UNIVERSITY OF TARTU
Institute of Technology
Robotics and Computer Engineering Curriculum

Roberts Oskars Komarovskis

Development and Implementation of ESTCube-2 Star Tracker FPGA Design

Master's Thesis (30 ECTS)

Supervisor(s): Kristo Allaje, MSc
Tõnis Eenmäe, MSc

Tartu 2024

Abstract:

Development and implementation of ESTCube-2 star tracker FPGA design Attitude determination and control play a critical role in a spacecraft, where one solution for attitude determination can be star tracking. Star tracker systems can provide high-accuracy attitude information based on star identification, and such a system is also a part of the ESTCube-2 nanosatellite developed by the Estonian Student Satellite Foundation. One of the primary components in the ESTCube-2 star tracker is a field-programmable gate array (FPGA) responsible for processing the captured images. This thesis aims to develop and implement FPGA system design and necessary algorithms for star tracking and assess the performance of the obtained design. Different FPGA design components were made to interface with external devices and perform different functions related to star tracking.

Keywords:

ESTCube-2, Star Tracker, Attitude Determination, FPGA

CERCS:

T170 Electronics; T320 Space technology

ESTCube-2 tähejälgija FPGA disaini arendus Enamasti on satelliidi asendi määramine ja kontroll kosmilises ruumis missiooni seisukohast kriitiline. Asendi määramiseks kasutatakse laialdaselt tähtede teadaolevaid koordinaate taevasfääril ning juba tuvastatud tähtede abil on võimalik asendikontrolli algoritmidel satelliiti tähtede suhtes nõutud asendis hoida. Tähejälgimissüsteemid suudavad võrreldes enamike muude asendi määramise meetoditega anda väga suure täpsusega asendiinfot ning selline süsteem on loodud ka Eesti Tudengisatelliidi Sihtasutuse poolt välja töötatud nanosatelliidi ESTCube-2 jaoks. ESTCube-2 tähejälgimissüsteemi üks peamisi komponente on programmeeritav loogikamassiiv (ik. field-programmable gate array) (FPGA), mille ülesandeks on tähejälgimiskaamera sensoriga jäädvustatud piltide töötlemine. Käesoleva lõputöö eesmärk on välja töötada ja implementeerida FPGA disain, tähtede jälgimiseks vajalikud algoritmid ning hinnata saadud disaini toimivust. Töö käigus loodi erinevad FPGA disainikomponendid, mille abil liidestutakse välisseadmetega ja täidetakse erinevaid tähtede tuvastamisega seotud funktsioone.

Märksõnad:

ESTCube-2, tähejälgija, kosmoseaparaadi asendikontroll, FPGA

CERCS:

T170 Elektroonika; T320 Kosmosetehnoloogia

Contents

List of Figures

List of Tables

Acronyms and Abbreviations

ADU	Analog-to-digital unit
CCD	Charged-coupled device
CMOS	Complementary metal oxide semiconductor
COTS	Commercial off-the-shelf
DEMUX	Demultiplexer
FPGA	Field-programmable gate array
FIFO	First-in, first-out
FP	Floating-point
FV	Frame valid
ID	Identification
IMU	Inertial measurement unit
IP	Intellectual property
LSB	Least significant bit
LV	Line valid
MCU	Microcontroller
MSB	Most significant bit
MUX	Multiplexer
NASA	National Aeronautics and Space Administration
PCB	Printed circuit board
PLL	Phase-locked loop
RAM	Random access memory
SDRAM	Synchronous dynamic random-access memory
SPI	Serial peripheral interface
UART	Universal asynchronous receive transmit
USB	Universal serial bus
VHDL	Very high-speed integrated circuit hardware description language

1 Introduction

In recent years, the popularity of smaller spacecraft has increased dramatically, and the source for this increase in popularity may be related to numerous reasons. For example, a small spacecraft's lead time and expenses can be considerably smaller.[2] Lower development costs allow smaller research groups like universities to develop their own spacecraft. Teams developing these types of spacecraft are more permissive to using commercial-of-the-shelf (COTS) components. [3] These devices may have lower reliability in harsh space environments but allow much higher ambitions as the most recent technology has been placed at arm's length.

The Estonian Student Satellite Foundation is a small research group that successfully launched the ESTCube-1 nanosatellite to space in 2013. Ten years later, in October of 2023, ESTCube-2 was launched, but unfortunately, there was some malfunction during the launch, and the satellite was regrettably lost. Regardless of this tragic event, the thesis was continued because the Estonian Student Satellite Foundation continues to plan the following satellite missions, including the possibility of manufacturing another ESTCube-2 satellite, where the results of this thesis might become relevant and valuable. ESTCube-2 was a $10 \times 10 \times 30$ centimeter nanosatellite that follows the CubeSat standard, where a unit corresponds to specific dimensions of $10 \times 10 \times 10$ centimeters. [4] Numerous systems on the satellite are responsible for different tasks required for maintaining the satellite operation, like sending data to the ground stations, determining and controlling the attitude, and others. Attitude determination/control is one of the most critical tasks as it allows one to maintain or change the satellite's attitude depending on the requirements present at that specific moment.

One of the ways ESTCube-2 determines its attitude is with a star tracker system. Star trackers determine the satellite's attitude by taking an image of the stars. Considering that the star positions in the sky are static, and if the star tracker can effectively determine which stars it is observing, it can determine the spacecraft's attitude relative to the stars. However, this memory and computationally heavy task requires significant hardware and software effort. One of the central components of the ESTCube-2 star tracker system is the field-programmable gate array (FPGA), which is responsible for image data processing and manipulations. This thesis focuses on developing algorithms necessary for the final attitude acquisition and a backbone structure of the FPGA that allows it to interface with other devices present in the system and to test the developed algorithms. The system's hardware has already been finalized and will not be covered in detail. The main goals of this thesis are:

- Continue the implementation of the ESTCube-2 star tracker system based on previous ESTCube-2 member efforts
- Develop an efficient interface between the FPGA, other on-board devices, and the user
- Implement ESTCube-2 star tracker FPGA design that is capable of performing star tracking algorithms
- Test the implemented FPGA components

2 Star Tracking

This section describes the history, main principles, core algorithms, advantages, and disadvantages of star tracking systems in space, and it aims to lay a solid theoretical foundation for further practical development of the ESTCube-2 star tracker FPGA design.

2.1 History of Star Trackers

People used star tracking long before any spacecraft was ever made. The origins could be found with the sailors who used observable stars to navigate through the seas, where other visual clues are scarce. [5]

As space exploration reached new depths in the late sixties of the previous century, so did the star tracker systems and their usage. Star-tracking origins in space exploration could date back to the National Aeronautics and Space Administration (NASA) Gemini VII mission, where an on-board operator used a hand-held photometer to observe and track a star as it passes beyond the Earth's horizon. The data was recorded for post-flight analysis to update the existing atmospheric model for horizon-based measurement systems. [6] Gemini missions gave critical insight into real-time operation troubleshooting, experience in advanced space operations, and evaluation of existing navigation systems. [7] [8]

In NASA Apollo missions, attitude determination was an important task to overcome, considering the new and poorly studied lunar environment and the dire consequences of inaccurate measurements. Gyroscopes were commonly used to determine the attitude, but they tend to drift, and uncertainties may arise due to changing gravitational forces. Thus, they require periodic calibrating. Apollo missions used an optical system consisting of a telescope and a variation of a sextant to adjust the spacecraft's inertial measurement unit (IMU) or change the IMU reference point. The operator could insert a program into the computer to align the optics to a specific star for drift corrections. If IMU had drifted, there was a misalignment in the sextant, and the operator could manually adjust the error. Sextant could also be used to change the reference point of the IMU by aligning an image of one or two known stars with the Earth's or Moon's horizon and measuring the angle used as an input for the guidance computer. Though ground-based and inertial measurements were at the heart of the navigation system, observation and usage of celestial bodies still played a considerable role. [8, 9, 10]

The methods described in the previous paragraphs depend on the operator's input and would be considered the first generation of star trackers, where the star had to be manually locked and required some additional attitude instrument. Different factors directed the advancement of space technology in the second half of the 20th century; for example, the first creation of complementary metal-oxide semiconductor (CMOS) transistors and integrated circuits, which, following Moore's law, increased their possible density twice every two years or so. The advancement could also be related to increased space and military efforts related to the "Cold War".[11]

Before the first autonomous star tracker was developed, there were significant leaps in optics, material, sensor technology, and analog and digital electronics. [12] The first fully autonomous star tracker (at the time named an advanced stellar compass) was on board the Danish satellite Ørsted that was successfully launched into space in 1999[12]. This advanced stellar compass mainly consisted of a Sony CDX039AL charge-coupled-device (CCD) camera and an Intel 80486 processor. Compared to its predecessor, this system could do much more than output the coordinates of specific stars - it had a storage of a star catalog, and it was capable of fully acquiring the attitude of the spacecraft and outputting the information as quaternions or stellar coordinates. [13, 14]

Since the dusk of the 20th century, autonomous star trackers have become a standard practice in space missions. Generally, the core principles of these systems have remained the same, but technological advancements have continued, allowing for more efficient and compact systems.

2.2 Principles of Star Tracking

A typical star tracker system consists of a camera, a star catalog, and a processing unit, which can determine star coordinates in the acquired image and apply pattern matching between these coordinates and those embedded in the star catalog. This results in some form of attitude information relative to the satellite. Several parameters describe the operation of the star tracker in terms of usage in a spacecraft - accuracy, size, weight, power consumption, radiation resistance, lifetime, etc. In 2001, a typical star tracker weighed 1-7 kg, consumed 5-15 W, had an update rate of 0.5-10 Hz, and had accuracy in several arcseconds. [15]. Twenty years later, commercial solutions can have a 4-10 Hz update rate, generally 10-arcsecond accuracy, 2-3.2 W power consumption, a weight of 350-760 g, and a lifetime of over a decade. [16, 17, 18] Even if the performance has not significantly improved, there are some noticeable improvements regarding the star tracker system size, weight, and power consumption - highly favorable areas for smaller missions such as nanosatellites.

The driving factor in using a star tracker in a spacecraft attitude determination system is their high accuracy – a matter of arcseconds relative to other methods that provide accuracies in the range of arcminutes. [11] The angular resolution of an image sensor is directly related to the number of pixels of the camera. This relation is described with the equation 1, where ζ is the angular resolution of one pixel, θ is the half opening angle of the lens, and N is the number of pixels across the image sensor. [11]

$$\zeta = \frac{2\theta}{N} \tag{1}$$

When applying typical star tracker camera parameters to the equation, it becomes apparent that the angular resolution of a pixel is higher than the advertised star tracker accuracies. Probabilistic methods or the use of hyperacuity can help achieve these advertised sub-pixel accuracies in a star tracker. [11] In general, the accuracy of a star tracker is a function of field-of-view, the routine that battles the sampling theorem, image sensor resolution, and its other characteristics, such as susceptibility to noise.

Generally, there are two modes of operation for star tracking systems – ”lost in space” and ”tracking”. The system is lost in space when it has no prior attitude information and must obtain the initial attitude. This is the most computationally intensive state, as the full image must be digitized and processed, and it may take several seconds to obtain the initial attitude. When the attitude is obtained from the ”lost in space”, the system can switch to tracking mode, where different predictions can be made. This is more efficient because the system can predict the possible area where stars will be in the following image, which allows for avoiding digitizing and processing the whole image. [15, 19] Different algorithms and data manipulations are required to obtain the final outputs at different steps, as seen in Figure 1.

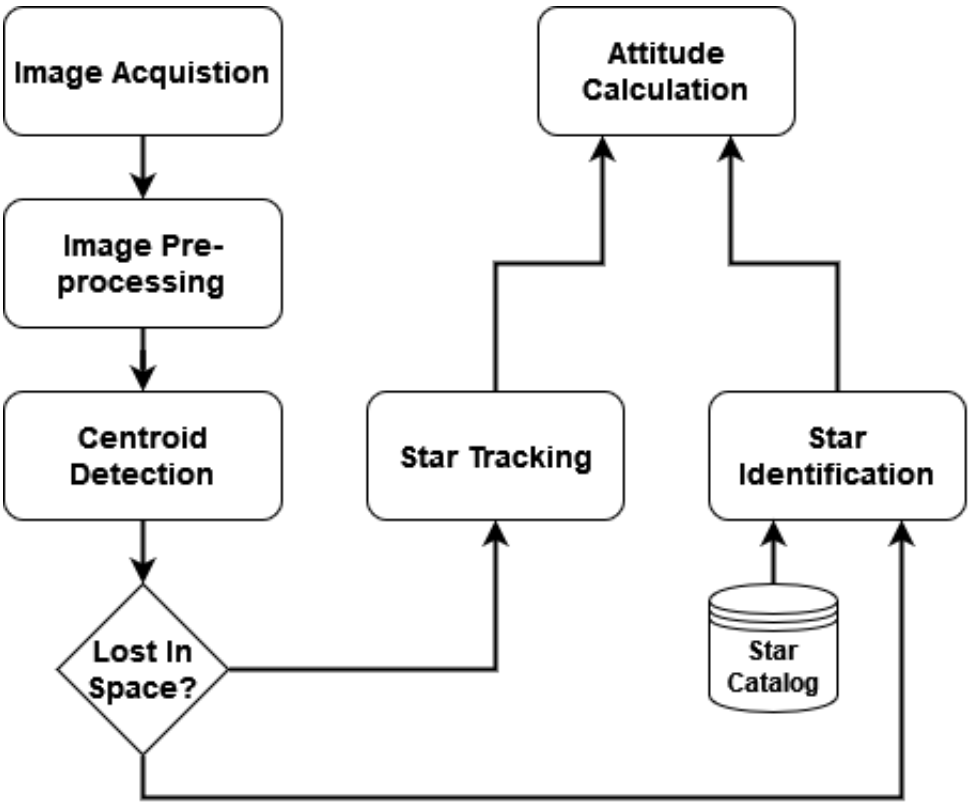


Figure 1. Simplified flow chart of a typical star tracker system operations.

2.3 Star Detection

Being capable of detecting stars is one of the core abilities of the star tracker system. When light from a star reaches the image sensor, it illuminates a pixel or a set of pixels, depending on the star's magnitude and the sensor's resolution and optics. This set of pixels visualizing the star is commonly known as a blob. As there is more than one pixel in the blob, its center or centroid, which represents the star's position in the image plane, is initially unknown. An algorithmic procedure must be introduced to obtain this centroid.

In the star detection process, there are also sources of noise that may affect the image quality and the centroids' determination. Generally, there are five types of noise when detecting stars - photon noise, readout noise, dark current noise, electronic noise, and mechanical noise. [20] These types of noise in the image may appear as illuminated pixels even in the absence of light, random variations in the signal, distorted images, and others. Light from other celestial objects, such as the Earth or the Sun, can affect the quality of the image in terms of star tracking and may render the star tracker incapable of performing its task.

Stars could be considered noise if not present in the stored star catalog because they do not provide meaningful information and may cause a faulty star identification process. Thresholding is a common preprocessing method in star tracker image acquisition, where pixel intensity threshold value is estimated based on probabilistic or empirical methods. Pixel intensity values below this threshold are considered not to originate from a star's light. When targeting certain magnitude stars, the blob size or intensity can also be used as a factor when filtering out unwanted stars. Empirical methods can obtain the best working value, though the light conditions might change once the satellite is in orbit. Statistical analysis can be performed to obtain this value based on the mean and standard deviations of the pixel intensities in the image. The statistical method of obtaining the threshold value was performed in prior work on the ESTCube-2 star tracker system. [21] The standard deviation of the image pixel intensity mean was set as the primary factor for differentiating between stars and image background noise. Equation 3 represents the standard deviation calculation, where N is the number of pixels, I is the intensity of a single pixel, and $I_{[mean]}$ is the mean value for pixel intensities. The illumination source is likely a star if a pixel's variation is five times the standard deviation. The probability of noise level being five standard deviations over the average, if it follows normal distribution, is 0.023%. [21]

$$I_{mean} = \frac{\sum_{i=1}^{N} I}{N} \tag{2}$$

$$S_n = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(I - I_{mean})^2} \tag{3}$$

Weighted sums of the blob pixel intensities can be used to find the center of the intensity. This method can be applied in both vertical and horizontal directions. The blob can occupy multiple rows, and the center of intensity for each row can be found with equation 4, where the n

is the number of pixels in the ongoing row of the blob, I is the intensity of the pixel, and X_{start} is the first detected pixel in the row of a blob.[22]

$$X_{center} = \frac{\sum_{n=1}^{w}(n \cdot I)}{\sum_{n=1}^{w} I} + X_{start} \tag{4}$$

A blob's final X position can be determined by summing all previously obtained blob centers of individual rows and dividing by the total number of rows in a blob as seen in equation 5. [22]

$$X_{blob} = \frac{\sum_{n=1}^{w} X_{center}}{number\ of\ rows} \tag{5}$$

The final Y position of a blob can be determined from the Equation 6, where the I_{row} is the integrated intensity of the row, Y is the vertical position of the blob, and Y_{start} is the vertical starting point of the blob.[22]

$$Y_{blob} = \frac{\sum(I_{row} \cdot Y)}{\sum I_{row}} + Y_{start} \tag{6}$$

There might be cases where a pixel in the middle of a blob is darker than the required intensity level. For example, these cases might be induced by damaged or "dead" sensor pixels. If not addressed, these cases might falsely stop the star detection process, even if the following pixels have brightness above the required threshold and are a part of the same star. For these situations, an additional ”ghosting” method can be introduced, which allows a limited amount of dark pixels without stopping the detection process. If this ghosting level is not exceeded and a new bright enough pixel appears, the new and previous dark pixels are considered part of the same blob. [22]

3 Prior Work

Before explaining the author's contribution in more detail, it is essential to mention and describe the effort of other ETCube-2 members related to the ESTCube-2 star tracker system. Though there are many contributions to different extents and from various parties, this section will focus on published work that is most relevant to the topic of this thesis.

