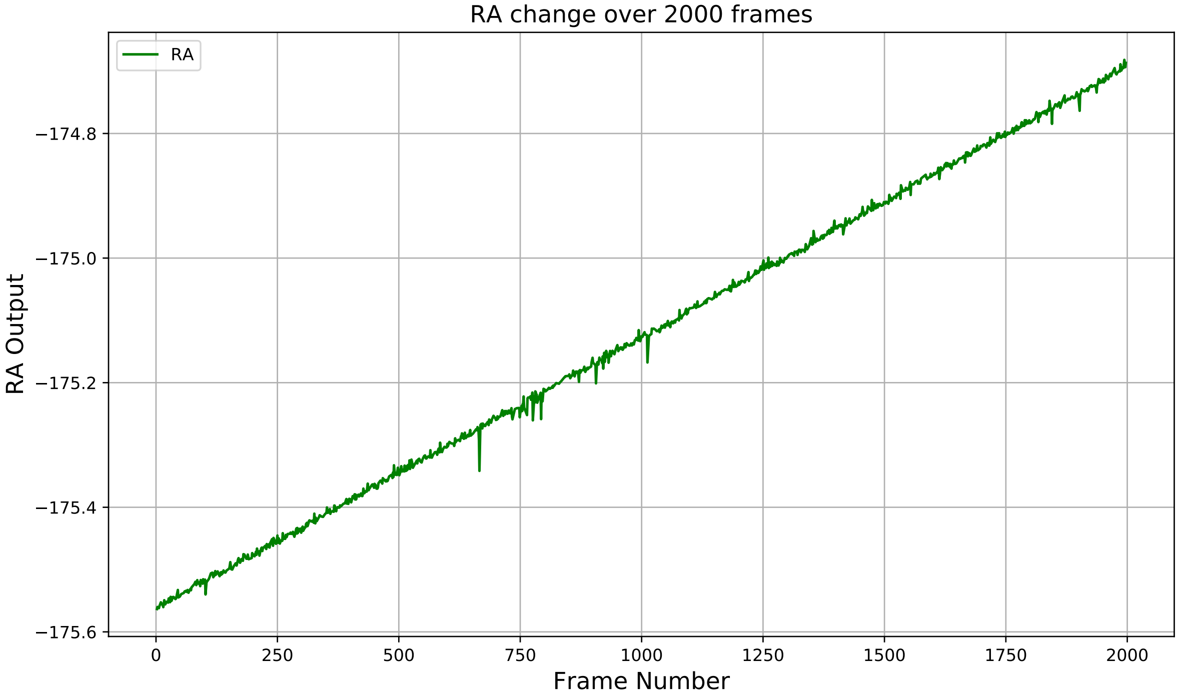


Figure 5.12: Right-Ascension output of the star tracker plotted against frame number for 2000 frames.

$$\omega_{earth} = \frac{360 \text{ degrees}}{23 \text{ hours, } 56 \text{ minutes}} \tag{5.13}$$

$$= \frac{360 \text{ degrees}}{86160 \text{ seconds}} \tag{5.14}$$

$$= 0.004178 \text{ degrees}/s \tag{5.15}$$

This result gives a good indication that the estimated attitude is accurate. It is not possible to determine the exact accuracy using this type of setup since the real attitude of the boresight is not known. A similar plot is made for the DEC output. We should expect no change in DEC. The plot is shown in figure 5.13. One or two jumps are seen due to centroid and matching errors, but the results are mostly consistent with a mean of -58.56 degrees and a standard deviation of 0.0052 degrees. The middle of the Crux constellation is found at 186 (-174) degrees RA and -60 degrees DEC. The result in figure 5.12 and 5.13 shows that the star tracker correctly determines the boresight coordinates

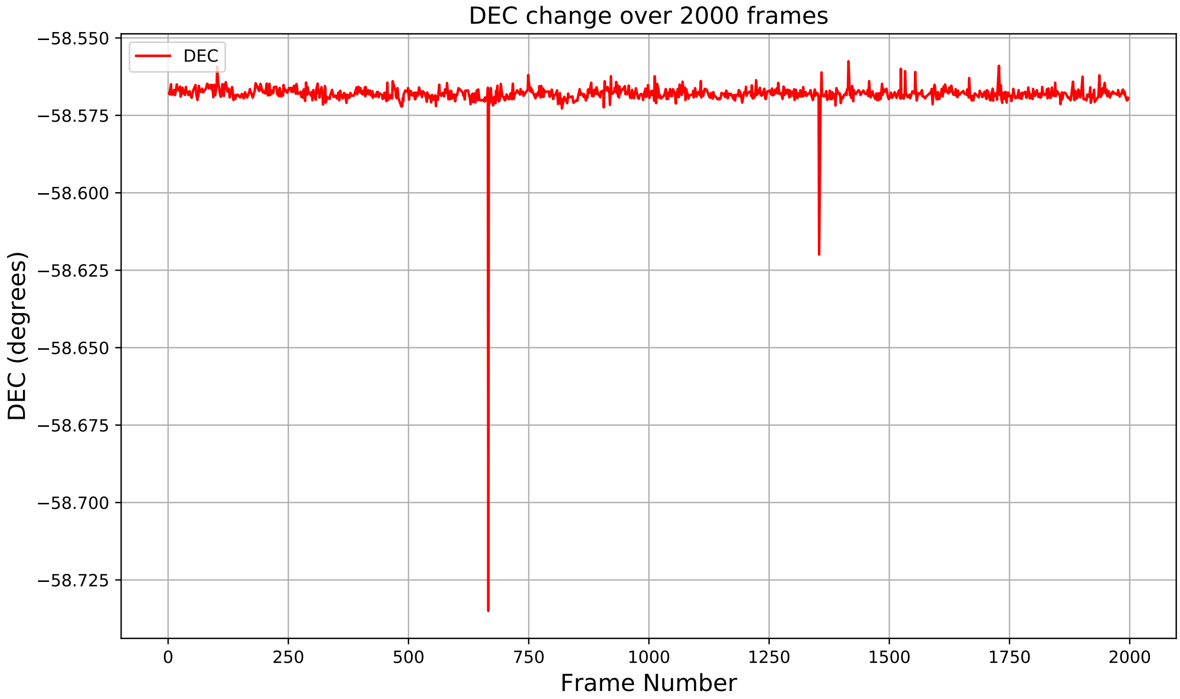


Figure 5.13: Declination output of the star tracker plotted against frame number for 2000 frames.

since the output is a point close to the centre of the constellation. (no attempt was made to point the bore-sight at an exact location, only that the Crux constellation was visible at all times).

