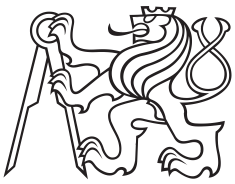
Master Thesis



Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Cybernetics

Implementation of Image Processing in the UVDAR System on FPGA

Bc. Vojtěch Vrba

Supervisor: Ing. Viktor Walter
Field of study: Cybernetics and Robotics
Subfield: Cybernetics and Robotics
January 2023

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Vrba Vojtěch** Personal ID number: **474409**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Cybernetics**

Study program: **Cybernetics and Robotics**

Branch of study: **Cybernetics and Robotics**

II. Master’s thesis details

Master’s thesis title in English:

Implementation of Image Processing in the UVDAR System on FPGA

Master’s thesis title in Czech:

Implementace zpracování obrazu v systému UVDAR na FPGA

Guidelines:

The MRS group uses on its robotic platforms designed for communication-free cooperation the visual relative localization system UVDAR. Since the current implementation of the system is divided into numerous discrete components and its software elements use the processing power of onboard computers. Therefore, there is currently a need to at least in part integrate the system into a single device. Research the operational principles of the UVDAR system. Based on this research evaluate which elements of the system can be consolidated into a single device or efficiently implement on FPGA. Design and implement the selected system elements on a FPGA development board and subsequently perform analysis of parity with the current implementation and of the computational performance improvement brought by the new implementation.

Bibliography / sources:

[1] Walter, Viktor a Staub, Nicolas a Saska, Martin a Franchi, Antonio - „Mutual Localization of UAVs based on Blinking Ultraviolet Markers and 3D Time-Position Hough Transform“ – Munich, 2018.
[2] C ı z ek, Petr a Faigl, Jan - ”Real-Time FPGA-Based Detection of Speeded-Up Robust Features Using Separable Convolution“ – USA, 2018.
[3] Scaramuzza, Davide, et. al. - ”A Flexible Technique for Accurate Omnidirectional Camera Calibration and Structure from Motion” - New York, 2006.

Name and workplace of master’s thesis supervisor:

Ing. Viktor Walter Multi-robot Systems FEE

Name and workplace of second master’s thesis supervisor or consultant:

Date of master’s thesis assignment: **21.01.2022** Deadline for master's thesis submission: **10.01.2023**

Assignment valid until: **30.09.2023**

Ing. Viktor Walter	prof. Ing. Tomáš Svoboda, Ph.D.	prof. Mgr. Petr Páta, Ph.D.
Supervisor’s signature	Head of department’s signature	Dean’s signature

III. Assignment receipt

The student acknowledges that the master’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master’s thesis, the author must state the names of consultants and include a list of references.

. _____ Date of assignment receipt

_____ Student’s signature

© ČVUT v Praze, Design: ČVUT v Praze, VIC

Acknowledgements

I would like to thank my thesis supervisor Ing. Viktor Walter for all his guidance throughout this work and for his invaluable feedback. I am also thankful to doc. Ing. Jan Fischer, CSc., for providing the imaging sensors necessary for conduction of the experimental part of this work. Finally, I must express my very profound gratitude to my parents for their unfailing support and continuous encouragement during my years of study.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, 10 January 2023.

Signature: .....