The first contribution to mention is the bachelor's thesis written by Jürgen Laks, where the scope of the thesis covered the development of the ESTCube-2 star tracker system prototype and its initial tests. [23] Besides the hardware development, Laks was able to verify that the FPGA is operational and can be programmed and that the FPGA can obtain CMOS image sensor data. However, only 16×16-pixel image snippets were obtained because full image acquisition required large FPGA development efforts outside this thesis's scope. [23] Further discussion on hardware specifics primarily set by this thesis is covered in section 4.

Another important contribution to mention is a bachelor's thesis written by Andreas Ragen Ayal [21]. This work was mainly targeted toward implementing an adaptive thresholding method for star detection that is more robust against differentiating stars from the background noise. Ayal also worked on improving the star detection method proposed by Lindh [22] that involves the proposed thresholding method but was unable to achieve a synthesizable result. This thesis will explore integrating the Ayals thresholding method into a developed FPGA solution.

The last contribution to mention is Sandra Schumman's master's thesis exploring star identification algorithms. [24] The aim of this thesis was to explore different algorithms that would be the most optimal for star detection and matching and determine the expected performance of the ESTCube-2 star tracker under various conditions. This work was highly theoretical and simulation-based but provided solid information about the parameters and conditions the ESTCube-2 satellite and its star tracker system will experience.

4 Star Tracker Electronics

ESTCube-2 star tracker system hardware was already developed before the writing of this thesis. The initial prototype of the ESTCube-2 star tracker electronics was conceptualized and partially developed in 2015, consisting of selecting the CMOS image sensor, the creation of its initial electronics, optics, and mechanical hardware, and the decision to use the combination of FPGA, microcontroller (MCU) and external random access memory (RAM). The prototype consisting of all the components was made in 2019 and is documented in the thesis written by Jürgen Laks. [23]

The hardware of the ESTCube-2 star tracker can be summarised in three domains - image acquisition and processing, satellite interface, and power management. Of these three, image acquisition and processing are of the highest interest in terms of this thesis. This domain consists of an MT9P031 CMOS image sensor [1], Cyclone IV EP4CE22E22C6 FPGA [25], IS42S16320D-7TLI synchronous dynamic random access memory (SDRAM) device [26], MT25QL512ABB1EW9-0SIT flash memory device [27]. These are the components that are directly involved in attitude determination.

The FPGA is connected to all the previously listed devices, and a simple interface diagram is visualized in Figure 2. To ensure data transfer between these devices, the FPGA must be capable of using different communication protocols and interfaces with these devices. These communication protocols and interfaces include universal-asynchronous-receive-transmit (UART) for communication with the MCU, serial peripheral interface (SPI) for communication with the flash memory and again with the MCU, and parallel data transfer for communicating with the image sensor and the SDRAM.

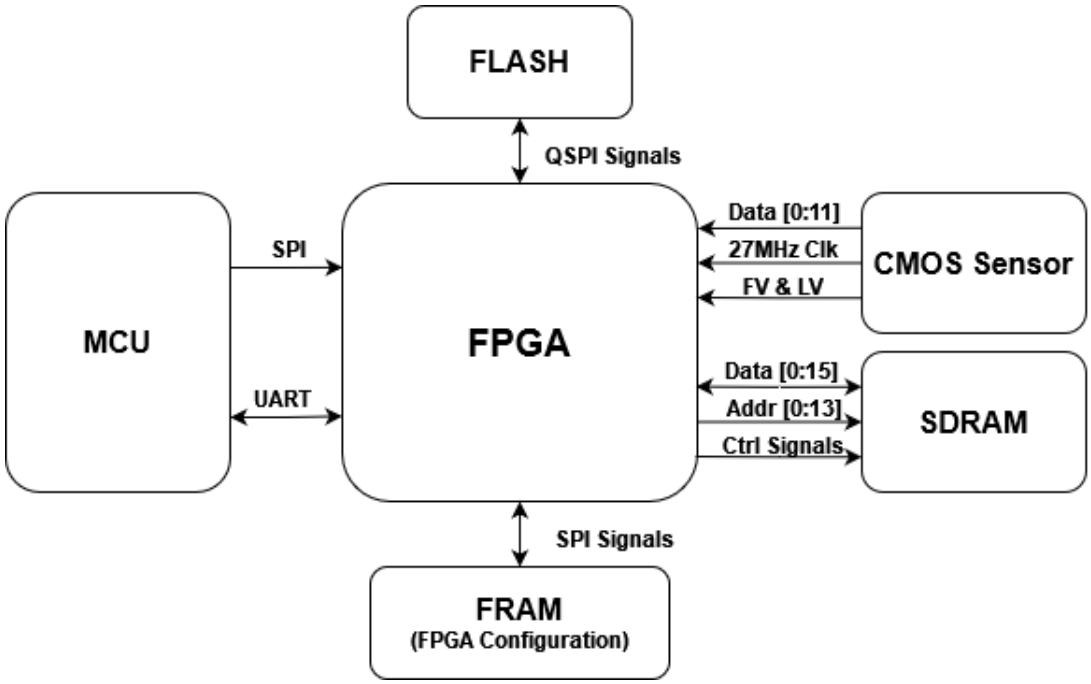


Figure 2. Interface diagram of the FPGA and other devices present on the ESTCube-2 star tracker system.

The prototype operated well, though additional changes and improvements were made once the design was migrated onto an engineering model - a model resembling the final flight model. The author of this thesis contributed to the finalization of the schematics and also designed the final printed circuit board (PCB) design for the engineering and flight model, where the final design of the ESTCube-2 star tracker electronics can be seen in figure 3, and the critical components are pointed out. However, a detailed description of the hardware in the following sections is avoided as it is outside this thesis's scope.

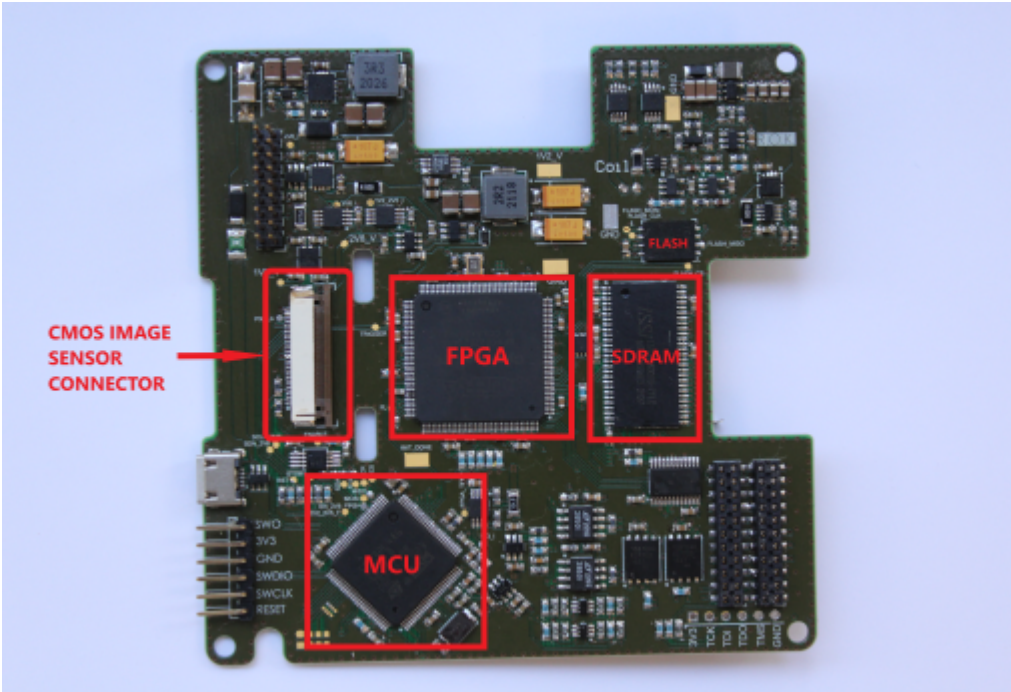


Figure 3. Image of the final ESTCube-2 star tracker PCB top side where the primary components are pointed out.

5 Methodology

The practical part of this thesis focuses on developing and implementing the ESTCube-2 star tracker FPGA internal hardware design and behavior capable of interfacing with external devices and detecting star coordinates in an obtained image. In this work, the behavior of the internal FPGA elements was programmed using the Very High-Speed Integrated Circuit Hardware Description Language (VHDL).

A development workflow was adopted and followed with some exceptions to ensure the proper performance of individual or multi-component designs. This workflow consists of multiple development steps and encourages testing and backtracking when results in different stages do not meet expectations. The states and transitions of this procedure are visualized in figure 4. The author attempted to write additional test components with pre-defined behavior to achieve similar conditions in the simulation and on the FPGA. The only inputs for the test components were clock signals. The designed components were placed inside the test component, and the output as UART data or individual signals were recorded.

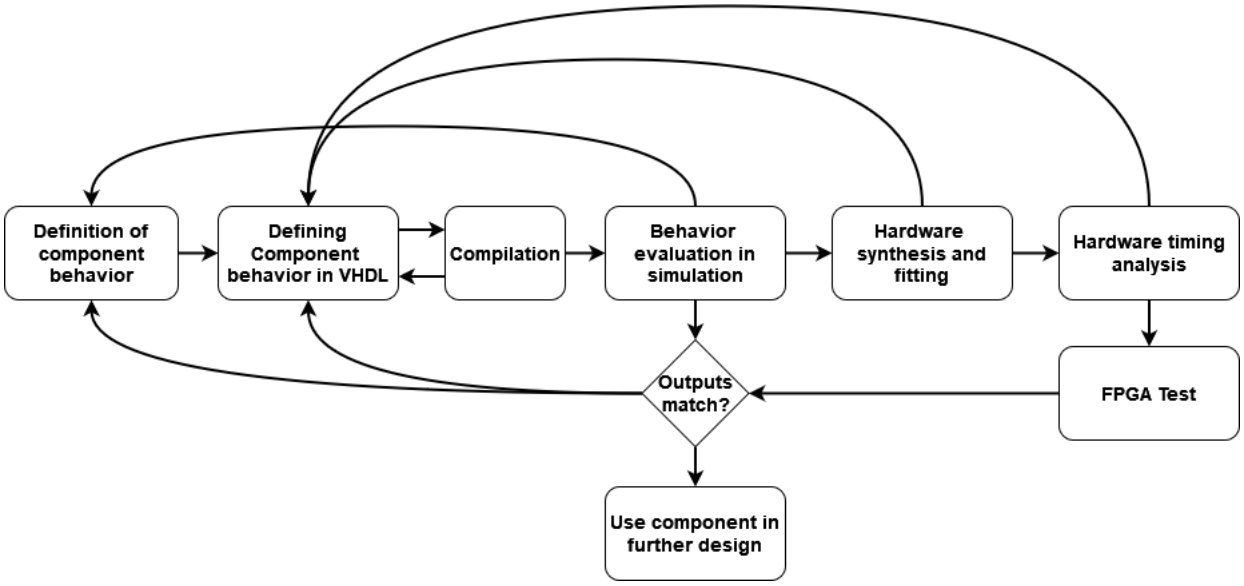


Figure 4. Flowchart visualizing the adopted workflow for creating different FPGA components

Based on the workflow, the first task is to define the requirements, behavior, and inputs/outputs of the to-be-designed component, followed by writing equivalent VHDL code. If the VHDL code cannot be compiled, it must be modified accordingly until it is successful. After the component has been successfully compiled, its internal and boundary signals can be evaluated in more detail. In this stage, more critical issues may arise; for example, some behavior or signal definition has been based on a false assumption, or some signals do not have correct timing characteristics, leading to their re-definition. If the simulation of a component behaves as expected in the simulation, an attempt is made to synthesize hardware analog to it and fit in on the FPGA. The synthesizer makes assumptions based on the provided VHDL code and infers hardware elements and their function. The synthesizer may falsely infer hardware from the provided VHDL code; for example, logic elements are used for data storage instead of using

internal RAM. It is also possible to define some elements that can be simulated but can't be synthesized entirely; for example, a flip-flop with two clock inputs. These sorts of situations then have to be corrected. After successful synthesis, timing characteristics must be carefully monitored to avoid any unpredicted violations due to signal lagging in complementary logic leading to hold/setup time violations. Some exceptions can be made if the timing violation is expected. Lastly, the component must be programmed to the FPGA, and the outputs must be measured. In an ideal case, the outputs match, indicating that the component design is correct and can be used in further development. Otherwise, the underlying reason for the inconsistency must be identified, and corrections must be made in the VHDL code or the predicted component design.

The usage of the described workflow may be unreasonable when directly working with signals to or from external devices. In this case, the testbench should also contain the behavior of the external device, which can significantly increase the complexity and time required to design the testbench and the simulation itself. Some components may require many output signals that are hard to verify in real life due to limited hardware probing points.

For maintaining the FPGA project, configuring the FPGA pinout, configuring timing constraints, performing compilation, synthesis, and fitting, viewing the inferred hardware and state machines, timing analysis, and obtaining the general information about the compiled design Quartus Prime 19.1 Lite Edition by Intel [1] was used. The Quartus Prime 19.1 Lite Edition by Intel provides access to Intel's multiple basic intellectual properties (IP) hardware units/cores. It must be noted that this is a free-access software that only provides the functionality and resources, including the IPs, in a limited evaluation mode. In the context of this thesis, IP is considered a standalone hardware unit that can be imported into an existing FPGA design and provides a specific hardware behavior. It is considered a property where the rights are reserved to its owner.

ModelSim-Intel® FPGA Edition Starter edition[2] was used to compile different components, combinations of components, and test benches and then view subsequent waveforms of signals provided by the resulting simulation.

The FPGA was programmed through the USB Blaster V2 Programmer/Debugger tool. A UART-to-USB hardware module designed by ESTCube members was used with HTerm software [3] to give commands and transfer data to and from the FPGA. a Rigol DS1202 oscilloscope was used to view and verify the outputs of the FPGA. The author also wrote small Python programming language scripts to perform minor utility tasks such as converting images into binary files, displaying images, and translating raw floating-point numbers into readable form. DAOStarFinder[4] function from the photutils.detection module in Python was used to obtain a reference for the centroid detection component. The PCB designed by the author (Figure 3.) was used to design and verify different FPGA components.

[1] https://www.intel.com/content/www/us/en/software-kit/660907/intel-quartus-prime-lite-edition-design-software-version-20-1-1-for-windows.html

[2] https://www.intel.com/content/www/us/en/software-kit/750368/modelsim-intel-fpgas-standard-edition-software-version-18-1.html

[3] https://www.der-hammer.info/pages/terminal.html

[4] https://photutils.readthedocs.io/en/stable/api/photutils.detection.DAOStarFinder.html

6 Implementation of Peripheral and Controller Layer

The ESTCube-2 star tracker FPGA is connected to various external devices. Data transfer between these devices and synchronization with the other FPGA processes must be managed and maintained. As the FPGA does not have predefined hardware functionality, it has to be implemented by the author. The final design is expected to contain components, clock signals, data widths, and other conditions that may cause a complex solution if not approached systematically. This section discusses the controller and peripheral layers, which aim to reduce the complexity of the interface between internal FPGA components and external devices.

Each external device has a component that is considered to be on the peripheral layer. They reside on the edge between the FPGA and the environment, have a direct connection with the pins of the FPGA, and pass signals from the controller layer component to them. The peripheral layer component consists of data buffers, data type conversions, and controller layer components. The image sensor is an exception, having only a controller component, because data is managed in real-time, and there is no need for additional data buffer. The image sensor controller component is also relatively simple, and there is no need for further abstraction.

The controller layer is less abstract than the peripheral layer. It defines the precise signal changes and timing for generating the proper hardware behavior for interacting with the external device. This layer also has device-specific knowledge, such as device operational codes or routines unique to the device. Generally, the internal FPGA components should not directly interface with this controller component as it would introduce additional constraints to the internal components.

Figure 5. visualizes the proposed structure of the ESTCube-2 star tracker FPGA. The author must note that SPI and Flash peripherals (indicated with dashed lines in the figure) were not implemented; however, the groundwork was laid, and the same approach could be maintained to further develop these components.