In most of the frames, 4 stars were detected. Figure 5.14 displays the occurrence count (frequency) for different number of stars per frame. Not many stars are detected at such a low exposure time. It should be mentioned that in some cases, two stars were detected in close proximity, mostly due to the Bayer pattern forming gaps in regions. The preview of the captured Crux is shown in figure 5.15. Brightness and contrast adjustments are made to the image for display purposes. The noise level is high and not many stars are visible at such a short exposure.

Centroid Detection Detail

The positions of the centroids change over time as the earth rotate, but for this specific optical setup and at 10 Hz, the change is expected to be less than 0.05 pixels per frame. To interpret the centroid detection accuracy, the change in radial pixel distance between a star in two consecutive frames are measured (given that the star is detected). Technically speaking, these distances should only change by less than 0.05 pixels; therefore, it can be interpreted as a centroid detection error. Figure 5.16 display histograms showing the change in centroid detection position in two consecutive frames for Crux stars. Detail on each of these stars are shown in table 5.4. The average change for bright stars is around 0.2 pixels. Delta crux was less bright than the others, and it is clear that it caused more errors and was less likely to be detected.

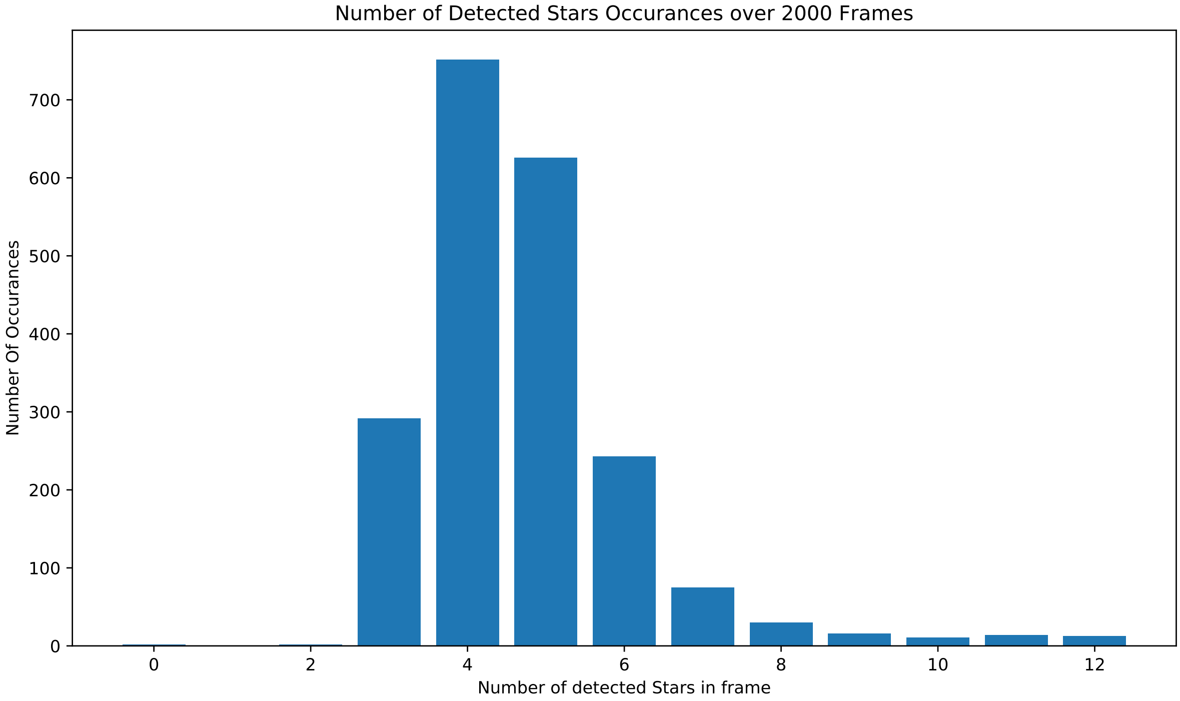


Figure 5.14: Occurrences of number of stars detected per frame.

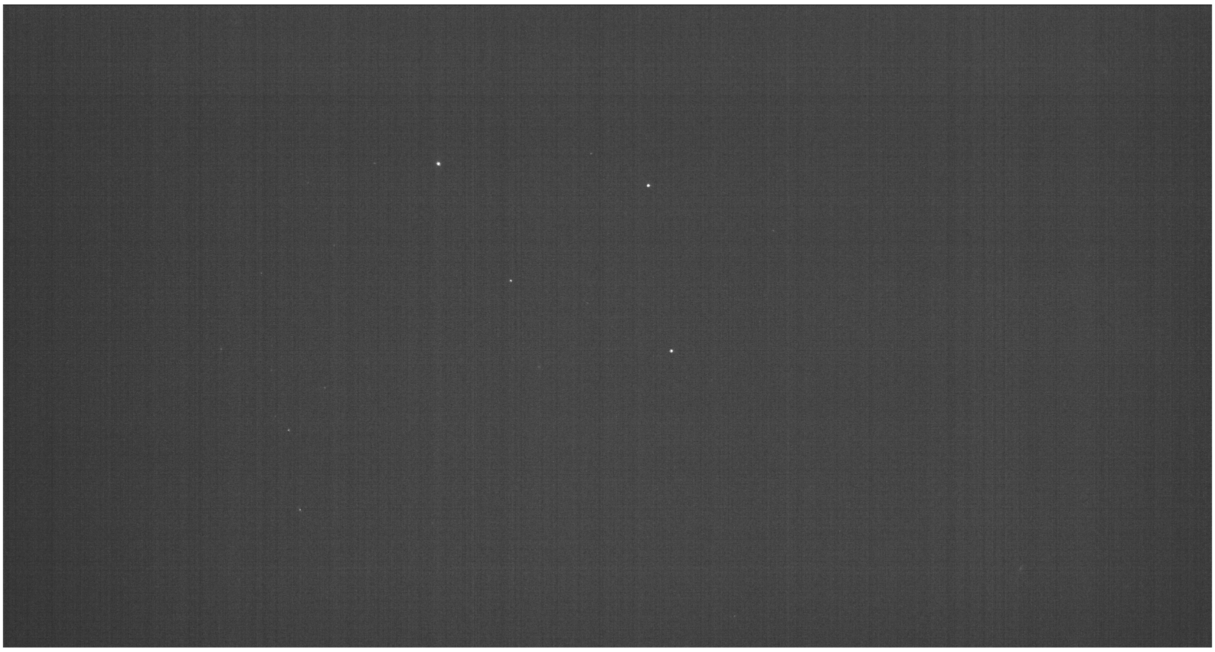


Figure 5.15: Sample image of Crux where the Gecko was pointed. For this preview, levels and contrast are adjusted to make the stars more visible.