Abstract

The thesis focuses on the design of a Field Programmable Gate Array (FPGA) implementation of image processing algorithms used in the UltraViolet Direction And Ranging (UVDAR) system, developed for mutual relative localisation of Unmanned Aerial Vehicles (UAVs). In general terms, the UVDAR system consists of ultraviolet LED markers producing narrow-band signals, a UV-sensitive camera and image processing algorithms implemented in the C++ language. At first, the original software implementations of the algorithms were analysed in order to assess their computational complexity and memory allocation requirements. The main focus was on the specialised variation of the Features from Accelerated Segment Test (FAST) algorithm used for detection of active LED markers in camera images and on the 4D Hough Transform used for retrieval of linearly approximated image trajectories of the detected markers. Based on the analysis results, a FPGA architecture implementing the FAST-like algorithm was designed, producing identical outputs as the original C++ implementation. A selection of a suitable FPGA development board followed, together with a hardware design of a compatible circuit board with integrated CMOS sensor. The development platform Terasic DE10-Nano containing Cyclone V System-on-Chip (SoC) was selected for conducting real FPGA experiments. The VHSIC Hardware Description Language (VHDL) was used for implementation of the proposed FPGA architecture. Additional circuitry for visualisation of output data was designed using VHDL as well, allowing an output through HDMI interface in the form of an annotated video stream. Lastly, the current results, as well as an optional implementation of the UVDAR system into an embedded device are discussed, with further proposals for future development on the SoC.

Keywords: UVDAR, UAVs, mutual relative localization, imaging sensor, bright spot detection, Hough Transform, 3D line fitting, FPGA

Supervisor: Ing. Viktor Walter

Abstrakt

Tato diplomová práce se zabývá návrhem implementace obrazových algoritmů, které využívá systém UVDAR (UltraViolet Direction And Ranging) vyvinutý pro vzájemnou relativní lokalizaci dron v rojích, na programovatelných hradlových polích (FPGA). Systém UVDAR je ve svém základu složen z ultrafialových (UV) LED diod produkujících signály v úzkém pásmu spektra, kamerového senzoru citlivého v UV spektru a obrazových algoritmů implementovaných v jazyce C++. Nejprve byly analyzovány původní softwarové implementace těchto algoritmů a byla vyhodnocena jejich výpočetní a paměťová náročnost, konkrétně u speciální variace algoritmu FAST (Features from Accelerated Segment Test) používané pro detekci aktivních jasových značek ve snímcích z kamery a u 4D Houghovy transformace používané pro nalezení lineární aproximace trajektorií detekovaných značek v obraze. Na základě výsledků analýzy byla navržena FPGA architektura pro implementaci variace FAST algoritmu, která produkovala shodné výstupy jako původní softwarová implementace. Následoval výběr vhodné vývojové desky s FPGA společně s návrhem hardwaru pro kompatibilní desku plošných spojů s CMOS kamerovým senzorem. Pro experimentální část práce byla vybrána vývojová deska Terasic DE10-Nano obsahující čip Cyclone V SoC (System-on-Chip). Poté byla implementována předložená FPGA architektura v jazyce VHDL (VHSIC Hardware Description Language). Ve VHDL byla implementována i dodatečná logika pro vizualizaci výstupních dat v reálném čase ve formě anotovaného videa skrze rozhraní HDMI. V závěru práce jsou diskutovány dosažené výsledky, možnosti implementace systému UVDAR do samostatného zařízení a autorovy náměty pro další vývoj.

Klíčová slova: UVDAR, drony, vzájemná relativní lokalizace, obrazový senzor, detekce jasových značek, Houghova transformace, prokládání 3D přímkami, programovatelná hradlová pole

Překlad názvu: Implementace zpracování obrazu v systému UVDAR na FPGA

Contents

Figures

Tables

Introduction

Localisation algorithms are used extensively in modern robotics, particularly for multi-robot systems. The localisation task can be approached via multiple different ways, traditionally using absolute localisation systems that cooperate with an external source of localisation signals (e.g., a navigation satellite), requiring appropriate receivers to be mounted on the robots. However, these systems are limited to outdoor or well defined indoor environments because of their working principle (e.g., due to a signal perception requirement) and with growing numbers of robots in the multi-robot system, they usually introduce new challenging problems that need to be solved for a proper functionality. In order to overcome the practical limitations of the absolute localisation approaches, relative localisation systems are being researched as they require only computational and sensory resources carried by the robots themselves.