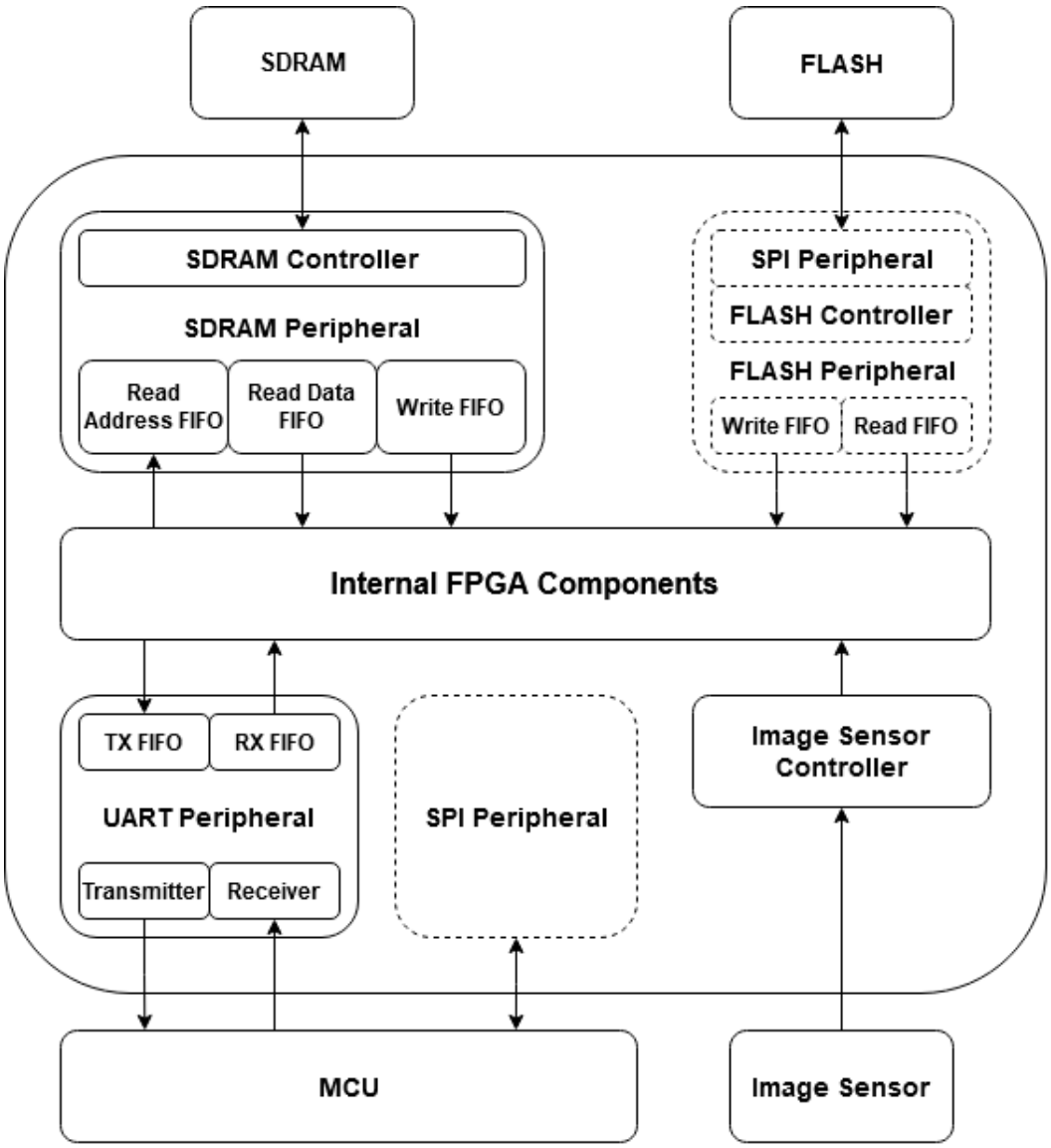


Figure 5. Block diagram of the proposed peripheral layer of ESTCube-2 star tracker FPGA. Arrows only represent the direction of data flow. Control signals are not visualized in this diagram

6.1 Peripheral Interface

The ESTCube-2 star tracker FPGA communicates with different external devices that run on different clocks and have different communication protocols, data lengths, and other peripheral-specific parameters. Different clock domains are asynchronous, and to transfer data, some form of synchronization has to be implemented. Peripherals may have internal procedures and states that may not allow continuous interface with them, leading to lags in data communication or even data loss. Thus, a data buffer interface was determined to be used on the peripheral boundaries, allowing different clock domains to be used and simplifying the overall communication. This design choice leads to higher usage of available resources but allows more efficient and manageable design for other components.

Initially, there was an attempt for a universal, easy-to-use interface component to manage data between different peripherals, regardless of whether the peripherals run on the same or different

clocks. This interface component could operate at a lower frequency, though it did not properly function with higher-frequency peripherals such as SDRAM. Thus, the idea was eventually abandoned, and the author moved to Intel's existing first-in-first-out (FIFO) IP. [28] This IP proved to have the required functionalities; it was easily configurable and, most importantly, supported two different clocks.

6.1.1 UART Peripheral Layer Implementation

UART hardware interface is a common form of serial data communication between devices due to its simplicity. The ESTCube-2 star tracker uses UART to transfer data and commands between the MCU and the FPGA. UART is a configurable hardware communication protocol where both sides must employ the same configuration for successful communication. [29] This configuration includes the following:

- Length of data - 5 to 9 bits
- Data transmission order - most significant bit (MSB) or least significant bit (LSB) first
- Baud rate
- Parity check
- Amount of stop bits - 1 or 2 bits

UART peripheral layer component, which consists of a transmitter, receiver, and two FIFO buffers for each data direction, was made. It was decided that the UART would be created as simply as possible, as it does not play a crucial role in the star tracker operation. The UART packet and configuration used in the FPGA can be seen in figure 6. The configuration options can also be modified through the configuration file, which defines constants used in the design. Modifiable options include the baud rate, data transmission order, data length, and amount of stop bits. The parity check option was not implemented.

Figure 6. Configuration of UART interface in ESTCube-2 star tracker FPGA. The baud rate is set to 1 000 000 symbols/s

Two identical FIFOs transfer the data between the UART peripheral and the internal FPGA components. These FIFOs are synchronous and in the show-ahead mode, meaning that the same clock drives them and that the data must be acknowledged and not requested. The configuration of the FIFO can be seen in Table 1.

Table 1. Configuration of the FIFO IP component used in the UART peripheral

Side	Frequency (MHz)	Data width (bits)	Words	Mode
Input	100	8	64	Synchronous, show-ahead
Output				

6.1.2 Image Sensor Controller Layer Implementation

The controller layer component was made to acquire an image from the MT9P031I12STM-DP image sensor. [1] The image streaming is synchronized with a 27 MHz clock signal generated by the sensor. This clock signal is also supplied to the FPGA to synchronize the data capture properly. In addition to the clock and data lines, the image sensor provides line-valid (LV) and frame-valid (FV) signals indicating when the streamed image's pixel data contains valid pixel data. Figure 7 visualizes the timing diagram of the FV signal, LV signal, Clock, and 12-bit pixel data relationship.

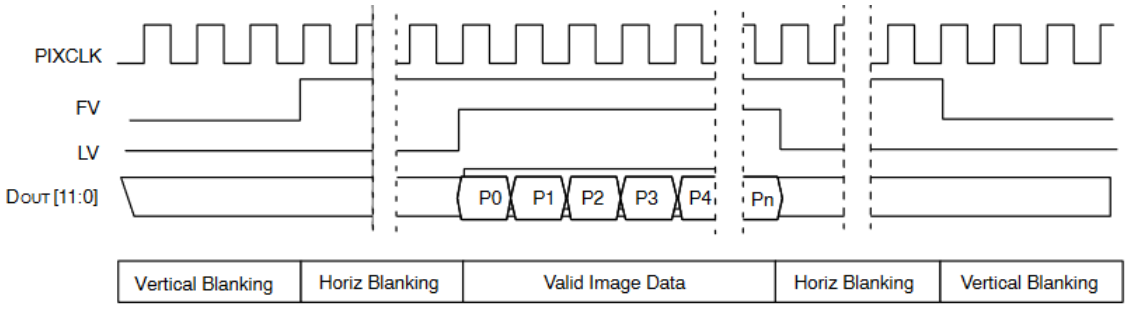


Figure 7. Timing diagram of the MT9P031I12STM-DP image senor visualizing the relationship between different image sensor signals [1]

The CMOS image sensor continuously streams out 1944×2592 pixel images row-by-row.; no trigger signal is required from the FPGA side to initiate the image read-off. When the image sensor controller receives an image capture signal from internal FPGA components, it is responsible for detecting the beginning of a new image frame to begin the capture process. During the image capture, the controller component indicates when the pixel data is valid (detects when both FV and LV signals have a high logic level). It keeps track of the image column and row and outputs a signal that indicates when all active image pixels have been processed.

No peripheral layer was designed because the image capture process is relatively simple; as the CMOS configuration is uploaded by the ESTCube-2 star tracker MCU, the FPGA only reads the data and has no control over the data streaming process, making additional buffers redundant and unnecessary.

6.1.3 SDRAM Peripheral

One of the more complicated devices in the ESTCube-2 star tracker system is the IS42S16320D-7TL SDRAM, which allows high-rate parallel data storage and reading. [26] These advantages also lead to more complex handling procedures from the FPGA point of view. The complexity of the SDRAM device arises from the precise timing requirements and the various states the device can and must transition through in order to perform read, write, or some other internal function of the SDRAM. The purpose of this device in the star tracker system is to store the captured image, which allows it to process the image after it has been taken or transmit it to other satellite systems.

The total memory capacity of the IS42S16320D-7TL SDRAM device is 512 mega-bits. The memory architecture of the device consists of 4 banks, each with 8192 rows and 1024 columns. Every memory position can store a single 16-bit data point. A single SDRAM bank has a capacity of 16 megabytes, so it can potentially store two full CMOS sensor images if a single 16-bit memory position stores one full 12-bit pixel and then an additional 4 bits of another pixel, requiring asymmetrical image storage. In this thesis, the asymmetrical memory storage was avoided due to its complexity, leading to a solution where only one pixel is stored in one memory position, utilizing 75% of it. Due to this inefficiency, only one full CMOS sensor image can be stored in a single SDRAM bank. Different banks can be used to store images, increasing the number of images that can be stored. A visualization of the SDRAM memory architecture can be seen in Figure 8.

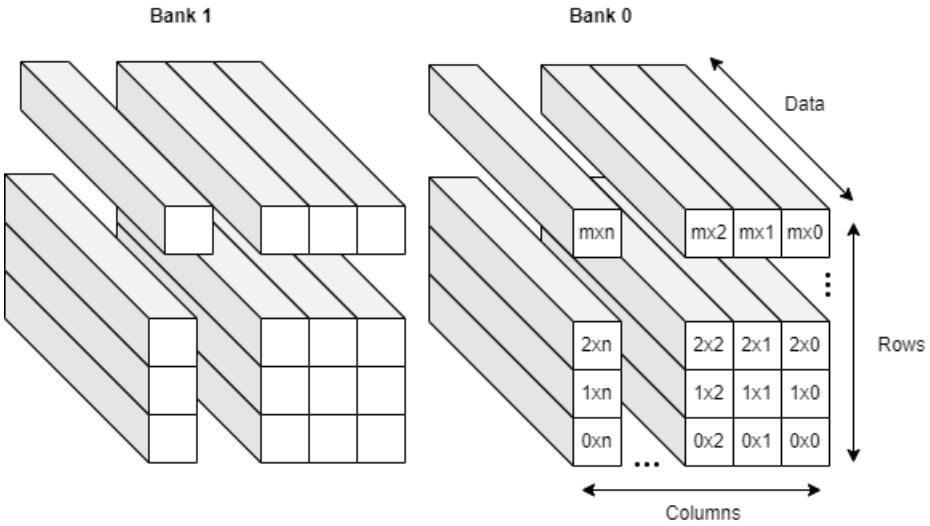


Figure 8. Memory architecture of a typical SDRAM device, which resembles a set of three-dimensional matrices.

A controller layer component was made for the SDRAM device, which is responsible for the power-up routine of the device, issuing commands, precise signal timing, and managing the data and address lines. The central element of the SDRAM controller component is the state machine, where the states initiate different SDRAM procedures and keep track of the latency periods required to perform a procedure. The visualization of this state machine can be seen in

Figure 9.

The SDRAM goes to the IDLE state after the powerup routine, which includes a $100\ \mu s$ wait period, one precharge procedure, two auto-refresh procedures, and mode programming. From the IDLE state, the SDRAM controller component can start operating normally. To access a column of some row, the row must be activated first, which is handled by either ”ACTIVATE WRITE” or ”ACTIVATE READ” states. Both states perform the same operation but differ in the paths they can take afterward. When a row has been activated, data reads and writes are performed through the READ and WRITE states, respectively. When the write or read is requested on a column residing on a row different from the active row, the active row must be precharged, and the activation procedure must be issued once again for the new row.

For the IS42S16320D-7TL SDRAM device, a refresh procedure must be issued every $8\ \mu s$ – every 800 cycles if the device is clocked with 100 MHz to avoid data corruption. An internal timer process in the SDRAM controller component keeps track of this period and issues a ”Do Refresh” signal when the timer has ended. Each state of the state machine monitors this signal, and when it becomes true, the state transitions to the IDLE state. If a row is active, it is deactivated with a PRECHARGE procedure, and finally, the state can move to the REFRESH state. REFRESH state issues a command for a refresh procedure, waits for ten clock cycles, clears the ”Do Refresh” signal, and then transitions back to the IDLE state.

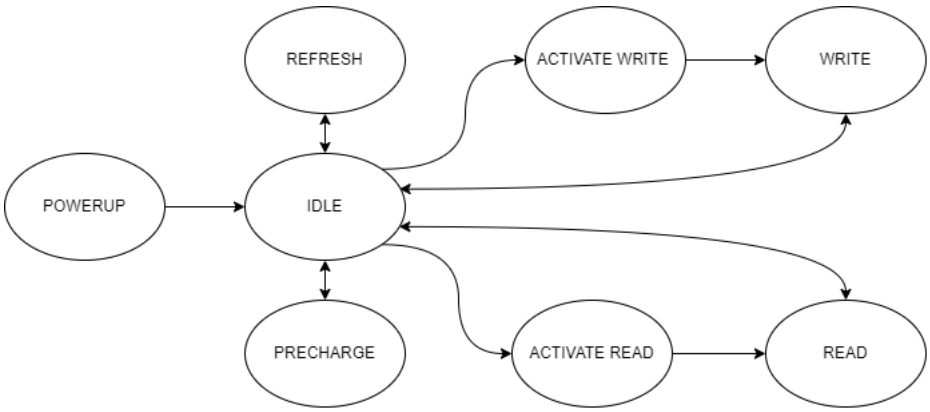


Figure 9. Simplified SDRAM controller state machine responsible for managing and transitioning between different SDRAM activities

In addition to the SDRAM controller layer component, a peripheral layer component was also developed to reduce SDRAM access complexity from other FPGA component perspectives. The peripheral layer comprises three Intel FIFO IPs and an SDRAM controller component described in the previous paragraphs of this section. Control output signals of these FIFOs, such as ”Empty” and ”Full”, are the core triggers for state transitions in the state machine visualized in Figure 9. The general parameters of all three FIFOs can be seen in Table 2.

An FPGA component can store data in the SDRAM using the peripheral layer by assembling a data composite and setting the FIFO ”Write Request” input signal to a logic high level. Write data composite is 41-bit long, and it consists of a column address (10 bits), row address (13 bits), bank address (2 bits), and data to be stored (16 bits). The actual data storage process is handled by the SDRAM peripheral and controller components that handle the FIFO data based on the

SDRAM device accessibility and state.

Reading requires two FIFO components; one FIFO component receives address composite from some FPGA component, and the other provides data corresponding to the address provided – address comes in, and data goes out. The read address composite is 24 bits long and contains the same structure as the beginning of the write FIFO composite – a column address (10 bits), a row address (13 bits), and a bank address (2 bits). The read data FIFO component just stores 16-bit long data that was read from the SDRAM.

Table 2. SDRAM peripheral layer FIFO general parameters

<table>
<tr><th colspan="5">Write FIFO</th></tr>
<tr><td>Side</td><td>Frequency (MHz)</td><td>Data width (bits)</td><td>Words</td><td>Mode</td></tr>
<tr><td>Input</td><td>27 or 100</td><td rowspan="2">41</td><td rowspan="2">128</td><td rowspan="2">Dual-clock, show-ahead</td></tr>
<tr><td>Output</td><td>100</td></tr>
<tr><th colspan="5">Read Address FIFO</th></tr>
<tr><td>Side</td><td>Frequency (MHz)</td><td>Data width (bits)</td><td>Words</td><td>Mode</td></tr>
<tr><td>Input</td><td rowspan="2">100</td><td rowspan="2">25</td><td rowspan="2">128</td><td rowspan="2">Synchronous,show-ahead, indicates when FIFO is almost empty</td></tr>
<tr><td>Output</td></tr>
<tr><th colspan="5">Read Data FIFO</th></tr>
<tr><td>Side</td><td>Frequency (MHz)</td><td>Data width (bits)</td><td>Words</td><td>Mode</td></tr>
<tr><td>Input</td><td rowspan="2">100</td><td rowspan="2">16</td><td rowspan="2">128</td><td rowspan="2">Synchronous, show-ahead, indicates when FIFO is almost full</td></tr>
<tr><td>Output</td></tr>
</table>

7 Implementation of FPGA Internal Components

7.1 Implementation of Command Center Component

An internal FPGA component, further in the text referred to as the command center component, was created to make testing and communication with the FPGA less challenging. This component consists of main process and a command-specific process for each command. The command center is connected to the SDRAM peripheral, UART peripheral, image controller, and other system-level processes and algorithms. A simplified block diagram of the core components is shown in Figure 10.