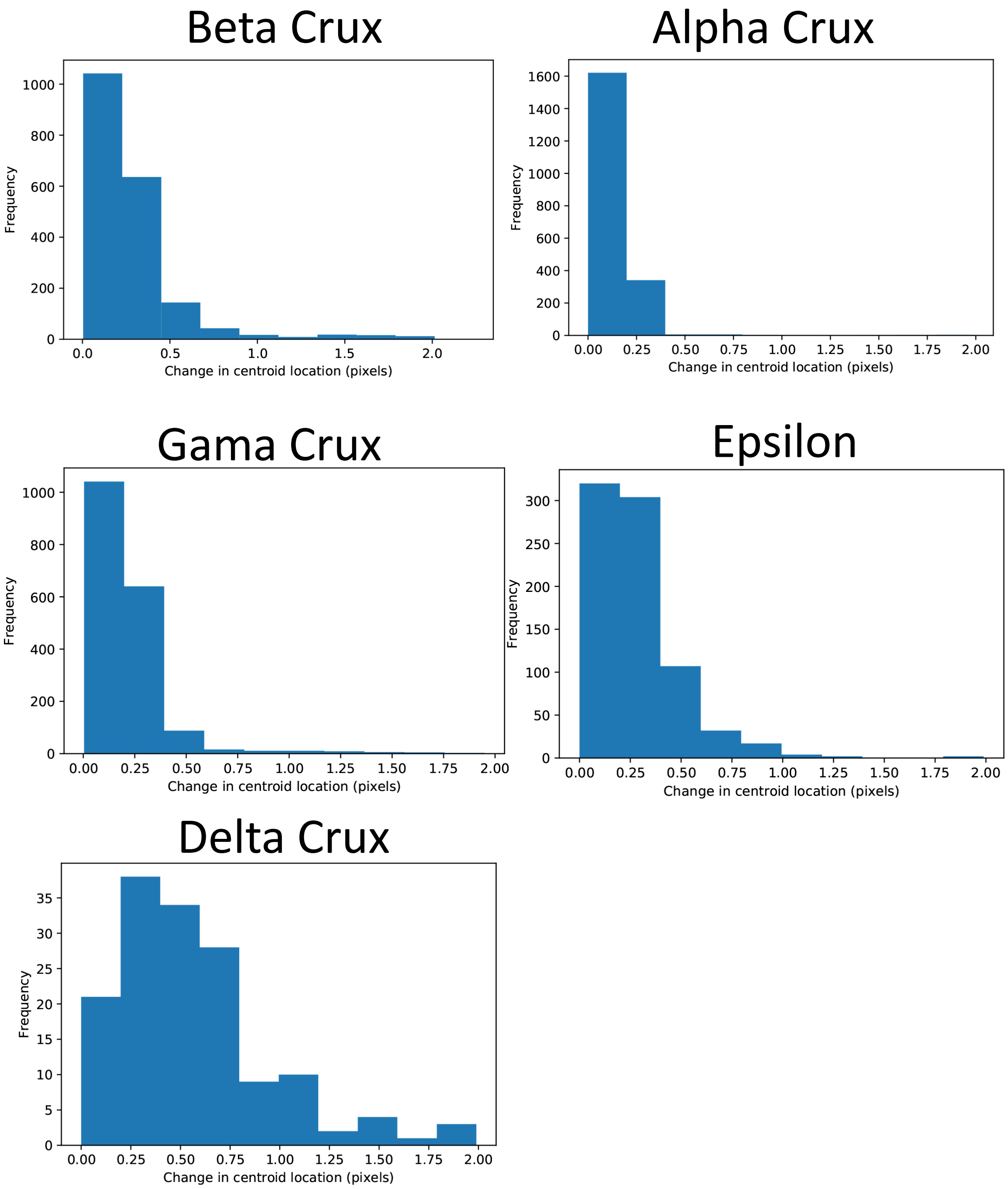


Figure 5.16: Histograms of effective centroid detection accuracy measured by the change in radial distance of a star between two consecutive frames. Notice that the change is mostly in the 0.2-pixel range for bright stars.

Star Name	**Alpha Crux**	**Beta Crux**	**Gamma Crux**	**Epsilon**	**Delta Crux**
Number of Times Detected	1995	1959	1912	1247	540
Detection Hit Ratio	99.75%	97.95%	95.60%	62.35%	27.00%
False Identification %	34.29%	5.72%	18.72%	13.71%	57.78%
Average Change in Centroid Location (pixels)	0.145	0.256	0.218	0.286	0.56
Sub-Pixel Centroid Detection	99.3985%	96.7841%	98.2218%	99.3585%	96.2963%

Table 5.4: Centroid detection details for Crux constellation.