The UVDAR (UltraViolet Direction And Ranging) system [1] developed by the Multi-Robot Systems Group at the Department of Cybernetics at the CTU is one of relative localisation systems and is described in detail in the chapter 1. Camera images are used as the only source of localisation information used by the UVDAR system as it relies completely on two image processing algorithms, particularly on FAST-like image feature detection and on a four-dimensional Hough Transform. Thus, the chapter 1 covers theoretical background of these algorithms and also describes the current software and hardware implementation of the UVDAR system.

The image processing algorithms are computationally demanding, which is the primary motivation for finding a way to separate the UVDAR system from computational resources which are used by the robots for many other important tasks, such as odometry calculation, mapping, mutual communication, etc. A FPGA (Field Programmable Gate Array) is considered for this task as is can be programmed to efficiently process large amounts of data in parallel. In order to evaluate feasibility of a FPGA implementation of the image processing algorithms, their computational and memory requirements are analysed in the chapter 2. State-of-the-art implementations of the FAST algorithm and of the Hough Transform are found in literature and their aspects are considered for the proposed FPGA architecture.

The chapter 3 is concerned with a selection of a suitable FPGA development board. Two different SoC (System-on-Chip) boards are compared based on parameters of

their FPGA and the CPU parts, as well as on their current price and availability on the market. Also, a custom camera board hardware design with a CMOS imaging sensor, which is originally utilised by the UVDAR system, is presented and a MCU board for its initial testing is described.

The actual FPGA experiments are discussed in the chapter 4. Used interfaces and their VHDL implementations are presented together with the VHDL implementation of the FAST-like algorithm. A setup of a HPS (Hard Processor System) of the SoC is described in detail, including a development of a Linux kernel module and a user space software application. At the end of the chapter, the total FPGA resources utilised by the proposed architecture are shown.

A discussion of shortcomings of the proposed architecture follows in the chapter 5. Ideas concerning a FPGA implementation of the Hough Transform and future improvements of the application software are discussed. The chapter 6 concludes the thesis.

Chapter 1

The UVDAR System

This chapter focuses on explanation of the main aspects of the UVDAR system. The system is overall complex and many different objectives must be fulfilled in order to achieve a robust system and to deploy it to real-world UAV swarms (i.e., multi-UAV systems).

The mutual localisation of the robots in cooperative swarms is an important tool for the performance of their cooperative tasks. All necessary information describing the swarm nodes (e.g., the current position, orientation and dynamics) must be always provided to each of them, otherwise no cooperation can be performed. Usually, the tools used to address the problem are dependent on the surrounding environment and also on the network architecture, which may be centralised or distributed.

The centralised architecture uses one central node with a processor which fuses all available information simultaneously to retrieve the state of all nodes. The information is then transmitted to the nodes using some conventional communication infrastructure, such as Wi-Fi, Bluetooth or another radio-based transmission method. The downsides of this approach are primarily:

- All nodes need to communicate with the central node. Whenever a node disconnects from the network, the state estimation of the swarm is incomplete.
- The processing power requirements of the central node grow rapidly with every node added to the swarm.
- The central node also presents a single point of failure, so a back-up central node should always be available in the swarm.

On the other hand, in **the distributed architecture** all the nodes process the available data about the swarm locally. The processing power grows slowly with the increasing swarm size, which results in better scalability of the whole system. But not all the important information is always available for each node which often means worse accuracy of the state estimation.

There are many methods of acquiring the current position and the orientation of the nodes. The most frequently used methods for **absolute localisation** (i.e., providing exact localisation data in global and/or local reference frames) among others are:

- GNSS (Global Navigation Satellite System) - This method relies on the perception of the satellite navigation signal where each node carries its own GNSS receiver. Nowadays the precision of the GNSS-RTK (Real Time Kinematics) localisation method is in a range of millimetres. However this method may only be used outdoors in the open-sky conditions with a direct satellite link and it may also suffer from jamming and/or spoofing. [2]

- Motion capture (MoCap) systems - These external localisation solutions are suitable for well defined indoor environments. A ground station (the central node) equipped with a set of cameras (usually infrared) calculates positions of all nodes inside the covered area and then provides the results for their coordination. [3]

Relative localisation frameworks are intended to be more flexible when used in GNSS-denied or unknown environments. They can be separated into two main categories:

- Distance-based frameworks - The relative distances between the nodes are acquired by e.g., wireless communication devices such as ultra-wideband radio (UWB) [4], Wi-Fi [5] or Bluetooth modules [6]. Their obvious advantage is the omnidirectionality of the working principle, although a signal jamming and/or wireless traffic overload may occur with increasing swarm size.