Different command-specific processes may have to reuse the same external components. Multiplexing is used to switch between command-related signals that are connected to these components, which also allows for saving resources by not using duplicate components. An example of this multiplexing can be seen in the Figure 10, where the red dashed line indicates the sharing of the image controller by multiplexing. The selection of a multiplexer channel is dependent on the active command. In this case, the image controller component is shared between 0x03 and 0x04 commands. Commands and command-specific processes are described further in this section.

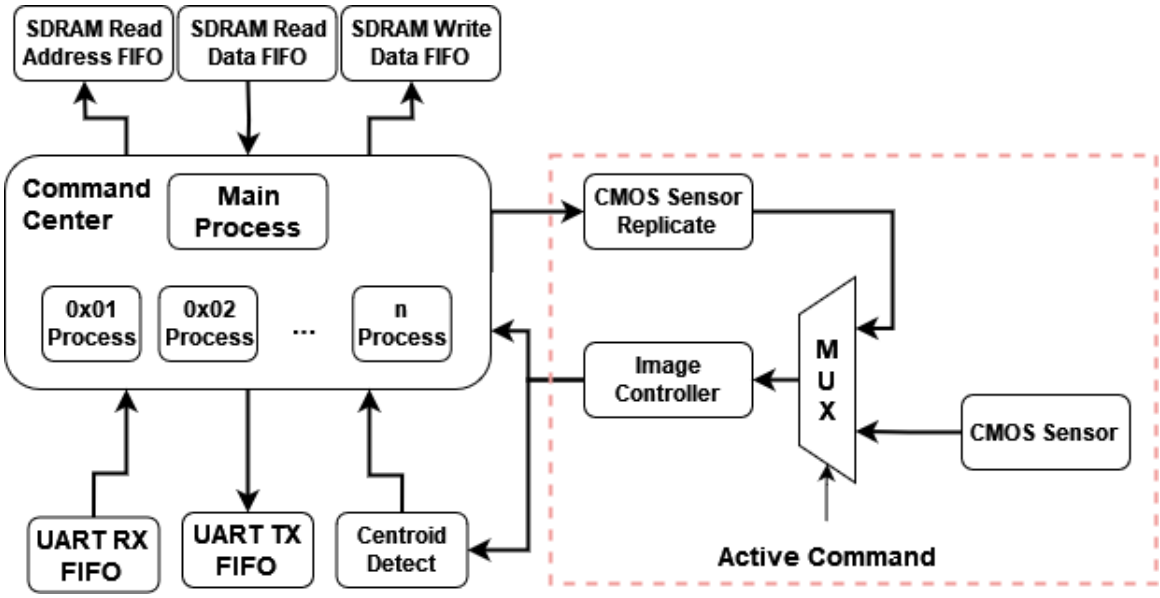


Figure 10. Simplified block diagram of the command center component containing connections with other core components. Red dashed line indicates area where multiplexing takes place to share the image controller component

The main process is responsible for parsing the messages from the UART receiver FIFO, decoding the messages, and managing other command-specific processes. A typical command message structure consists of one identification (ID) byte (0xEC) and one command byte, followed by command-specific data transmitted or received. A summary of the commands can be seen in Table 3.

Table 3. Summary of the commands that the command center can parse and the subsequent data that is either received or transmitted

Command	1. ID Byte	2. Command Byte	3.,4.,...
Write Test Image in SDRAM	0xEC	0x01	3. Row address (7 downto 0) 4. Row address (13 downto 8) 5. Column address (7 downto 0) 6. Column address (9 downto 8) 7. Pixel data (7 downto 0) 8. Pixel data (15 downto 0)
Read Test Image		0x02	3. 0th Row, 0th Column pixel data 4. 0th Row, 1st Column pixel data ... 1000x1500 test image pixel data stream Column by column, row by row
Perform Test Image Centroid Detection		0x03	-
Capture CMOS sensor image + Store in SDRAM + Centroid Detection		0x04	-
Read CMOS snesor image		0x05	-
Download Centroid Data		0x07	3. First Centroid X (7 downto 0) 4. First Centroid X (15 downto 0) 5. First Centroid X (23 downto 16) 6. First Centroid X (31 downto 24) 7. First Centroid Y (7 downto 0) 8. First Centroid Y (15 downto 8) 9. First Centroid Y (23 downto 16) 10. First Centroid Y (31 downto 24) 11. First Centroid I (7 downto 0) 12. First Centroid I (15 downto 0) 13. First Centroid I (23 downto 0) 14. Second Centroid X (7 downto 0) 15. ... Continues for up to 32 centroids

While no active command has been issued, the command center is in an IDLE state, which keeps all other connected components in a reset state. Once a byte is available in the UART peripheral receiving FIFO, the command center process checks if the byte corresponds to the identification byte 0xEC. If it does not match, the byte is discarded, but if there is a match, the process moves into the DECODE state that waits for the next byte in the UART receiver FIFO. When a command byte is received that corresponds to a valid command, the command center process waits for additional data if the command requires it or moves to the ACTIVE state, toggles the reset signal of the processes that correspond to the command tasks, and waits for a signal that indicates that the process has finished. After receiving a ”Done” signal, the process returns to the IDLE state.

7.1.1 Implementation of Test Image Write and Read Commands

This command aims to write test image data into the SDRAM. The main element responsible for successfully implementing the command is the command center's main process that reads and temporarily stores the received address and data information and initiates the 0x01 command-specific process. The SDRAM write FIFO 41-bit data input is set with a data composite consisting of the received address and pixel data, and the command-specific process triggers the write signal if the FIFO is not full. From the moment the last packet is received, it takes exactly three clock cycles from the Command Center perspective to finish this process and return to the IDLE state.

The received information consists of two bytes of row address, two bytes of column address, and two bytes of pixel data. Two bytes are associated with a column and a row because the SDRAM row and column lengths are 13 and 10 bits, respectively. This also means that the received row and column bytes contain some information that is not used and is discarded. The 0th row of the test image is also offset by 6000 rows in the SDRAM because the test image and the obtained CMOS image are stored in the same bank of the SDRAM. Visualization of this storage can be seen in Figure 11.

Complimentary to the 0x01 test image write command is the 0x02 test image read command, which allows the user to read off the test image written in the SDRAM for verification. This command comprises the main process, the 0x02 command-specific process, the SDRAM peripheral, and the UART peripheral. The 0x02 command-specific process iterates and requests all 1000×1500 pixel addresses from the SDRAM Peripheral through the read address FIFO. Once the data appears on the output of the read data FIFO, the command-specific process transfers this data to UART transmitter FIFO and keeps track of how much data has been processed. When all data has been transferred, the command-specific process sets a signal to logic high level that indicates to the main process that it has completed its task.

Because the image spatial resolution is 1000×1500, but the SDRAM row only has 1024 positions, two SDRAM rows are used to store a single test image row. Because of this approach, every two test image rows leads to 476 unutilized memory positions in the SDRAM. The equations 7. and 8. describe the relation between SDRAM memory positions and test image pixels that are being stored. Visualization of this storage can be seen in Figure 11.

$$\begin{aligned} R_{SDRAM} &= R_{Test} \cdot 2 && , if\ C_{Test} < 1024 \\ &= R_{Test} \cdot 2 + 1 && , if\ C_{Test} \geq 1024 \end{aligned} \tag{7}$$

$$\begin{aligned} C_{SDRAM} &= C_{Test} && , if\ C_{Test} < 1024 \\ &= C_{Test} - 1024 && , if\ C_{Test} \geq 1024 \end{aligned} \tag{8}$$

7.1.2 Implementation of Test image Centroid Detection and Transfer Commands

Test image centroid detection is invoked by the 0x03 command, which results in centroid detection for the image uploaded by command 0x01. The process for this component is slightly more complicated compared to the previous commands. This command involves SDRAM peripheral read address FIFO and read data FIFO (described in section 6.1.3), it involves the centroid detection component (described in section 7.2), image controller (described in section 6.1.2), and an additional component that replicates the CMOS image sensor control signals. As other command-specific processes also use these components, their input signals must be multiplexed between them once a process is active. An example of similar multiplexing was visualized in Figure 10.

The main goal of this command-specific process is to access the address of the test image stored in the SDRAM by writing the address in the SDRAM peripheral read address FIFO. Several cycles later, the data in the corresponding address appears in the SDRAM peripheral read data FIFO. This data is then supplied to the CMOS sensor replicate component that outputs this data along with the frame-valid and line-valid signals to the image controller and further to the centroid detect component. When all of the image data has been processed and centroid component also indicates that it has finished the procedures, the main process move back to the IDLE state.

The detected centroid can be accessed through the 0x07 command that will read the centroid data from the centroid detect component internal RAM. There are 255 positions for 88-bit wide centroid data composites, but this number can be easily modified based on the application needs. The centroid component is shared between the command 0x03 and 0x04, so 0x07 transmits either the test image centroids or CMOS image sensor centroids, depending on which command was invoked last.

The centroid data is passed to the UART peripheral component that transmits it over UART in multiple packets. For a single centroid, an 11-byte frame has to be transmitted – 32-bit X position, 32-bit Y position, and 24-bit intensity. The range 0th down to 31st bit contains a floating point number representing the centroid X coordinate, the range 63rd down to 32nd bit contains a floating point number representing the Y coordinate, and the range 87th down to 64th bit contains an integer number representing the centroid's intensity.

7.1.3 Implementation of CMOS Sensor Image Capture, Storage, and Centroid Detection Command

The 0x04 command initiates CMOS image sensor capture, stores it in SDRAM and runs the pixel data stream through the centroid detection component that detects centroids in real-time. The command-specific process initiates image retrieval from the image sensor controller, the data is passed to the SDRAM peripheral write FIFO and to the centroid detection component. The command-specific process waits for the indication from the image controller and the centroid detection component that all data have been processed. The process then indicates to the main process that the command has been executed.

A single row of the CMOS sensor (2592 pixels in a row) does not fit inside a single row of the SDRAM (1024 positions in a row). In the implementation provided in this thesis three SDRAM rows are used to store a single image row. The storage of the image can be further explained with equations 9 and 10, where R_{SDRAM} and C_{SDRAM} stands for SDRAM row and column, respectively, and R_{Cam} and C_{Cam} stand for CMOS sensor image pixel row and column, respectively. For every image row, 480 memory positions remain unutilized in the SDRAM. However, knowing the equations 9. and 10., exact location of these unutilized memory positions can be determined and then used to store some other form of data if necessary. The storage is further visualized in Figure 11, which also indicates the position of the test image.

$$\begin{aligned} R_{SDRAM} &= R_{Cam} \cdot 3 &&, if\ C_{Cam} < 1024 \\ &= R_{Cam} \cdot 3 + 1 &&, if\ C_{Cam} < 2048 \\ &= R_{Cam} \cdot 3 + 2 &&, if\ C_{Cam} < 2592 \end{aligned} \tag{9}$$

$$\begin{aligned} C_{SDRAM} &= C_{Cam} &&, if\ C_{Cam} < 1024 \\ &= C_{Cam} - 1024 &&, if\ C_{Cam} < 2048 \\ &= C_{Cam} - 2048 &&, if\ C_{Cam} < 2592 \end{aligned} \tag{10}$$

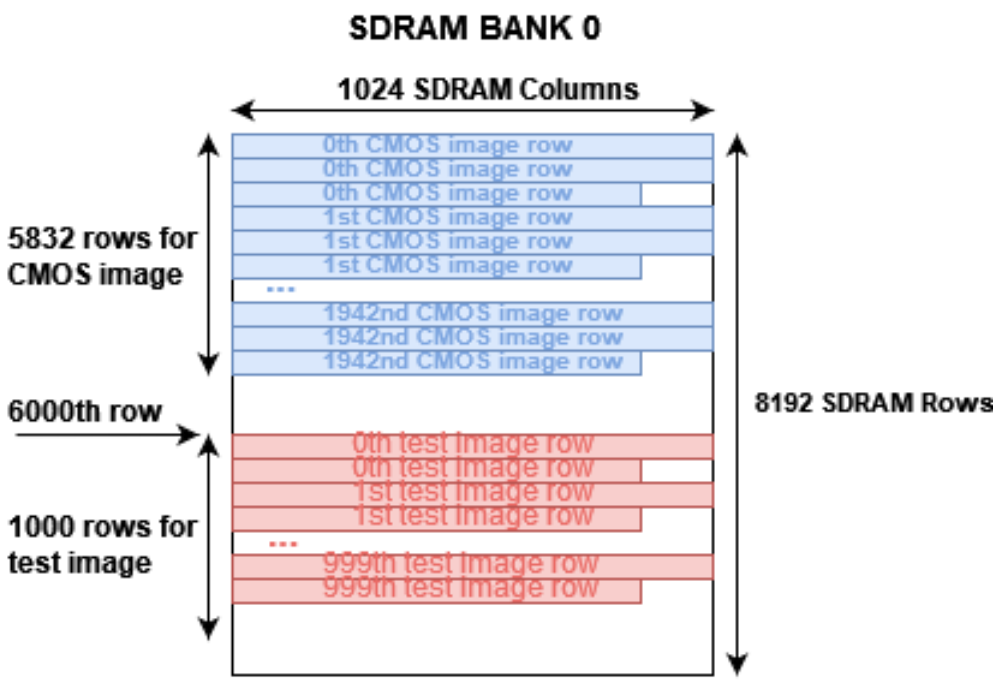


Figure 11. Visualization of SDRAM bank 0 memory allocation for CMOS sensor image and the test image

Finally, the centroid detection at this point only provides a signal that the image has been processed. If any centroids are detected, they are stored in FPGA internal RAM, which is part of the centroid detection component and can be accessed through the 0x07 command.

7.1.4 Implementation of CMOS Sensor Image Read-Out Command

Command 0x05 initiates the CMOS sensor image read-out from memory. This is a relatively simple task started off by the command-specific process requesting the pixel addresses of the CMOS image from the SDRAM peripheral and subsequently from the SDRAM device. The received data is then transferred to the UART peripheral, and transmitted out of the FPGA. However, this process takes significant time because 1944×2592 pixels must be transmitted, resulting in approximately 1 minute download time through UART at 1 Mbaud/s. Currently, only 8 of the most significant bits (MSB) are transferred instead of all 12 bits of a pixel to decrease the download time by half. The full 12 bit implementation was not implemented as in this thesis this command was used to verify that the image has been successfully captured and stored. However, simple modifications, consisting of using an additional cycle per memory access to transfer the less significant bits to the UART peripheral, can be added to download images with 12-bit wide pixel data.

7.2 Centroid Detection in Images

Before the stars can be identified in the image, the system has to detect the star positions. The general principles of star detection were discussed in section 2.3. This section focuses on the ESTCube-2 star tracker FPGA-based algorithm implementation that detects star centroids in real-time as the pixel data is streamed from the image sensor. The general concept for detecting the set of potential star pixels in the image was based on the implementation described in [22], which also focuses on real-time star detection with an FPGA-based system. The goal of this algorithm is to execute equations 4, 5, and 6 as the pixels are being streamed out of the image, and then store the obtained coordinates in memory. However, the proposed steps and implementation of [22] were not necessarily followed, and a modified version was implemented. For example, during the algorithm implementation, IP cores capable of performing floating point calculations were used to obtain sub-pixel centroid coordinates, though only in evaluation mode, available in the Intel® Quartus® Prime Lite Edition. [30]
The following list summarizes the simplified steps of the implemented centroid detection component:

1. Detect a set of nearby potential star pixels in a row (star-line in further text)
2. Store parameters of the star-line once it has ended
3. Compare it to previously obtained intermediate star elements that are stored in a FIFO memory component
    - If the FIFO is empty, then store star-line parameters as a new star element
    - If there is a match, merge the star-line parameters with an intermediate star element
    - If the row of the star line is two rows lower than the intermediate star element, then the star element is released as a completed star
    - If the star-line element does not match any intermediate star elements, then it is stored as a new star element

Steps 1. and 2. are handled by a single process, which detects a set of nearby pixels in a row (star-line) and keeps track of the information related to these adjacent pixels. When the star-line ends, this information is stored in a FIFO memory component (FIFO-A in further text). A more detailed description of this process is described in subsection 7.2.1.

At the core of step 3. is a process that handles conditions that may arise when processing previously stored star-line elements. This process iteratively merges star-line elements that are a part of the same potential star creating an intermediate star element. This process stores intermediate star element data composites in another FIFO memory component (FIFO-B in further text). It releases these elements from the FIFO-B when all of the pixels of a potential star have been processed. A more detailed description of this step is described in the subsection 7.2.3.