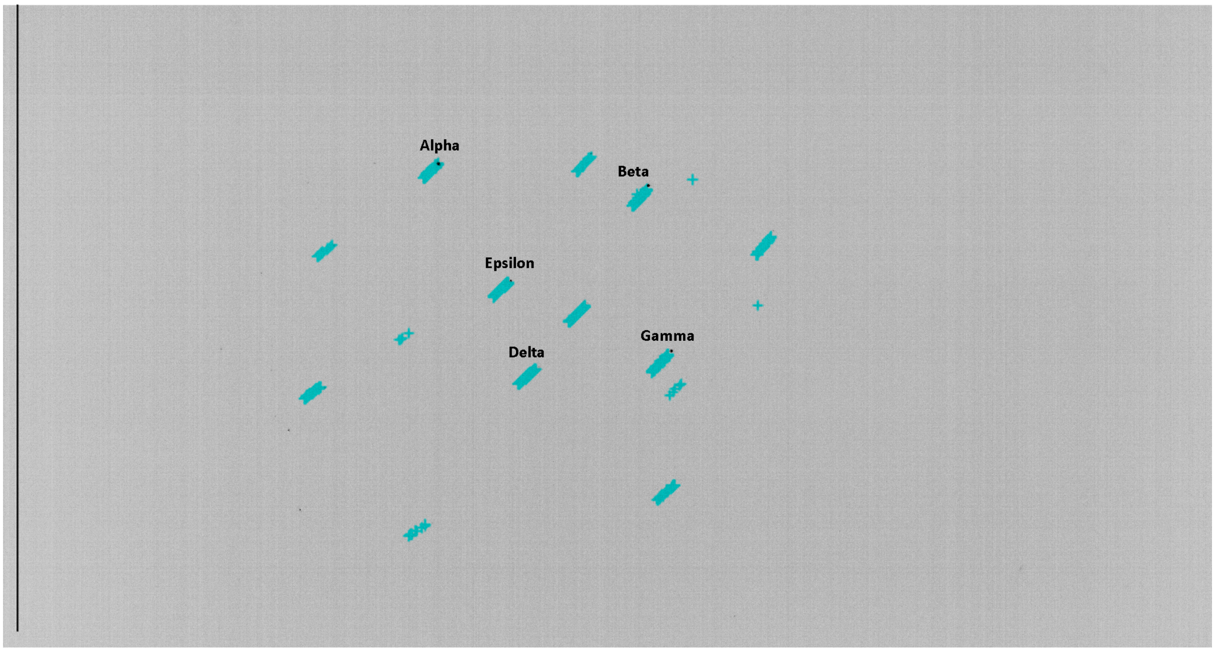


Figure 5.17: All of the detected centroids for 2000 frames overlay on the image.

Results Conclusion

Overall, sub-pixel centroid detection worked consistently, but a high percentage of false matches were found, which lead to attitude estimation errors. The inadequacies of the imager for star photos at short exposure times caused errors in centroid detection as discussed. These errors caused inaccurate angular separation measurements between star pairs that resulted in the voting algorithm to fail. Figure 5.17 shows all of the detected centroids for 2000 frames, overlay on the image.

Out of 2000 frames, attitude detection failed 31% of the time. Each of these failures are caused by false matches. Out of all of the failed frames, 44% of them had only 3 or less detected stars. It is clear that more work should be put into the matching algorithm to better handle frames with a small number of stars and high noise.

5.8 Timing Profile

The sensor hosted on the Gecko is set up to start image exposure of the next frame during read-out of the previous one [4]. This interleaved sequence is illustrated in figure 5.18 below.

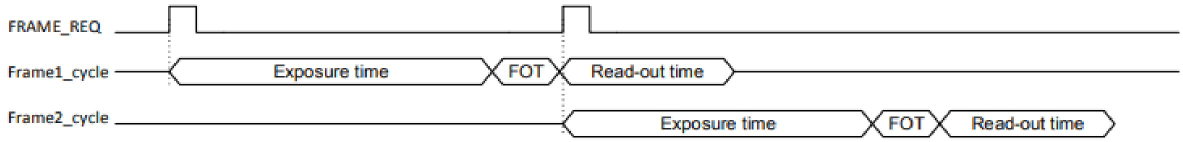


Figure 5.18: Exposure and Read-out timing of the sensor. Image taken from datasheet [4].

Although the exact read-out time of the Gecko is not given, the 200 MHz dual-channel output results in roughly 40 ms for a full resolution frame. As this is shorter than the chosen exposure time, it does not directly affect the update rate given the interleaved exposure of the sensor (figure 5.18). The validation round of voting and QUEST algorithm is profiled independently using an image with 12 stars. The timing result on the Zynq PS was 1 ms for counting and validation. QUEST was also tested and resulted in an almost insignificant time of 80μs. Figure 5.19 shows the final interleaved timing sequence of the star tracker.

It becomes clear from the diagram that exposure time is the most significant determining factor of the update rate for this star tracker. The processing speed of the real-time detection algorithm on FPGA is limited to the maximum speed that the logic configuration allows. For the final design, it is found that anything faster than 200 MHz caused setup and hold time violations and lead to unexpected errors. Thus technically, the algorithm can handle any parallel 8-bit image data at a 200 MHz rate. For any arbitrary 2 MB image, the fastest read-out time that the FPGA algorithm can handle would be 10 ms when using this baseline timing constraint. Should a high-performance sensor with a good SNR, capable of capturing bright and usable star photos with an exposure time close to 10 ms and interleaved read-out at high speed be used, one can see how the update rate can increase significantly for this optimised star tracker design.