- Vision-based frameworks - The on-board camera sensors are used to detect neighbour nodes carrying specific patterns (passive markers) or light-source (active) markers to be easily detected by basic computer vision algorithms. These methods overcome wireless network limitations but introduce new ones, such as the FoV (Field-of-View) limits, dependence on the visibility and lighting conditions and also higher computational complexity.

The UVDAR system is an example of the distributed, relative localisation, vision-based framework which uses UV-light sources as node markers together with UV band-pass filters mounted on the lens of the cameras to avoid the sunlight saturation in the visible spectrum. [1] It is comprised of a complete software pipeline as well as a commercially available and a custom designed hardware underlay.

The following parts of this chapter present the overview of the system design as a whole, introduce the theory of the incorporated algorithms and also review the current implementation of the system.

1.1 System Design Overview

The UVDAR system can be understood as a 4-step pipeline as shown in fig. 1.1. Each step can be treated independently of the others as long as it produces data in a format expected by the next step.

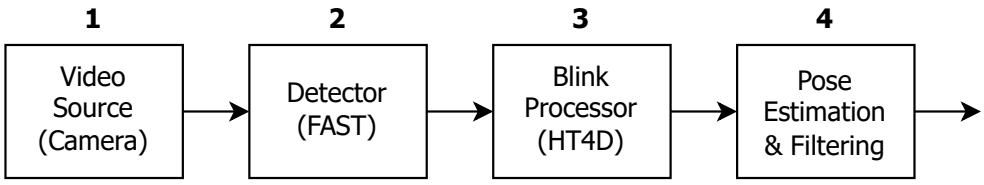


Figure 1.1: Block diagram of the UVDAR pipeline

1. **Video Source** - In a real-world application, the pipeline input is represented by a suitable camera device. The output of this block provides the raw images (video frames) at a reasonably high frame rate. The UVDAR system requires only monochromatic (grey scale) images at the input stage for the next steps.

2. **Detector** - This block implements a bright-spot detection algorithm. In particular, the FAST-like algorithm (Features from Accelerated Segment Test) is used. This block outputs a set of (x, y) pixel coordinates of the detected bright points as well as a set of pixels belonging to the projection of the sun used for additional filtration, while both sets are obtained once per each video frame.

3. **Blink Processor** - The sets of the bright spots (or markers) gathered from individual frames are first buffered as (x, y, t) triplets (referred to as *t-points*) in *t-sets*, with the latest frame corresponding to $t = 0$. The t-points lie along curves in (x, y, t) space (when the physical swarm node dynamics are limited) that can be approximated by straight lines, defined by their origin points and the pitch and yaw angles (ϕ, ψ). The lines are produced by the computation of the 4D Hough Transform (HT4D). The transform is also used to separate the sets of blinking markers belonging to individual swarm nodes as the markers belonging to the same rigid body (node) tend to lie on parallel lines. The swarm nodes can then be identified by their unique blinking patterns and when multiple blinking patterns are used by the same swarm node, its orientation can be also estimated. [7]

4. **Pose Estimation & Filtering** - Each set of the blinking markers should fit a real model of a swarm node carrying the blinking LEDs. With the prior knowledge of the mutual distances and positions of the LEDs on the UAV model, multiple methods can be used to obtain a relative pose of the actual swarm node carrying the camera w.r.t. an another swarm node observed as a set of its blinking markers, such as a method based on the Unscented Transform (UT) [8] [9], a solution to the Perspective-Three-Point (P3P) problem or the Iterative Closest Points (ICP) algorithm, which is currently implemented in the UVDAR system. These methods also translate the known precision of markers' detection into the covariance of the pose estimation for the next use in a linear Kalman filter for correction of the measured poses and for additional improvement of the robustness of the system.