7.2.1 Star-Line Element Storage

The first process involves detecting a set of nearby pixels in a row being streamed out of the sensor. The further text will reference this set of pixels as a star-line. A star will consist of more than one star-line element. Because the image sensor streams out the pixels row-by-row, there will be additional data between the next star-line element in the following row, including other star-line elements that are part of a completely different star. Due to this reason, FIFO-A is used to store these star-line elements for further processing, where the relation between star-line elements is based on the stored characteristics. Because this process and the input side of the FIFO-A directly interact with the image data stream, they are synchronized with the clock signal supplied by the image sensor.

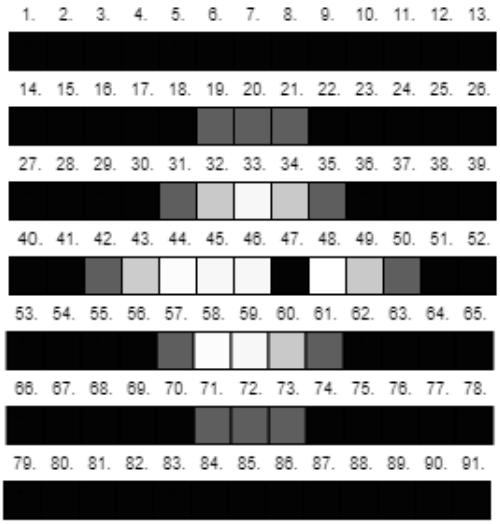
Each pixel in the row is compared to a predetermined threshold, and if the pixel value exceeds the threshold, it is considered a potential star pixel. When a bright enough pixel is detected, star-line detection begins by storing the active image row and the column, setting the pixel count to one, and setting the total intensity of the star-line to the pixel's intensity. If the following pixels exceed the threshold value, the pixel count is incremented, and the pixel's intensity is summed with the total intensity.

In some situations, there might be pixels below the required threshold, even if the following pixels are bright enough and are a part of the same star. One such cause could be a dead pixel in the image sensor, which leads to the pixel being permanently black. An additional "ghosting" period was included to avoid such cases, allowing a predefined amount of pixels under the required threshold but not stopping the detection process. When the ghosting period is exceeded, the detection process stops, and a data composite consisting of data related to the star-line element is stored in the FIFO-A. Figure 12 summarizes the data contents of the data composite stored in the FIFO-A.

Star-line element data composite (72 bits)				
Start Column (12 bits)	Active Row (12 bits)	Pixel Count (8 bits)	Weighted Pixel Sum (20 bits)	Total Pixel Intensity (20 bits)

Figure 12. Data composite consisting of information describing a set of nearby pixels detected in a row. This data composite is 72 bits long and is stored in FIFO-A, where it will wait for further processing

Typical actions during this process are visualized in Figure 13., where an example image of a star and a table of actions taken at each pixel is presented. One case when the recorded values are discarded is when the number of pixels in the set of row pixels exceeds a predefined limit, which indicates that there are too many pixels to be a part of a star. Similarly, the recorded values are discarded if there are too few pixels.



Actions	Pixels
Record active row	32, 43, 58
Record start column	32, 43, 58
Add to total intensity	32, 33, 34, 43, 44, 45, 46, 48, 49, 58, 59, 60
Increment pixel count by 1	32, 33, 34, 43, 44, 45, 46, 49, 58, 59, 60
Increment pixel count by 1 + ghost count	48
Increment ghost count	35, 36, 47, 50, 51, 61, 62
Reset ghost count	37, 48, 52, 63
Store composite	37, 52, 63
Reset values (including ghost count)	37, 52, 63

Figure 13. Example image of a star with labeled pixels. The label indicates the order in which pixels would be accessed in the image. The table describes what actions are taken during pixel access.

7.2.2 Calculation of X_{Center} and Y_{center}

The center values for star-line and intermediate star elements are obtained with a component that includes multiple Intel IP cores that allow different floating-point manipulations and arithmetic functions. Figure 14 visualizes the signal flow, inputs/outputs, and the used IP cores and their connections. Each IP core is associated with a latency period between the change in the input signal and valid states appearing on the output. In total, 22 cycles are required to obtain a valid output from this component.

The difference between equations 4 and 6 are the input values. Based on this observation, it was possible to reuse the component to obtain the star-line element X_{center} value and intermediate star element y_{center} value by multiplexing the input signals in different states of the centroid detection component. The X_{center} values are calculated by default, and the multiplexer switches to Y_{center} calculations only when the intermediate star elements in FIFO-B are completed and have to be released.

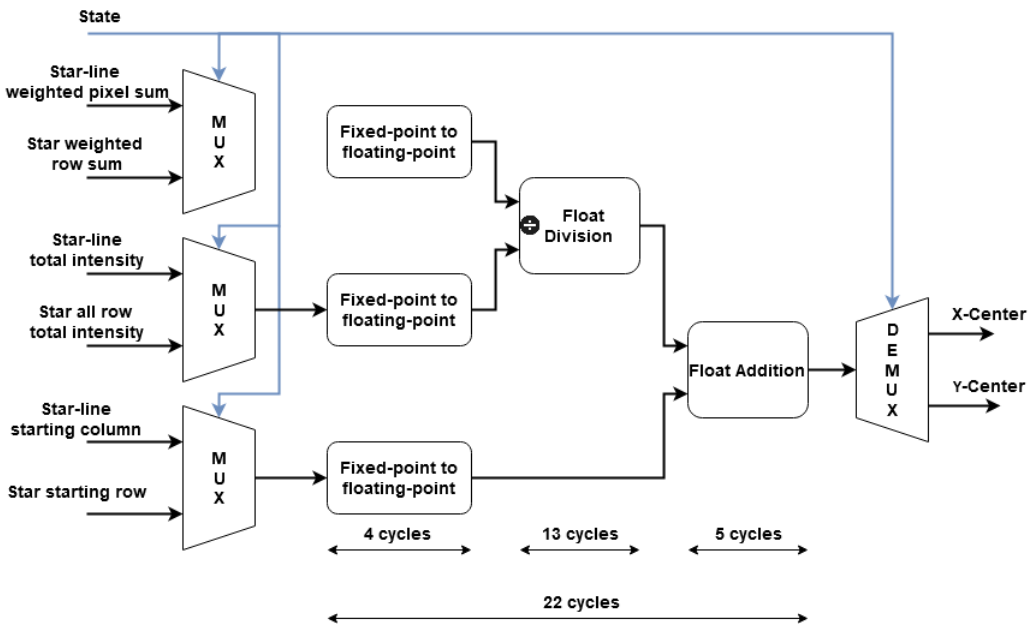


Figure 14. Block diagram of the hardware component responsible for replicating equations 4 and 6.

7.2.3 Processing of Intermediate Star Elements

The process described in this section is responsible for forming and keeping track of intermediate star elements that span multiple rows. Another FIFO memory component, previously mentioned as FIFO-B, lies at the heart of this process. Intermediate star element processing lies in the 50 MHz clock domain, which differs from the process responsible for the star-line element processing clocked from the CMOS sensor 27 MHz clock.

FIFO-B stores information about intermediate star elements that are iteratively modified until all star-line elements of the same potential star have been processed. Newly formed elements from the FIFO-A star-line elements are compared to the existing intermediate star elements of FIFO-B. Based on the differences between the newly formed and existing intermediate star elements, the state of the FIFO memory components, and the image transfer state, the process must handle different conditions. The overall flow diagram of this process and its different operations can be seen in Appendix I.

The intermediate star element data composite stored in the FIFO-B comprises information that characterizes a potential star in the image. Information stored in this composite eventually provides data required to calculate the final centroid X and Y coordinates through equations 5 and 6. The contents of this data composite are visualized in Figure 15.

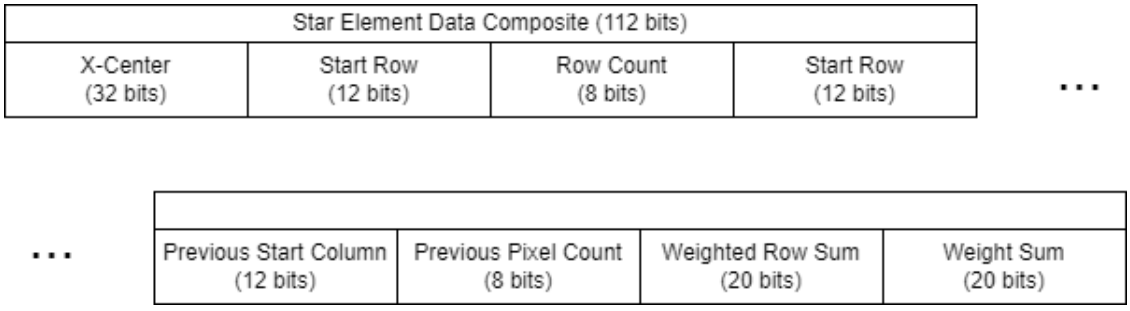


Figure 15. Star element data composite stored inside the FIFO-B memory component. This composite is 112 bits wide and consists of 8 different values characterizing a potential star.

FIFO-A not being empty triggers this process to transfer from idle to active state. When a star-line element is available in the FIFO-A, but FIFO-B is empty, a new intermediate star element is initialized with a calculated X_{center} value and other characteristics obtained from the FIFO-A star-line element.

When FIFO-B is not empty, the process compares the active row of the FIFO-A star-line element and the last row of the intermediate star element present at the FIFO-B output. The last row is obtained by summing up the starting row and the total number of rows of the intermediate star element. There are three different comparison outcomes, the rows match, the difference between the rows is equal to one, and the difference between rows is greater than one. Based on these outcomes, the process takes different paths and performs different procedures, also visualized in Appendix I.

In principle, the latest row in the intermediate star element can't match the row of the star-line element since two star-line elements residing in the same row can't be a part of the same star.

So if the rows match, the intermediate star element is irrelevant at that moment, but it must be re-queued in FIFO-B because it might be relevant for the following star-line elements. If all of the elements in FIFO-B have not been checked, a new intermediate star element is requested, and comparison occurs again. If all elements in FIFO-B have been compared to the star-line element, then a new intermediate star element data composite must be formed and stored in FIFO-B, where the active star-line element will set the base parameters.

When the difference between the rows is equal to one, the star-line element resides in the next row compared to the last row of the intermediate star element. This gives the star-line element the potential to be a part of the same star that the intermediate star element is forming. For a star-line element to merge with the intermediate star element, the distance between the starting or ending columns must remain below a threshold to ensure that the star-line is close enough to be merged. If the threshold is exceeded, the star-line element is too far away from the intermediate star element to be a part of the same potential star, and it is re-queued in FIFO-B. If all the intermediate star elements in FIFO-B have not been checked, a new element is requested, and a comparison occurs again. Otherwise, a new intermediate star element is formed based on the star-line element and queued in the FIFO-B. If the distance is below the threshold, then the star-line element is a part of the same star, and the process can move to update the accessed FIFO-B intermediate star element with new values and queue this new element in FIFO-B.

The last condition between the star-line and intermediate star element rows is when the difference is greater than one. It indicates that there will be no further star-line elements related to the accessed intermediate star element, and it can be released. Once the element has been released, the comparison continues with other intermediate star elements in FIFO-B. Similar to other cases, if all elements have already been checked, a new intermediate star element is formed based on the star-line and queued in FIFO-B. A more detailed description of the intermediate star element release procedure is explained in section 7.2.4.

There is a special condition that is not included in Appendix I. This condition occurs at the end of the process when all of the image pixels have been streamed out of the sensor, and a signal is supplied to the process indicating that this state is reached. When this signal indicates the end of an image, and the FIFO-A becomes empty, a different procedure is executed, and the remaining star elements in FIFO-B are released according to the procedure described in the following section 7.2.4. With the last element released from the FIFO-B, the centroid detection component will remain idle, and a reset signal must be invoked to successfully perform another centroid detection procedure.

7.2.4 Star Element Release and Centroid Storage

When all of the star-line elements of a potential star have been processed and merged, the intermediate star element is completed. Final calculations can be performed, and the centroid coordinates with the total intensity are stored in memory for further access by other components.

The final calculations include implementing the equations 5 and 6. The Y_{center} position is obtained with the component described in section 7.2.2. switching the input signals through multiplexing. The final X_{center} position is a simple averaging function. It includes a simple division of the previously accumulated X_{center} values of individual star-line elements and a division of rows in the intermediate star element.

A dual-port, dual-clock RAM is inferred as the memory component responsible for coordinate and centroid intensity storage. One side of the RAM is used for writing centroids, which is used by the centroid detection component, and a 50 MHz clock signal clocks it. The other side is clocked by 100 MHz, and it is meant only for reading procedures and is accessible by other FPGA components. X coordinate, Y coordinate, and the intensity are stored as a single data element in the RAM. The current configuration of the RAM component allows 255 centroids. However, this configuration can be easily changed by changing the generic parameters of the RAM component and making changes in the centroid detection algorithm that correspond to the maximum allowable centroids. Once all centroids have been processed and stored in RAM, the centroid detection component outputs a signal to other components indicating that it has finished its tasks.

8 Results

8.1 Consumption of FPGA resources

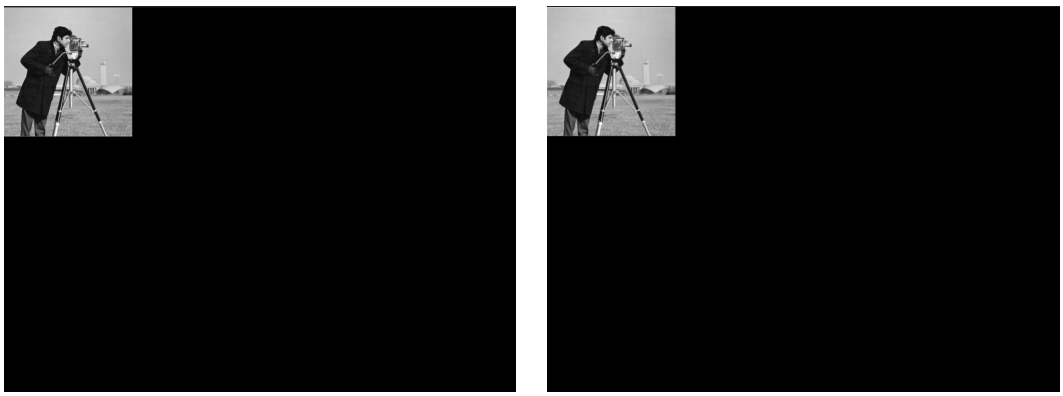
This section describes the final hardware resource usage by the designed components discussed in the previous sections. The latest configuration of the FPGA uses 31% (6913 of 22320) of the available logic elements. The internal FPGA RAM uses approximately 22% (134578 of 608256 bits). One phase-locked loop (PLL) is used to convert 50 MHz to 100 MHz. In the parts where the design uses 27 MHz, the frequency is sourced from and synchronized with the CMOS sensor. In the final design of the FPGA uses 30% (39 of 132) embedded multiplier elements.

8.2 Performance of the Commnad Center Component

The command center component currently implements six different commands that the user can send to initiate some process, write, or read data. These commands can then be used to evaluate the performance of other components.

By invoking command 0x01, the user can upload images with the maximum size of 1000×1500 pixels. The image upload takes approximately 2 minutes because each pixel transmission is associated with 8 bytes as described in Section 7.1.1 and visualized in Table 3.

Correct storage of the uploaded test image can be verified with command 0x02, which initiates the test image extraction through the UART peripheral. It takes approximately 15 seconds to download a 1000×1500 image with UART at 1 MBaud/s. Figure 16(a). visualizes the image sent to the FPGA with series of 0x01 commands. Figure 16(b) visualizes the image that was obtained with the 0x02 command. The original image of a cameraman is slightly modified to include additional padding with black pixels to obtain the required 1000×1500 pixel test image. The sent and received images were verified to be identical, indicating proper storage.



(a) Test image sent to the FPGA through series of 0x01 commands

(b) Test image received from the FPGA invoking 0x02 command

Figure 16. Test images sent to and received from the FPGA. 256x256 pixel image padded with black pixels to obtain a 1000×1500 pixel test image.

To verify commands 0x03 and 0x07, a test image containing stars was uploaded with command 0x01. Centroid detection was applied with command 0x03; then, centroid information was extracted from the FPGA through UART peripheral with command 0x07. Figures 19. and 20. were used to verify these commands.

Command 0x04 was used to capture an image from the CMOS image sensor, and then command 0x05 was used to extract the captured image out of the FPGA through the UART peripheral. The extraction of the 1944x2952 pixel CMOS image sensor image takes approximately 1 minute if the UART baud rate is set to 1 MBaud/s. The captured image is presented in Figure 17.