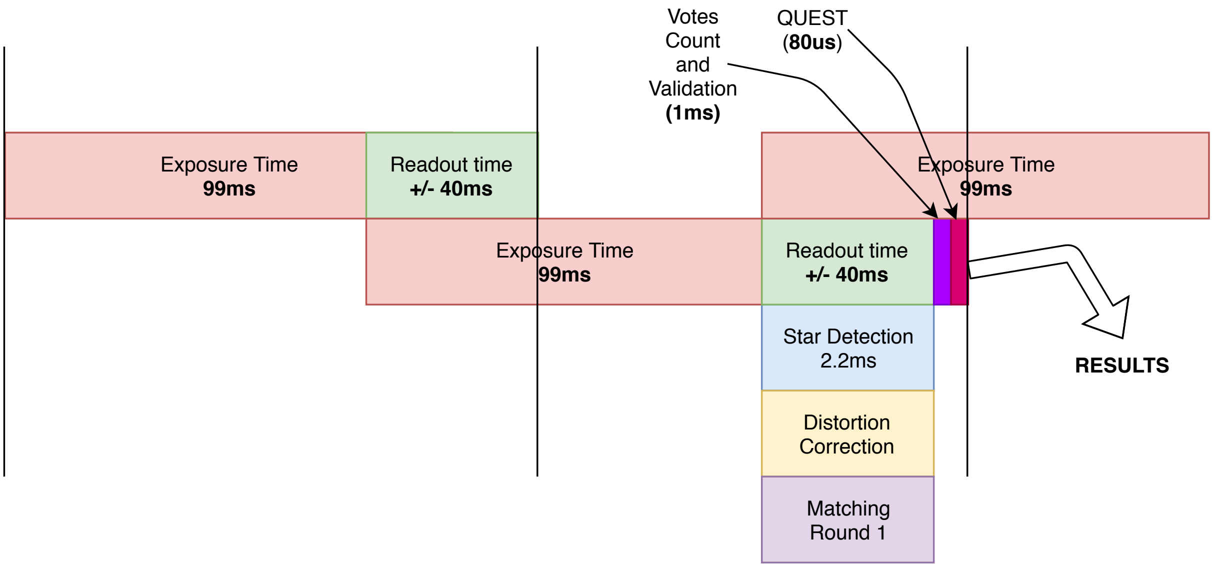


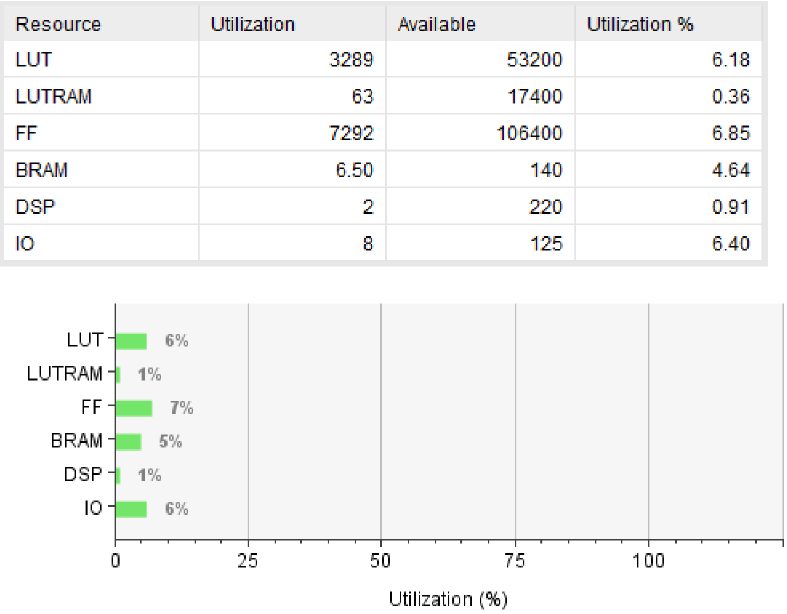
Figure 5.19: Interleaved timing sequence of the star tracker system.

5.8.1 Timing compared to conventional Centroid Detection

When this timing profile is compared to standard centroiding algorithms done on a processor, it becomes clear that the solution provides a significant speed advantage. Knutson [17] designed and tested a fast star tracker centroiding algorithm implemented on an ARM processor by utilising a tracking mode and windowed centroid extraction. For a full-frame, LIS solution, the algorithm took 211 milliseconds. After the initial attitude is acquired, a tracking algorithm predicts new locations of the centroids and a centre-of-mass algorithm is performed on small windows around these predicted locations. Even with this optimised technique, for a windowed solution where only five stars are tracked, the algorithm still takes 80 ms to execute, along with a 2 ms delay from the tracking algorithm. It is also important to keep in mind that his centroid extraction algorithm can only start once the entire image is saved to memory and matching can also only start once centroid detection is done. Even using a sensor with interleaved exposure and read-out, a 10 Hz update rate in tracking mode would require a very high-speed read-out and is unlikely to achieve consistently.

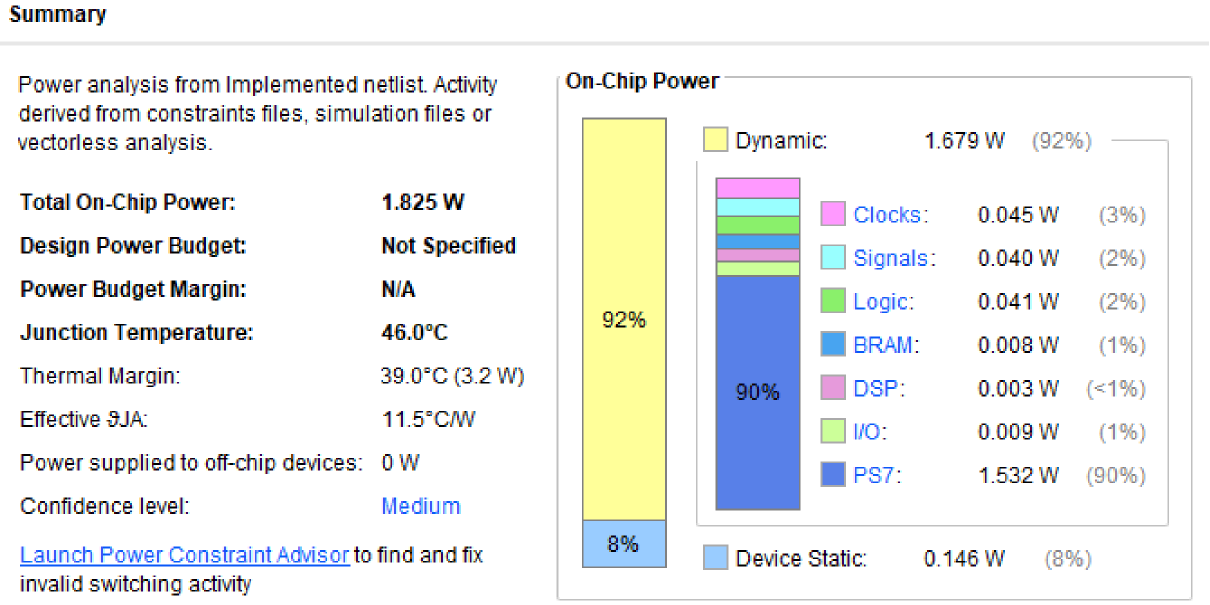
5.9 FPGA Resource Usage

The Xilinx Vivado tool reports resource utilisation of the FPGA after implementation. Figure 5.20 shows a screenshot of these results. It is clear that this design is very resource-light and can, therefore, be implemented on small FPGAs. Inspecting the power usage report, it is clear that the processor system (PS7) consumes 90% of the power whereas the FPGA design does not consume too much power. This confirms that power usage of a star tracker is not compromised if star detection is implemented on an FPGA. The report is seen in figure 5.21.



Resource	Utilization	Available	Utilization %
LUT	3289	53200	6.18
LUTRAM	63	17400	0.36
FF	7292	106400	6.85
BRAM	6.50	140	4.64
DSP	2	220	0.91
IO	8	125	6.40

Figure 5.20: FPGA resource usage report.



Summary

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**1.825 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**46.0°C**
Thermal Margin:	39.0°C (3.2 W)
Effective ϑJA:	11.5°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Medium

Launch Power Constraint Advisor to find and fix invalid switching activity

On-Chip Power

Dynamic:	1.679 W	(92%)
Clocks:	0.045 W	(3%)
Signals:	0.040 W	(2%)
Logic:	0.041 W	(2%)
BRAM:	0.008 W	(1%)
DSP:	0.003 W	(<1%)
I/O:	0.009 W	(1%)
PS7:	1.532 W	(90%)
Device Static:	0.146 W	(8%)

Figure 5.21: SoC Power usage analysis.

Chapter 6

Conclusion

Complex scientific requirements demanded by modern spacecraft requires high-precision attitude determination and control. As advancing technology such as CMG's becomes feasible in small satellites, star trackers must run at high update rates for a fast control loop while maintaining accuracy in the arcsecond range.