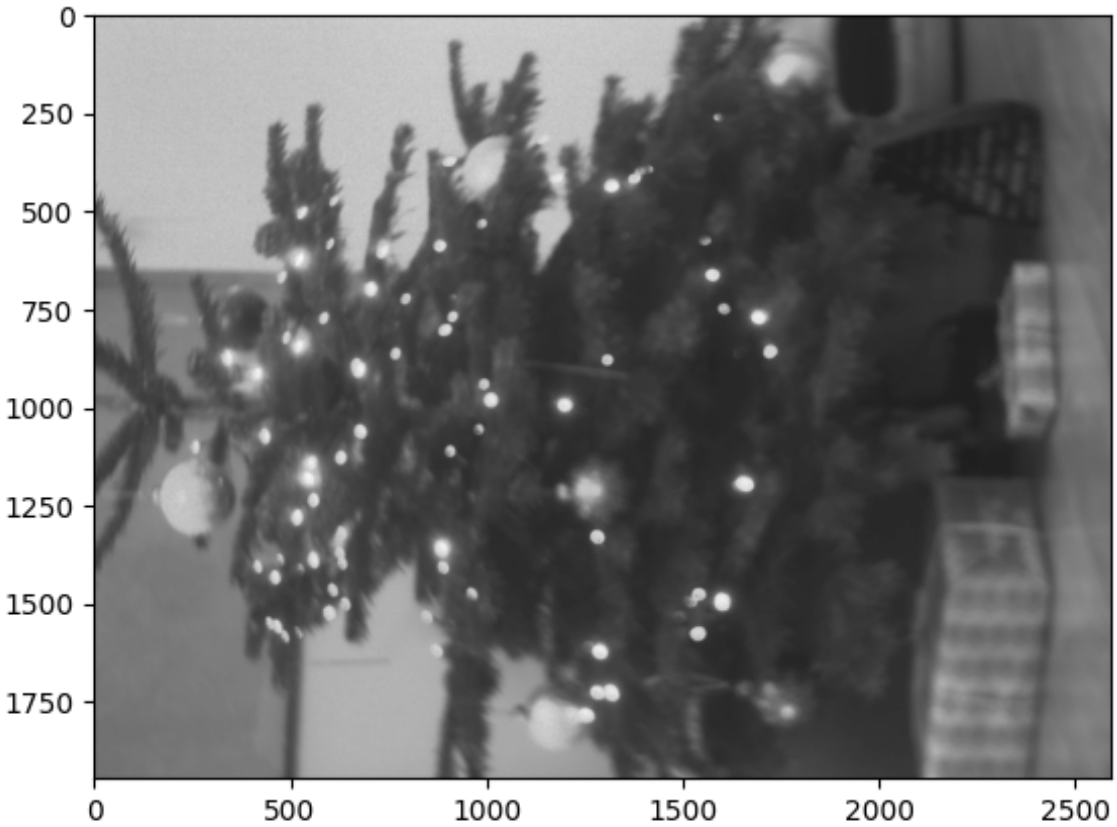


Figure 17. 1944×2952-pixel CMOS sensor image captured with 0x04 command and then transferred through UART with 0x05 command.

8.3 Centroid Detection Performance

The performance of the star detection algorithm was evaluated based on its capability of finding centroid coordinates in a simulation environment and on real FPGA as per the method described in section 5. A single test component was created that was placed both in the simulation testbench and in real-life FPGA, where in both cases, it received three different clock inputs: 27 MHz, 50 MHz, and 100 MHz. The test component's connection in FPGA and simulation can be seen in Figure 18. One output signal was used as a UART transmit signal to transfer the centroid data from the test component. The test component contained an embedded 10×20 pixel test image seen in Figure 19. passed through the centroid detection algorithm pixel by pixel like the CMOS sensor camera would. When all pixels have been streamed, and the centroid detection has also been finished, the centroid data is transmitted over UART. This image was manually constructed with arbitrary, pre-defined star elements and pixel values. It only serves the purpose of verification and may not fully represent the capabilities or downfalls of the centroid detection algorithm.

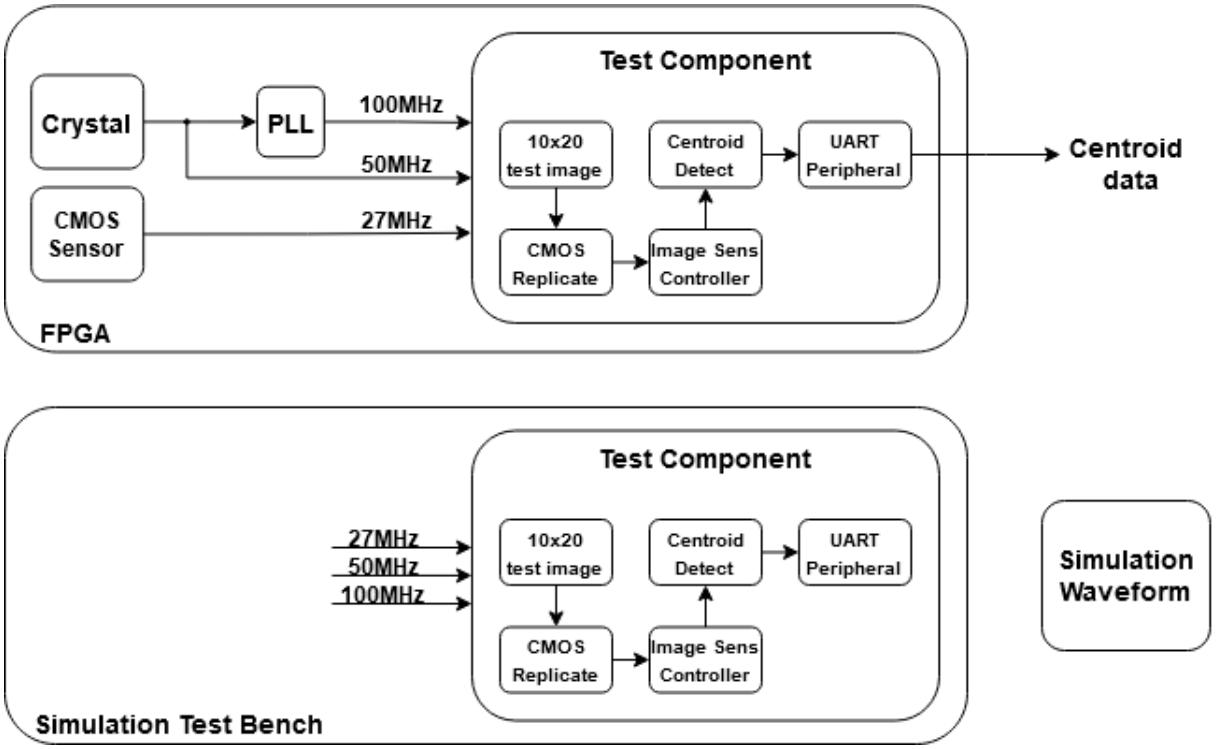


Figure 18. Simplified diagram of the setup used for verify the performance of the centroid detection algorithm based on a predefined 10×20 pixel image

The expected results were calculated by hand using equations 4, 5, and 6. In this case, the algorithm's threshold and ghosting values were also set to arbitrary values - four and three, respectively. These values were chosen based on the evaluation of the pre-defined test image seen in Figure 19. and would not serve any purpose in the final FPGA design. The pixel threshold level should be obtained through empirical or probabilistic methods as described in section 2.3. The final ghosting level could be based on the quality of the image obtained in the star tracker's final operation; it may require updates based on the damage the CMOS image sensor may experience during the mission period. The expected, simulation, and FPGA centroid coordinates for the pre-defined test image can be seen in Table 4.

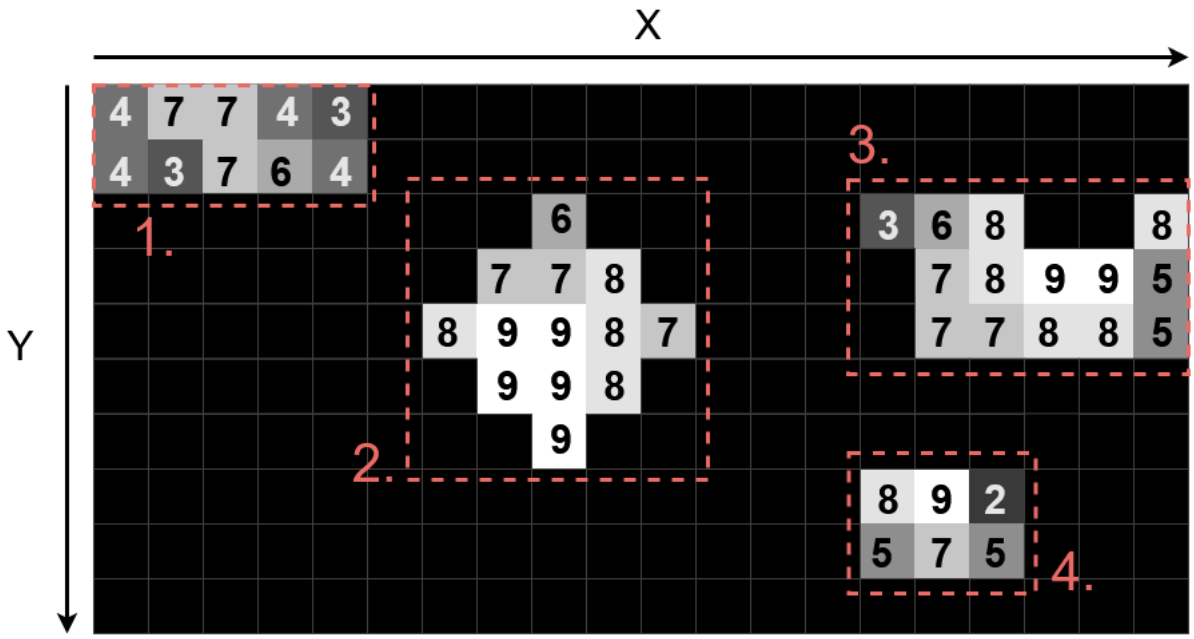


Figure 19. Manually generated test image containing star imitations. This image was used to verify that the output of centroid detection component in simulation and FPGA match the expected values

Table 4. Table summarizing the centroid positions obtained from the Figure 19 by manual calculations, the simulation and the FPGA.

Centroid Nr.	Expected Coordinate		Simulation Coordinate		FPGA Coordinate	
	X	Y	X	Y	X	Y
1	1.981	0.4815	1.981	0.4815	1.981	0.4815
2	7.987	4.067	7.987	4.096	7.987	4.096
3	16.88	3.137	16.88	3.137	16.88	3.137
4	14.76	7.500	14.76	7.500	14.76	7.500

Afterward, centroid detection was performed on a real image through the command center component. There were attempts to capture images of stars with the real image sensor, but there were issues with CMOS image sensor configuration and exposure time that denied acquiring an image with visible stars. Thus, the centroid detection component was supplied with test images

obtained with a CMOS sensor and optics similar to the ones used in the final ESTCube-2 star tracker hardware. The used image of the sky containing stars can be seen in Figure 20. As the original image size was larger than the supported test image size, only the first 1000 rows and 1500 were used. The used 1000×1500 pixel area is also pointed out in Figure 20. by the red rectangle. The centroid coordinate values obtained with the centroid detection component were compared with the centroid results obtained by the "DAOStarFinder" function, which is a part of the photutils.detection module in Python.

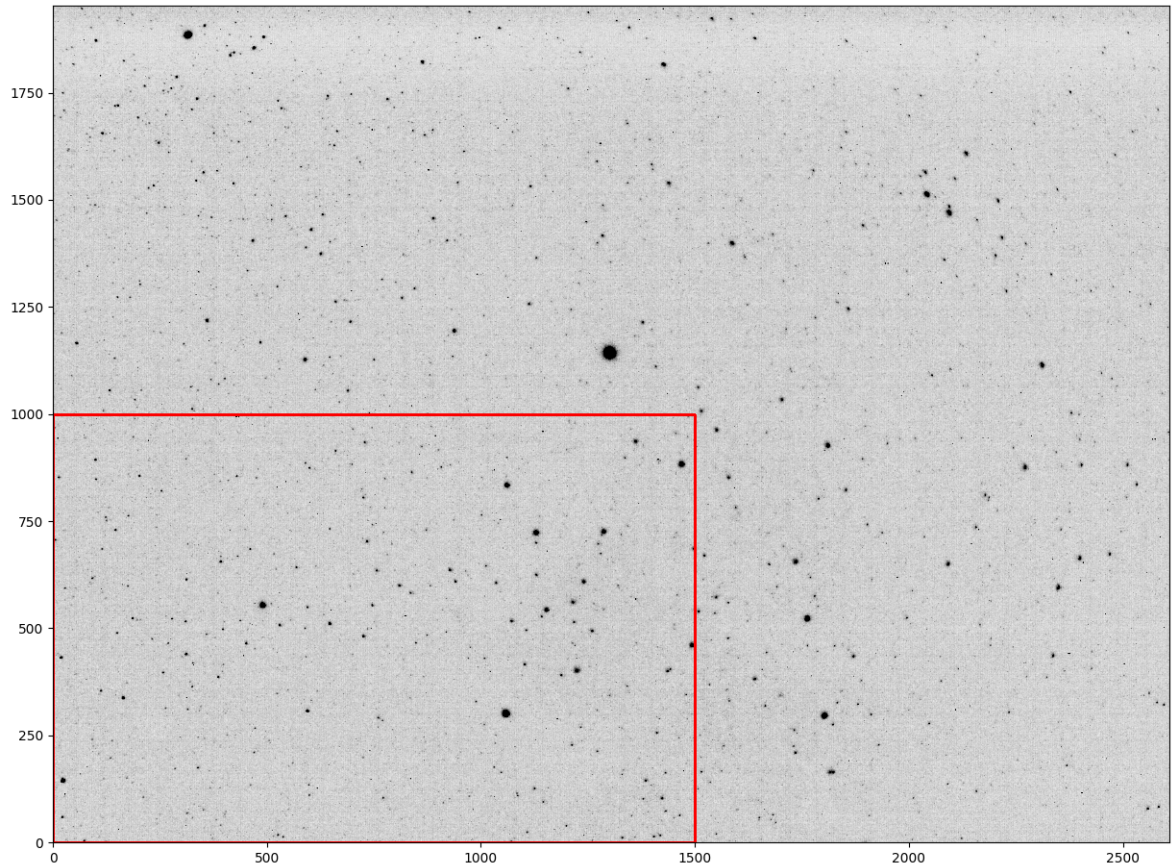


Figure 20. Original image of the night sky with stars obtained by ESTCube-2 star tracker CMOS sensor and optics similar to the one used in satellite flight hardware. The red border indicates the 1000x1500 pixel area being cropped for further usage with the centroid detection component.

Because FPGA implementation for adaptive thresholding was not successfully achieved in this thesis, a Python script was used to calculate the pixel intensity mean and standard deviation values of the image seen in Figure 20. The obtained standard deviation value was 34.6 analog-to-digital units (ADU). The threshold value for the centroid detection component was initially set to a pixel intensity value over the sum between the mean and five standard deviations, according to Ayal's solution [21]. For the image seen in Figure 20, this resulted in 173 ADU. However, the centroid detection component could not accurately differentiate stars with this threshold level.

After an unsuccessful attempt, the threshold value for the centroid detection component was raised to ten standard deviations over the mean - 346 ADU. The resulting coordinates can be seen in Figure 21., where the blue circles mark centroids found by the "DAOStarFinder" function, and the red circles mark centroids found by the designed FPGA centroid detection

component. The "DAOStarFinder" function found 309 stars; the designed centroid detection component found 148 centroids. Out of the 148 centroids, 128 centroids were within 10 pixels of a centroid found by the "DAOStarFinder" function. The mean X coordinate difference between these matching centroids obtained by the "DAOStarFinder" function and the centroid detection component was 0.520 pixels, and 0.641 pixels for the Y coordinate. Out of 128 cases, there were 14 cases where the distance between coordinates found by the "DAOStarFinder" and the centroid detection component was more than one pixel.