The work presented in this thesis demonstrates a fast star tracker design where centroid extraction is implemented in hardware on an FPGA device. The detection algorithm is designed, implemented and tested using various simulated and real images. An optimisation technique for the matching algorithm is also discussed, implemented and tested on the processing system of the SoC device. Distortion correction and QUEST was implemented to demonstrate an end-to-end system using the Gecko as the imager front-end. A high-speed LVDS interface is designed on the FPGA to connect to the imager. Night-sky simulation software is written that were used to produce test images. An emulated hardware sensor is created to simulate the Gecko transmission protocol and stream off-line images onto the FPGA.

Results from chapter 3 show that centroid extraction on an FPGA is successfully realised and behaves as expected. The matching algorithm is optimised and takes around 1 ms to execute during validation. This short time confirms that it is not necessary to implement matching on the FPGA as it takes significantly less time than standard centroid detection and therefore has a negligible effect on the update rate.

Section 5.7 exhibits a full working system, and it is clear that sensor exposure time almost completely determines the update rate of the star tracker when centroid detection is done in parallel to read-out. The gain of the Gecko had to be increased to meet the 10 Hz update rate objective, which caused a high noise level on the image and effectively decreased the accuracy of the star tracker. The experiment, however, still shows the high-speed capability.

The fast FPGA centroiding algorithm presented in this work shows a significant advantage compared to the conventional approach on a processor. There are very few limits on the number of stars that can be detected and used to determine attitude. It performs a full-frame LIS solution on every exposure, making it autonomous at all times. As shown in the timing diagram (figure 5.19), the centroiding algorithm now does not limit the update rate of the star tracker as before. Should even faster update rates, as high as

80-100 Hz, be required in future spacecraft ACDS's, real-time centroid extraction on an FPGA used alongside specialised high sensitivity image sensors can make this possible.

6.1 Improvements/Recommendations

6.1.1 High Sensitivity Sensor

A high sensitivity image sensor, capable of low light imaging is vital for a fast, accurate star tracker. Recent advances in image sensor technology, such as the STARVIS series of sensors by Sony, is back-illuminated CMOS pixel sensors which feature sensitivities of 1200mV or more per 1 μm^2 when imaging with a 706 cd/m^2 light source, at F8 aperture and 1/30s accumulation [8]. At short exposure times, these sensitive sensors can capture more stars with lower noise, resulting in higher accuracies. It is also possible to use an image intensifier [35] in the optical path: Yan et al. found centroid detection errors of less than 0.2 pixels at exposure times as short as 8 to 16 ms, using an image intensifier [34]. Such short exposure times would enable very high-speed star trackers and make them capable of handling fast slew rates due to reduced star smearing. Research in the field of image sensors is therefore as important as algorithm optimisations for the future of star trackers.

6.1.2 Dynamic Threshold

The current design uses a fixed threshold value per frame. In some situations, it may be desirable to have this threshold value adjust dynamically depending on the noise floor level of the image. A simple recommendation is to create a small sub-circuit that calculates the average pixel intensity per line (or multiple lines). The first line would use a default threshold, but after this, the value can be updated each line dynamically by using the average pixel intensity plus some offset threshold.

6.1.3 Radiation Hardening

Radiation effects can cause logic errors in digital circuits which corrupts the states of memory and can cause problems in the system. The detail on radiation in space goes beyond the scope of this thesis, but methods to mitigate these problems should be mentioned. In FPGAs, Triple Modular Redundancy (TMR) is typically used [15]. The TMR technique uses three identical logic blocks instead of one for each task and performs the same task in parallel. The outputs are then compared through majority voters to check for possible errors. This technique increases the resource usage of the system and should be kept in mind when implementing the design on limited sized FPGAs.

6.1.4 Abstracted Single Line Region Growing

The margin counter in the region growing part of the algorithm causes some problems when high noise is present or irregular large stars are found. An improvement (not yet implemented) can be made with a small change in the design. Instead of passing single pixels from the Threshold and Margin counter block, accumulated lines containing the X-Sum, accumulated intensity, start-X and end-X and pixel-count of direct pixel neighbours can be passed to the region manager block. If only one pixel is passed on and merges with an existing region, but the pixel has no direct neighbours, then it should be discarded. This idea or a similar procedure can hopefully improve detection accuracy and avoid multiple star detections at large regions.

6.1.5 FPGA Design Considerations

Not many engineers regularly work with HDL design on FPGA's. There are major differences between software for embedded systems compared to VHDL designs. Software engineers tend to modularize and often add complexity for the sake of cleaner design, re-usability and collaboration. In FPGAs, it is encouraged to keep the hardware flow as simple as possible to avoid timing issues and hardware debugging. The design presented in this thesis still contains a few elements that can be simplified and improved for hardware.

Appendix A

Detailed Hardware Setup

The prototyping setup is shown in figure A.1 below. A breakout carrier card is needed to access the PL pins on the Microzed. LVDS data and SPI lines are wired from the Gecko using an HDMI cable. Table A.1 shows the pin-out diagram of the HDMI cable coming from the Gecko. LVDS signals are directly connected to the PL. It is important to connect the LVDS clock to clock-capable (MRCC or SRCC - Multi/Single Region Clock Capable) pins. Figure A.2 shows the pin configuration in the Vivado floorplanning setup. Notice that all of the signals are connected to SRCC or MRCC pins and LVDS_25 (2.5 Volts LVDS) is selected as the IO standard. A selectIO interface wizard (IP core by Xilinx) is used as an input clock and data buffers on the PL (figure A.3). This IP core also drives clock circuitry triggered by the input clock, creating reliable clock and data signals. The SPI lines are routed through 100 Ohm series resistors and connected to the PS through the PMOD connector as shown in figure A.1.

Function	**HDMI pin**
LVDS_Clock_P	10
LVDS_Clock_N	12
LVDS_Data 0_P	7
LVDS_Data 0_N	9
LVDS_Data 1_P	4
LVDS_Data 1_N	6
LVDS_Control_P	1
LVDS_Control_N	3
None	14
None	19
SPI_CLK	13
SPI_nCS	15
SPI_MOSI	16
SPI_MISO	18
Ground	2, 5, 8, 11, 17

Table A.1: HDMI Cable pin diagram from Gecko.

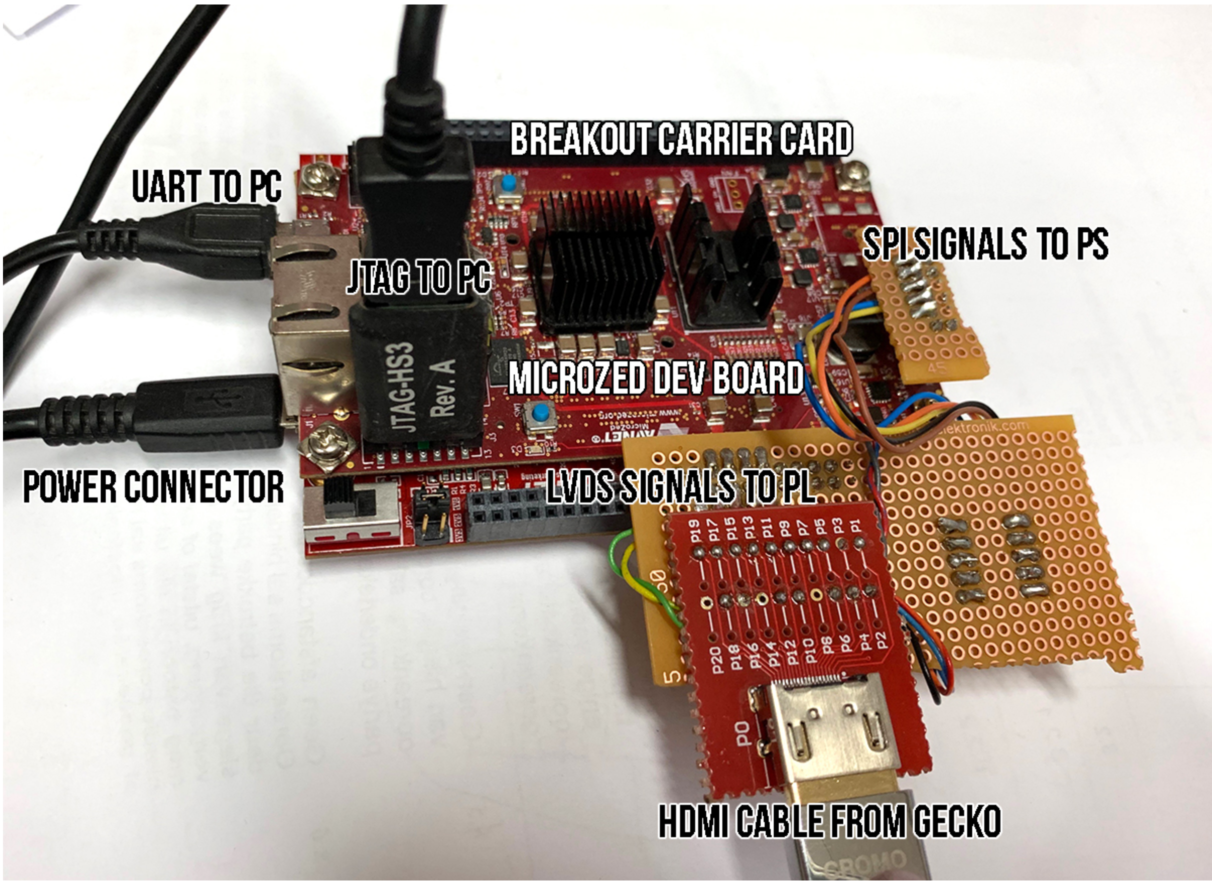
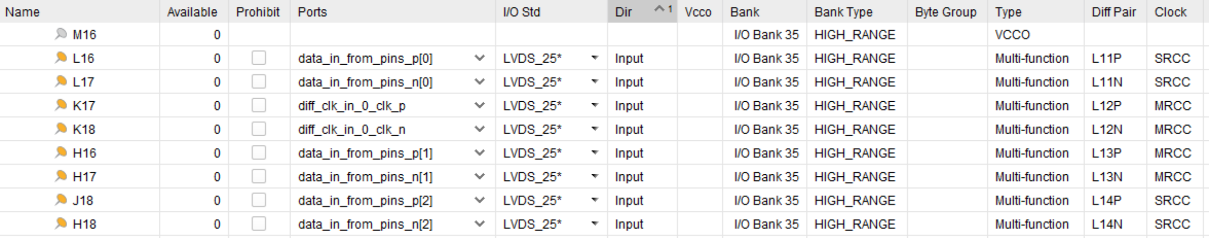


Figure A.1: Microzed development board on top of the breakout carrier card. The image shows all of the connections used during development and debug.



Name	Available	Prohibit	Ports	I/O Std	Dir	Vcco	Bank	Bank Type	Byte Group	Type	Diff Pair	Clock
M16	0						I/O Bank 35	HIGH_RANGE		VCCO		
L16	0	□	data_in_from_pins_p[0]	LVDS_25*	Input		I/O Bank 35	HIGH_RANGE		Multi-function	L11P	SRCC
L17	0	□	data_in_from_pins_n[0]	LVDS_25*	Input		I/O Bank 35	HIGH_RANGE		Multi-function	L11N	SRCC
K17	0	□	diff_clk_in_0_clk_p	LVDS_25*	Input		I/O Bank 35	HIGH_RANGE		Multi-function	L12P	MRCC
K18	0	□	diff_clk_in_0_clk_n	LVDS_25*	Input		I/O Bank 35	HIGH_RANGE		Multi-function	L12N	MRCC
H16	0	□	data_in_from_pins_p[1]	LVDS_25*	Input		I/O Bank 35	HIGH_RANGE		Multi-function	L13P	MRCC
H17	0	□	data_in_from_pins_n[1]	LVDS_25*	Input		I/O Bank 35	HIGH_RANGE		Multi-function	L13N	MRCC
J18	0	□	data_in_from_pins_p[2]	LVDS_25*	Input		I/O Bank 35	HIGH_RANGE		Multi-function	L14P	SRCC
H18	0	□	data_in_from_pins_n[2]	LVDS_25*	Input		I/O Bank 35	HIGH_RANGE		Multi-function	L14N	SRCC

Figure A.2: LVDS pin configuration in Vivado.

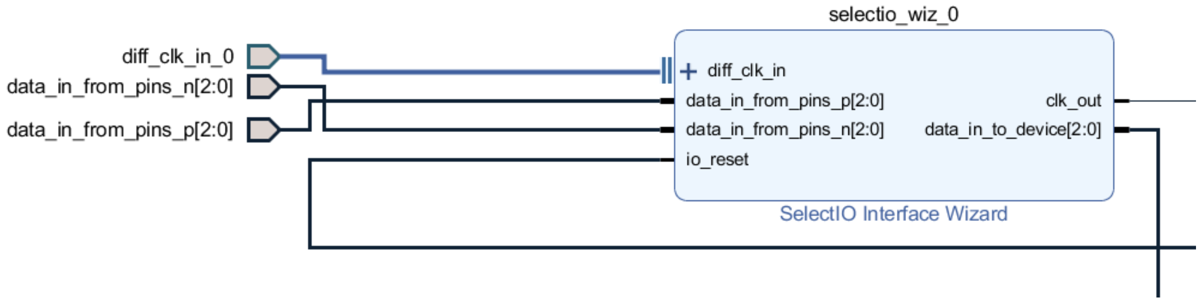


Figure A.3: SelectIO IP core. Input clock and data buffer.