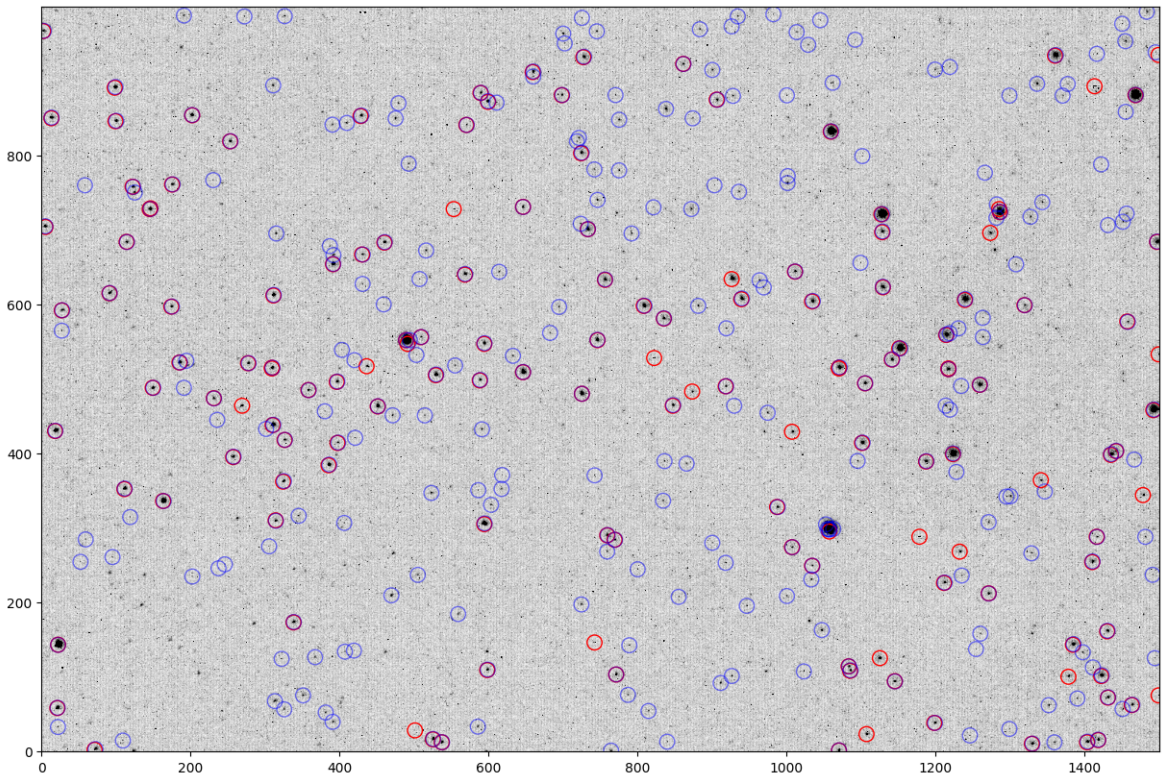


Figure 21. Image of the night sky. The red circles mark the stars detected by the designed FPGA centroid detection component; the blue circles mark the stars detected by the "DAOStarFinder" function in Python. The threshold for the centroid detection was set to 346 ADU.

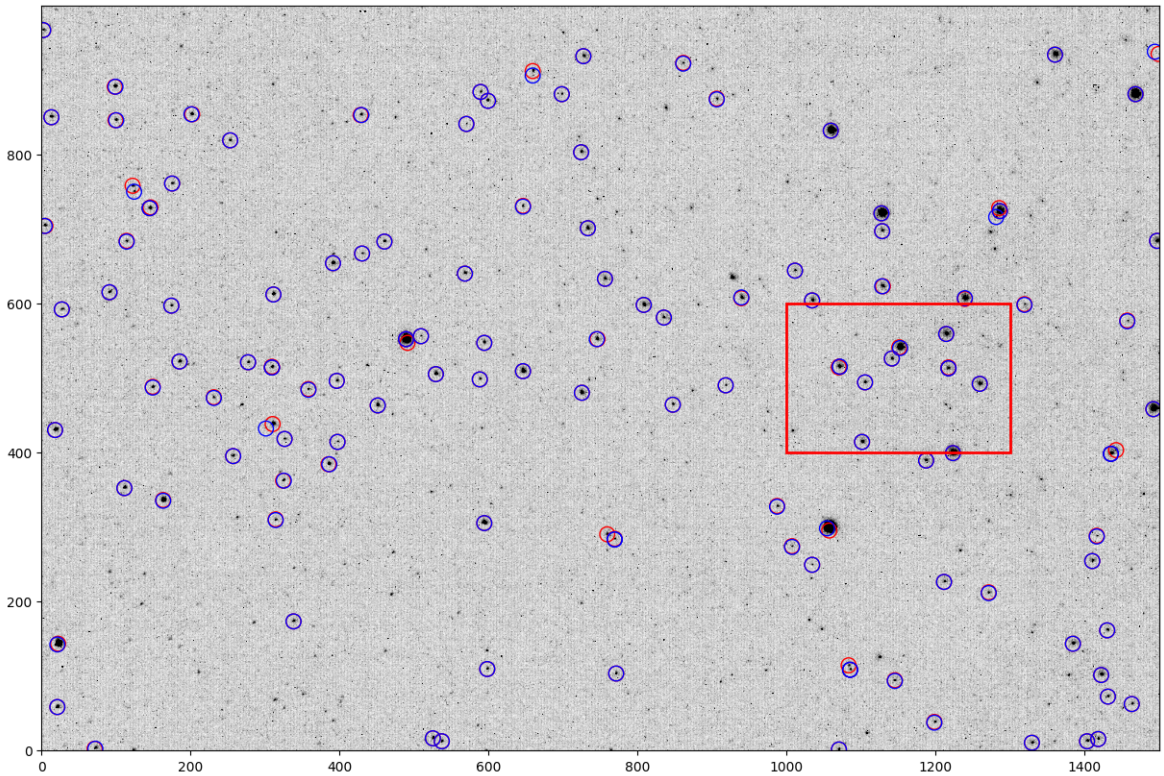


Figure 22. Image of the night sky, where stars detected by the "DAOStarFinder" function and the centroid detection component are within 10 pixels of each other and are marked with blue and red circles, respectively. The threshold for the centroid detection was set to 346 ADU.

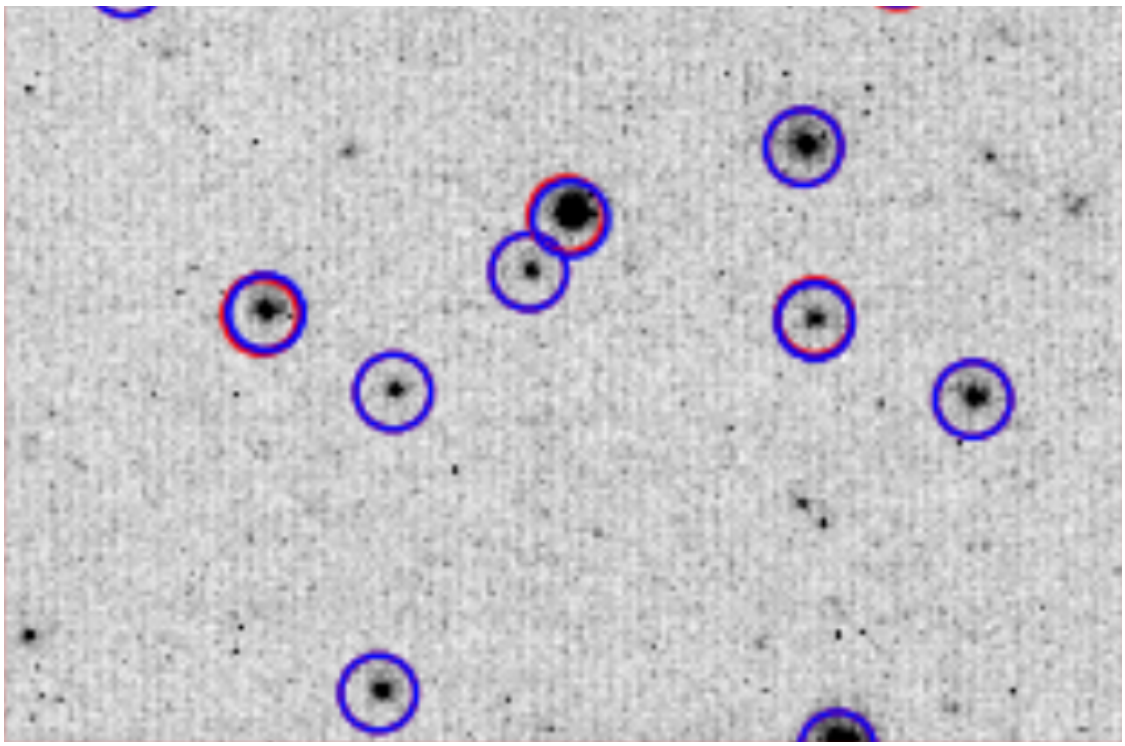


Figure 23. Zoomed-in area that is pointed out with a red rectangle in Figure 22., where multiple stars, found by both the "DAOStarFinder" function and centroid detection component, overlap.

8.4 Performance of Adaptive Thresholding Method

This section attempts to integrate and test Ayal's proposed adaptive thresholding method. [21] In Ayal's thesis, the VHDL source code of functions for division and square root integer operations was provided. In this thesis, source code was directly imported, and an additional test component was made to test their performance. The arithmetic function source code is hard-coded to handle 16-bit wide data, but the author slightly adjusted it to support any supplied data width. This test component was made according to the methodology described in section 5.

The only input for the test component is a 100 MHz clock signal, and the output of this component is the UART transmit signal. This component iterates through a varying integer signal from 1 to 255. A constant 255 numerator is used for the division function, and the varying integer signal is used as a denominator. The input for the square root function is the varying integer signal.

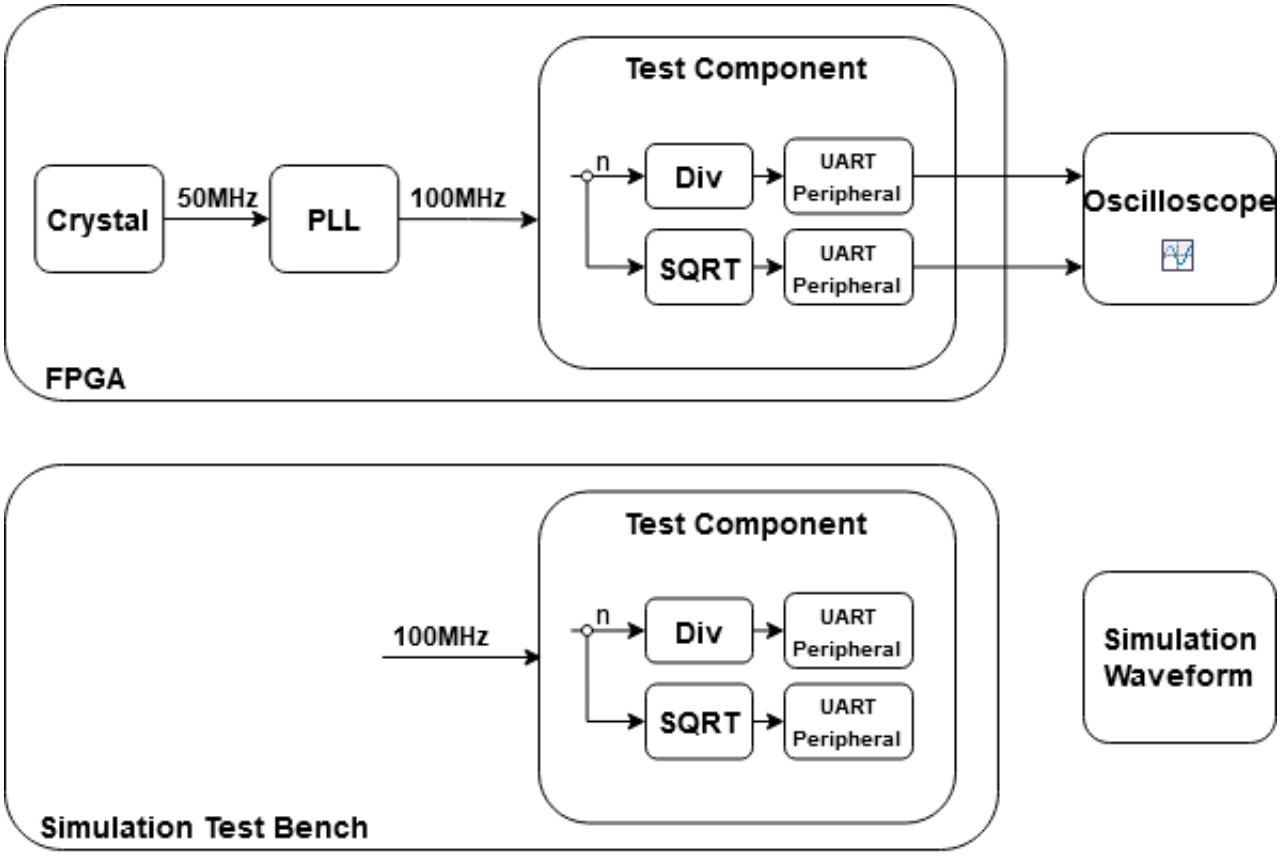
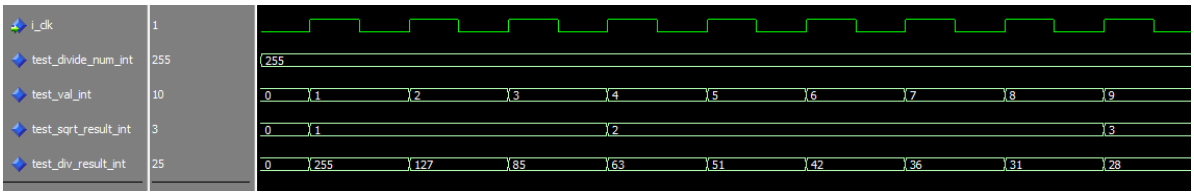
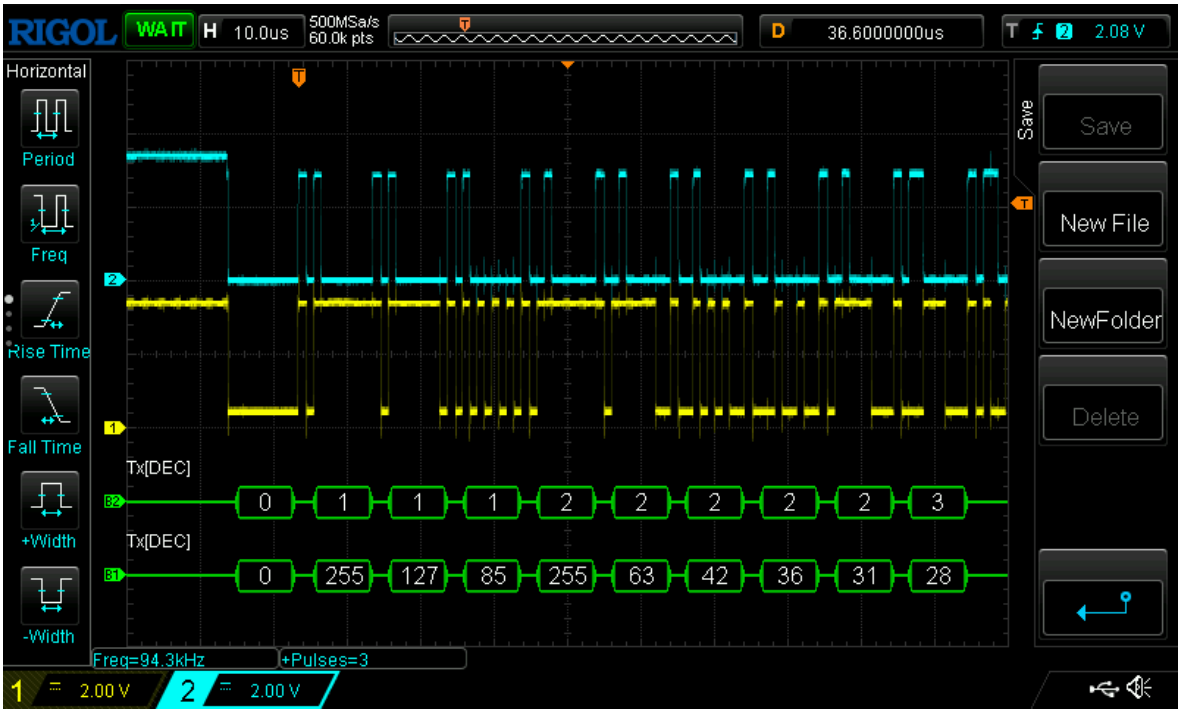


Figure 24. Block diagram of the test setup used to evaluate Ayal's solution to the integer division and square-root functions.

In simulation, the output of both functions acted as expected and provided correct results. However, the real-life results of the division function did not match the same UART TX output pattern obtained in the simulation, though the majority of values appeared correct. Multiple, but not all, simulation and real-life measurement results can be seen in Figure 25. The author believes the inconsistent real-life results are caused by using only combinational logic and a large amount of it (both arithmetic functions use approximately 1000 logic elements for 16-bit wide data) in the arithmetic functions and then supplying the inputs and registering the outputs with signals clocked by 100 MHz. This is also indicated by the Intel® Quartus® Prime Lite Edition timing report, where the maximum setup time[5] violation was 39 ns.



(a) Simulation results of integer division and square root functions. The top row is the 100 MHz clock signal, the second is the division numerator, the third is the division denominator and square root input, the fourth is the square root result, and the fifth is the division result.

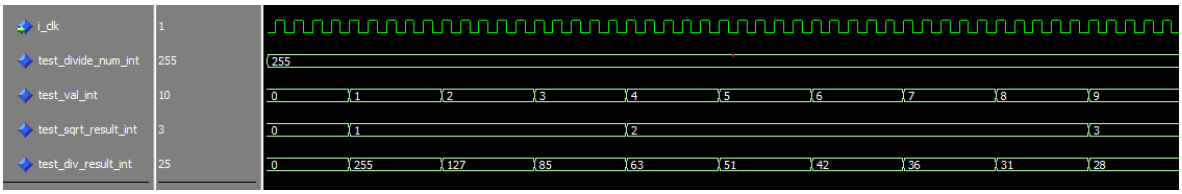


(b) FPGA test results measured with an oscilloscope. The blue channel shows the RS232-coded square-root result, and the yellow channel shows the RS232-coded division result. The top decimal row presents the decoded decimal value of the square root function. The bottom decimal row presents the decoded decimal value of the division function.

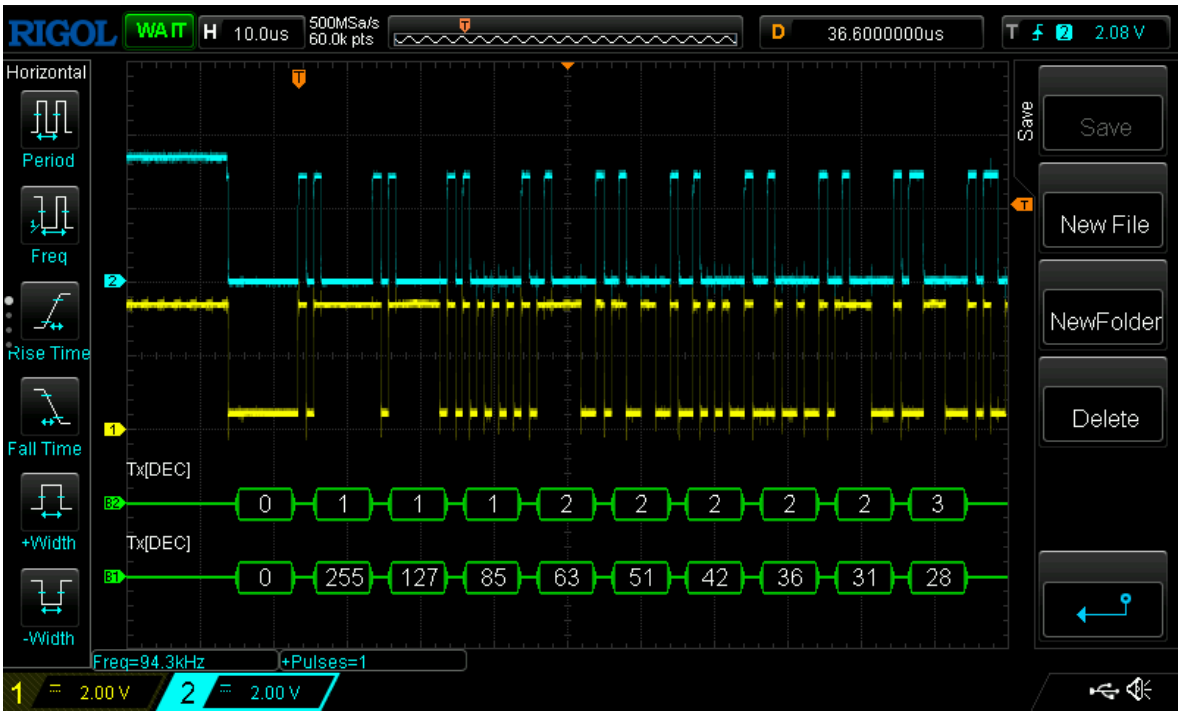
Figure 25. Inconsistent results between simulation and FPGA tests when performing integer division and square root functions. The division denominator and the square root input is incremented from 1 to 255. The figure does not visualize all results.

[5]The input signal must be stable for a certain amount of time before the latching event.

A proposed solution could be decreasing the clock frequency to provide more time for the signals to travel through combinational logic before registering the output of the arithmetic functions or increasing the time the input signals are stable. Additional frequency sources should not be added to the existing design to avoid further complexity and possible clock domain crossing issues. Because the timing violation was 39 ns and the period of the 100 MHz clock signal was 10 ns, five clock cycles were added to hold inputs to the arithmetic functions. In the FPGA, frequencies higher than 100 MHz are not used. This leads to results shown in Figure 26, where simulation and real-life results match.



(a) Simulation results of integer division and square root functions when introducing additional five cycle delay. The top row is the 100MHz clock signal, the second is the division numerator, the third is the division denominator and square root input, the fourth is the square root result, and the fifth is the division result.



(b) FPGA test results measured with an oscilloscope when introducing a five-cycle delay. The blue channel shows the RS232-coded square root result, and the yellow channel shows the RS232-coded division result. The top decimal row presents the decoded decimal value of the square root function. The bottom decimal row presents the decoded decimal value of the division function.

Figure 26. Consistent results between simulation and FPGA tests when performing integer division and square root functions when introducing a five-cycle delay. The division denominator and the square root input is incremented from 1 to 255. Figures do not visualize all results.

To successfully implement the adaptive thresholding, the image's pixel intensity standard deviation must be obtained following equations 2 and 3. However, the further usage of the provided functions causes concerns due to the possibility of significantly increasing resource consumption.

If the CMOS image size is 1944×2952 pixels and the maximum pixel value is 4095, then during the mean calculation, the image pixels must be continuously summed and divided by the number of pixels in the image. Although many high-intensity pixels are not expected, they can still be present and may lead to large values. For example, the star image in Figure 21 has 1000×1500 pixels (considerably lower than the original CMOS image) but already requires data widths of 29 bits to sum all of the image pixels. This issue becomes even more apparent in standard deviation calculation, where the difference between the mean and individual pixels is squared and then continuously summed, leading to a possibility of very large numbers before the division and root functions.

Due to these reasons, the author did not explore further solutions for adaptive thresholding, though there might be some optimizations or simplifications that would make this solution more efficient and viable to use in the final design.

9 Conclusion

This thesis aimed to develop and implement an FPGA design for the ESTCube-2 star tracker flight electronics that were created by the author prior to this thesis. The implemented FPGA design was expected to offer the possibility to interface with different devices and perform higher-level star tracker functions such as image acquisition/storage, star detection, star identification, and/or attitude determination.

Throughout this thesis, an emphasis has been placed on the seemingly simple interface between the designed FPGA components and the interface between external devices. The core idea behind this approach was to standardize the interface between components responsible for different peripherals and the designed internal FPGA components. Different abstractions of FPGA components were made in the FPGA design implementation leading to a standardized interface between different components and peripherals. This alleviated the complexity of individual components and made the processes more localized and less affected by events or states occurring in different parts of the FPGA device. This approach may simplify FPGA development efforts further by laying a solid foundation.

A command center component was implemented that allows users to interface with the FPGA, give commands, write and read data, and make testing easier. The command center implementation is simple and can be easily extended with new commands while not affecting the nature of the old commands. The current command center solution might not be helpful in the final autonomous star tracker version, but it can be modified to cycle through the described command processes based on factors other than commands.

A star detection algorithm was implemented for the ESTCube-2 star tracker system that can detect potential star coordinates in real-time as the data is streamed from the image sensor. This component involves single-precision floating-point calculations and outputs sub-pixel coordinates through involving Intel floating-point arithmetic IP codes. The algorithm was initially tested with a predefined image supplied to the simulation and the actual FPGA for algorithm verification purposes. In this case, the simulation and the FPGA provided similar results and aligned with the expected coordinate values. Afterward, the centroid detection algorithm was tested with an image of the sky obtained by a CMOS sensor and optics similar to the ones used in the ESTCube-2 star tracker system flight hardware. Unfortunately, acquiring an image of stars from the CMOS image sensor connected to the ESTCube-2 star tracker flight model electronics was unsuccessful.

Implementation of the proposed adaptive thresholding method in VHDL was unsuccessfully attempted. Possible reasons for this lack of success were theorized, and possible solutions and optimizations that could lead to positive results in future work were provided.

Currently, the ESTCube-2 star tracker FPGA can perform the minimum requirements of the star tracking procedure, including image acquisition, image storage, centroid detection, and transfer of the coordinates/images to the MCU. Future work would require implementing geometric parameter extraction, star identification, and star tracking functionality on the ESTCube-2 star tracker flight electronics with the available resources of the FPGA or the MCU.

References

[1] Onsemi, "MT9P031 - 1/2.5-Inch 5 Mp CMOS Digital Image Sensor." https://www.onsemi.com/pdf/datasheet/mt9p031-d.pdf, 2021.

[2] D. Turcu and G. A. Stan, "Purpose of using cubesat satellite technologies in the military domain," *Proceedings of the 17th International Scientific Conference Strategies XXI*, vol. 17, no. 1, 2021.

[3] S. Cole, "Small satellites increasingly tapping COTS components." Military Embedded Systems, June 2015. https://militaryembedded.com/comms/satellites/small-tapping-cots-components.

[4] S. Caldwell, "What are SmallSats and CubeSats?." https://www.nasa.gov/what-are-smallsats-and-cubesats/.

[5] S. Stoyanova, "Star trackers lights the way." NASA Podcasts, September 2008. https://www.nasa.gov/multimedia/podcasting/StarTrackers.html.

[6] G. V. M. E. Team, "Gemini program mission report," tech. rep., NASA, January 1966.

[7] B. Dunbar, "Project Gemini: Apollo's training ground," October 2008. https://www.nasa.gov/centers/kennedy/about/history/gemini.html.

[8] P. Ceruzzi, "Deep space navigation: The Apollo VIII mission," *Quest*, vol. 17, no. 4, pp. 8–17, 2010.

[9] R. E. Wilson, Jr., "Apollo Experience Report - Guidance and Control Systems," tech. rep., NASA, June 1976.

[10] D. G. Hoag, "Apollo Navigation, Guidance, and Control Systems," tech. rep., NASA, April 1969.

[11] C. C. Liebe, "Star trackers for attitude determination," *IEEE Aerospace and Electronic Systems Magazine*, vol. 10, no. 6, pp. 10–16, 1995.

[12] P. S. Jorgensen, J. L. Jorgensen, and T. Denver, "MicroASC a miniature startracker," *Proceedings of The 4S Symposium: Small Satellites, Systems and Services*, 2004.

[13] J. L. Jorgensen, C. C. Liebe, A. R. Eisenman, and G. B. Jensen, "The advanced stellar compass onboard the Oersted satellite," *Spacecraft Guidance, Navigation and Control Systems, Proceedings of the 3rd ESA International Conference*, 1997.

[14] J. L. Jorgensen, "In-orbit performance of a fully autonomous star tracker," *Spacecraft Guidance, Navigation and Control Systems, Proceedings of the 4th ESA International Conference*, 2000.

[15] C. C. Liebe, "Accuracy performance of star trackers–a tutorial," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 38, no. 2, pp. 587–599, 2002.

[16] Vectronic-Aerospace, "Star trackers VST-68M, VST-41M," 2023. https://www.vectronic-aerospace.com/star-trackers/.

[17] Terma, "T3 star tracker," 2023. https://www.terma.com/products/space/star-trackers/.

[18] RedWire, "SpectraTRAC star tracker," 2023. https://redwirespace.com/products/spectratrac/.

[19] F. L. Markley and J. L. Crassidis, *Fundamentals of Spacecraft Attitude Determination and Control*. Microcosm Press and Springer, 2014.

[20] S. B. Howell, *Handbook of CCD Astronomy*, vol. 5 of *Cambridge Observing Handbooks for Research Astronomers*. Cambridge University Press, 2 ed., 2006.

[21] A. R. Ayal, "Star detection algorithm for ESTCube-2 star tracker," 2016. Available at https://dspace.ut.ee/items/b6944b6f-141d-4946-94c7-1e407cf94f48.

[22] M. Lindh, "Development and Implementation of Star Tracker Electronics," 2014. Available at https://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-148955.

[23] J. Laks, "Developing hardware for the ESTCube-2 star tracker," 2019. Available at https://dspace.ut.ee/items/4afadfd1-a7ff-4d71-8220-d963cc2b2f50.

[24] S. Schumann, "Using star identification algorithms on ESTCube-2 star tracker," 2019. Available at https://dspace.ut.ee/items/b4f02328-cabb-46d1-8128-e1b508c92c2a.

[25] Altera, "Cyclone IV Device Handbook." https://www.intel.com/content/www/us/en/docs/programmable/767845/current/cyclone-iv-featured-documentation-quick.html, 2016.

[26] Integrated Silicon Solution Inc, "IS42/45R86400D/16320D/32160D IS42/45S86400D/16320D/32160D." https://www.issi.com/WW/pdf/42-45R-S_86400D-16320D-32160D.pdf, 2015.

[27] Micron Technology Inc., "MT25QL512ABB - Micron Serial NOR Flash Memory." https://media-www.micron.com/-/media/client/global/documents/products/data-sheet/nor-flash/serial-nor/mt25q/die-rev-b/mt25q_qlkt_l_512_abb_0.pdf, 2013.

[28] Intel, "FIFO Intel® FPGA IP User Guide." https://www.intel.com/content/www/us/en/docs/programmable/683522/18-0/user-guide.html, 2018.

[29] E. Peňa and M. G. Legaspi, "UART: a hardware communication protocol understanding universal asynchronous receiver/transmitter," *Analog Dialogue*, vol. 54, no. 4, 2020.

[30] Intel, "Floating-Point IP Cores User Guide." `https://www.intel.com/content/www/us/en/docs/programmable/683750/23-1/about-floating-point-ip-cores.html`, 2023.

Appendix

I Flow Charts

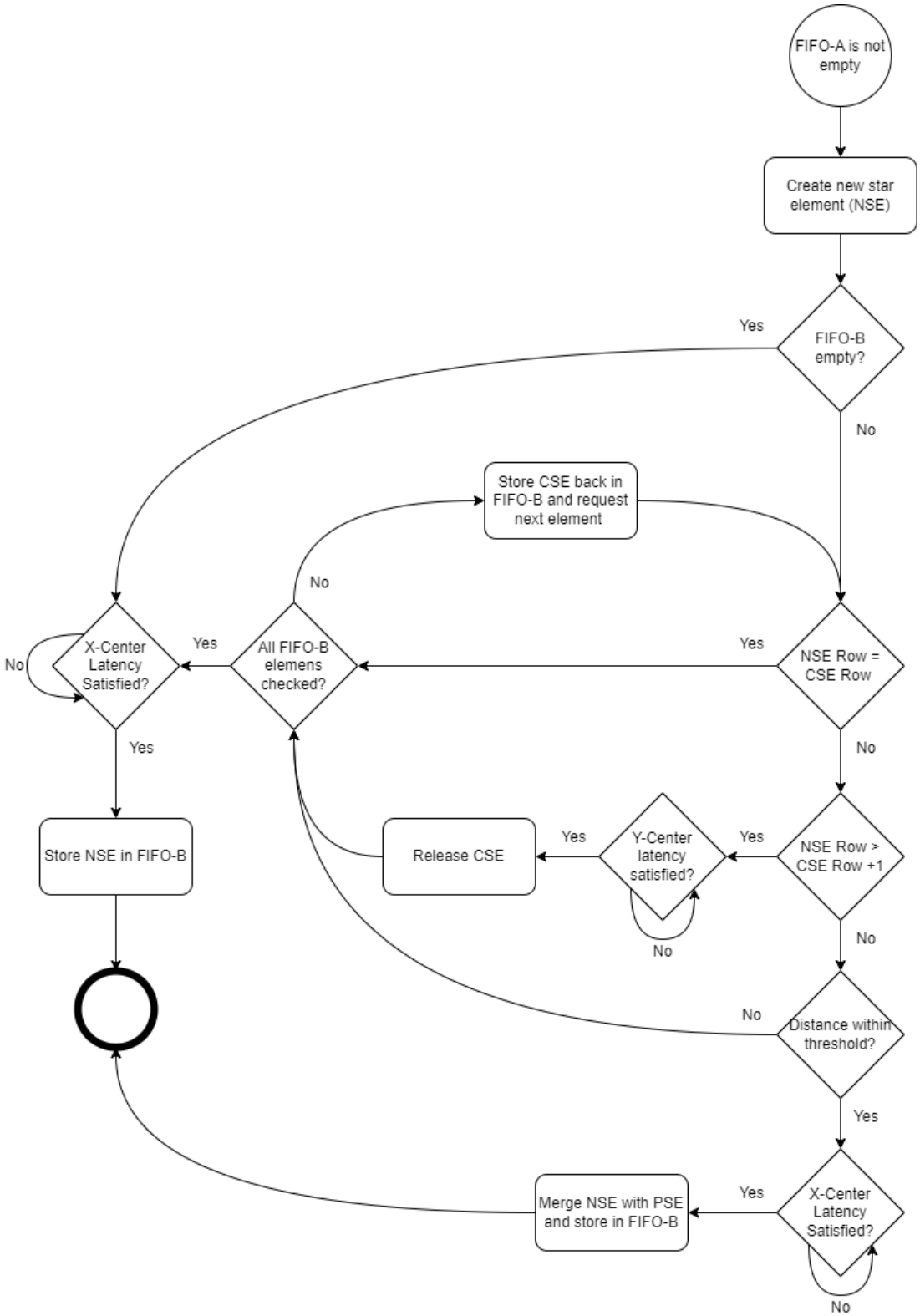


Figure 27. Flow chart of the process described in the section 7.2.3. NSE corresponds to ”New Star Element”, and CSE corresponds to ”Current Star Element” available at the output of FIFO-B memory component

II Licence

Non-exclusive licence to reproduce thesis and make thesis public

I, **Roberts Oskars Komarovskis**,

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to

   reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,

   Development and Implementation of ESTCube-2 Star Tracker FPGA Design,

   supervised by Kristo Allaje and Tõnis Eenmäe.

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.

4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Roberts Oskars Komarovskis
23/12/2023