Bibliography

[1] Sao - smithsonian astrophysical observatory star catalog. https://heasarc.gsfc.nasa.gov/W3Browse/star-catalog/sao.html, 1990. Accessed: 2019-06-08.

[2] The different frames and the keplerian elements. https://adcsforbeginners.wordpress.com/tag/earth-centred-inertial-frame, 2008. Accessed: 2019-06-08.

[3] John Leif Joergensen Allan Read Eisenman, Carl Christian Liebe. New generation of autonomous star trackers, 1997.

[4] AMS Semiconductors. *2.2 Megapixel machine vision CMOS image sensor.*, 2015. Rev. 3.8.

[5] Benjamin B. Spratling and D Mortari. A survey on star identification algorithms. *Algorithms*, 2, 03 2009.

[6] C.D.Hall. *Spacecraft Attitude Dynamics and Control*, chapter 4. 2003.

[7] Robert Collins. Lecture notes in camera projection, cse486, lecture 12. http://www.cse.psu.edu/ rtc12/CSE486/lecture12.pdf, 2008.

[8] Sony Semiconductor Solutions Corporation. Sony imx290llr monochrome cmos image sensor, 2019.

[9] James Diebel. Representing attitude: Euler angles, unit quaternions, and rotation vectors. 10 2006.

[10] Alexander Olaf Erlank. DEVELOPMENT OF CUBESTAR : A CUBESAT-COMPATIBLE STAR TRACKER. Master's thesis, Stellenbosch University, South Africa, 2013.

[11] J. Farrell, J. Stuelpnagel, R. Wessner, J. Velman, and J. Brook. A least squares estimate of satellite attitude (grace wahba). *SIAM Review*, 8(3):384–386, 1966.

[12] Thomas Grob. Implementation of a fpga-based interface to a high speed image sensor. 01 2010.

[13] Qian Hua-Ming, Li Hao, and Wang Hai-Yong. Design and verification of star-map simulation software based on ccd star tracker. pages 383–387, 06 2015.

[14] M. J. Jacobs. A low cost, High precision star sensor. Master's thesis, University of Stellenbosch, South Africa, 1995.

[15] F. L. Kastensmidt, L. Sterpone, L. Carro, and M. S. Reorda. On the optimal design of triple modular redundancy logic for sram-based fpgas. In *Design, Automation and Test in Europe*, pages 1290–1295 Vol. 2, March 2005.

[16] L. Kazemi, J. Enright, and T. Dzamba. Improving star tracker centroiding performance in dynamic imaging conditions. In *2015 IEEE Aerospace Conference*, pages 1–8, March 2015.

[17] Matthew W. Knutson. Fast star tracker centroid algorithm for high performance cubesat with air bearing validation. 2012.

[18] M. Kolomenkin, S. Pollak, I. Shimshoni, and M. Lindenbaum. Geometric voting algorithm for star trackers. *IEEE Transactions on Aerospace and Electronic Systems*, 44(2):441–456, April 2008.

[19] Vaios Lappas, Willem Steyn, and Craig Underwood. Practical results on the development of a control moment gyro based attitude control system for agile small satellites. 2002.

[20] C.C. Liebe. Accuracy performance of star trackers-a tutorial. *Aerospace and Electronic Systems, IEEE Transactions on*, 38:587 – 599, 05 2002.

[21] Marcus Lindh. Development and Implementation of Star Tracker Electronics. Master's thesis, KTH Royal Institute of Technology, Stockholm, Sweden, 2014.

[22] Yanfei Liu and Carlos Pomalaza-raez. *Self-Landmarking for Robotics Applications*. 08 2011.

[23] Landis Markley and D Mortari. Quaternion attitude estimation using vector observations. *Journal of the Astronautical Sciences*, 48, 04 2000.

[24] C. R. McBryde and E. G. Lightsey. A star tracker design for cubesats. In *2012 IEEE Aerospace Conference*, pages 1–14, March 2012.

[25] W.D. McClain and D.A. Vallado. *Fundamentals of Astrodynamics and Applications*. Space Technology Library. Springer Netherlands, 2001.

[26] D. Michaels and J. Speed. Ball aerospace star tracker achieves high tracking accuracy for a moving star field. In *2005 IEEE Aerospace Conference*, pages 1–7, March 2005.

[27] Curtis Padgett, Kenneth Kreutz-Delgado, and Suraphol Udomkesmalee. Evaluation of star identification techniques. *Journal of Guidance Control and Dynamics - J GUID CONTROL DYNAM*, 20:259–267, 03 1997.

[28] A. Read Eisenman and C. C. Liebe. The advancing state-of-the-art in second generation star trackers. In *1998 IEEE Aerospace Conference Proceedings (Cat. No.98TH8339)*, volume 1, pages 111–118 vol.1, March 1998.

[29] Carlos Ricolfe-Viala and Antonio-Jose Sanchez-Salmeron. Lens distortion models evaluation. *Applied optics*, 49:5914–28, 10 2010.

[30] M. D. SHUSTER and S. D. OH. Three-axis attitude determination from vector observations. *Journal of Guidance and Control*, 4(1):70–77, 1981.

[31] Casey Grant Smith. Development and implementation of star tracker based attitude determination. Master's thesis, Missouri University of Science and Technology, United States of America, 2017.

[32] Willem Steyn, M J Jacobs, and Pierre Oosthuizen. A high performance star sensor system for full attitude determination on a microsatellite. 04 2004.

[33] Matthew W. (Matthew Walter) Knutson. Fast star tracker centroid algorithm for high performance cubesat with air bearing validation. 03 2014.

[34] Jinyun Yan, Jie Jiang, and Guangjun Zhang. Modeling of intensified high dynamic star tracker. *Optics Express*, 25:927, 01 2017.

[35] Shuo Zhang, Fei Xing, Ting Sun, Zheng You, and Minsong Wei. Novel approach to improve the attitude update rate of a star tracker. *Opt. Express*, 26(5):5164–5181, Mar 2018.

[36] Fuqiang Zhou, Jingxin Zhao, Tao Ye, and Lipeng Chen. Fast star centroid extraction algorithm with sub-pixel accuracy based on fpga. *Journal of Real-Time Image Processing*, 12(3):613–622, Oct 2016.