1.2 Theoretical Background

1.2.1 Features from Accelerated Segment Test (FAST)

The FAST algorithm is a feature detection algorithm used in computer vision primarily for detection of corners or other visually distinctive features from an image based on characteristics of the neighbourhood pixels. Other popular feature detection algorithms include Harris corner detection, SIFT (Scale Invariant Feature Transform), SURF (Speed-Up Robust Features) or BRIEF (Binary Robust Independent Elementary Features).

Figure 1.2: Illustration of the FAST image feature detector. [10]

An illustration of a principle of the FAST algorithm is shown in fig. 1.2. As explained in the original proposal of this algorithm in [11], the segment test is based on pixel intensities of the circular neighbourhood of the candidate pixel at the centre of the circle. The test is passed if all contiguous pixels of the selected neighbourhood differ in their intensity from the intensity of the candidate pixel at least by a static threshold value.

The FAST-like algorithm

A simplified pseudocode of the algorithm used by the UVDAR system is summarised below. Intensity of every image pixel is tested for representing a marker point (the candidate pixel is brighter than the circular neighbourhood) or a sun point (the candidate pixel and the neighbourhood pixels are all very bright). When the image point retains its potential to be a marker point after the iterations over the neighbourhood pixels are finished, the pixel with highest intensity inside the interior (i.e., between the candidate pixel and the neighbourhood pixels) is stored as the detected marker.

In addition to the algorithm steps above, the UVDAR implementation also stores a binary mask of interiors related to the stored markers, which enables the algorithm to skip already processed marker areas so the markers are not processed and stored more than once. The sun points are used to filter out the detected markers too close to the projection of the sun as they might be just a result of a glare effect.

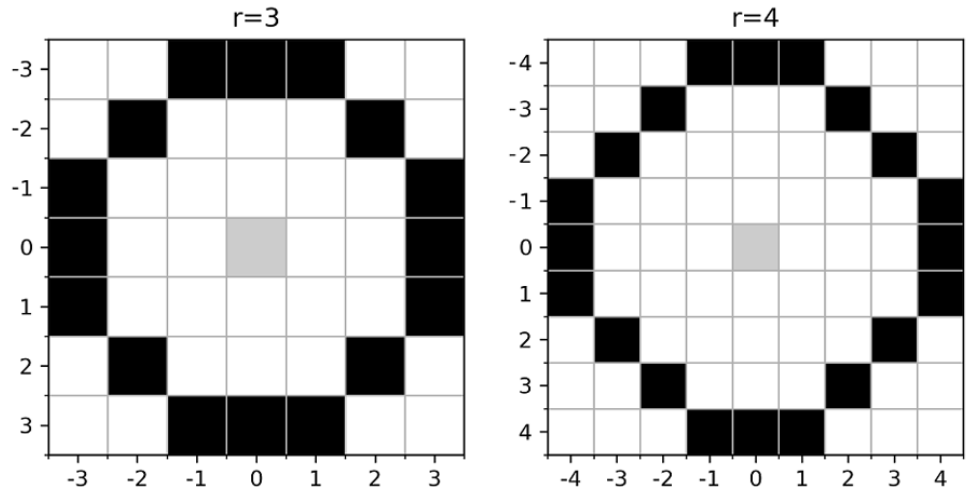
Algorithm 1 The FAST-like algorithm pseudocode

Require: $W > 0$ ▷ *image width*
Require: $H > 0$ ▷ *image height*
Require: $T > 0$ ▷ *static threshold value*
Require: $N[(i,j)]$ ▷ *array with opposing pixel coordinates (i,j) in sequence*

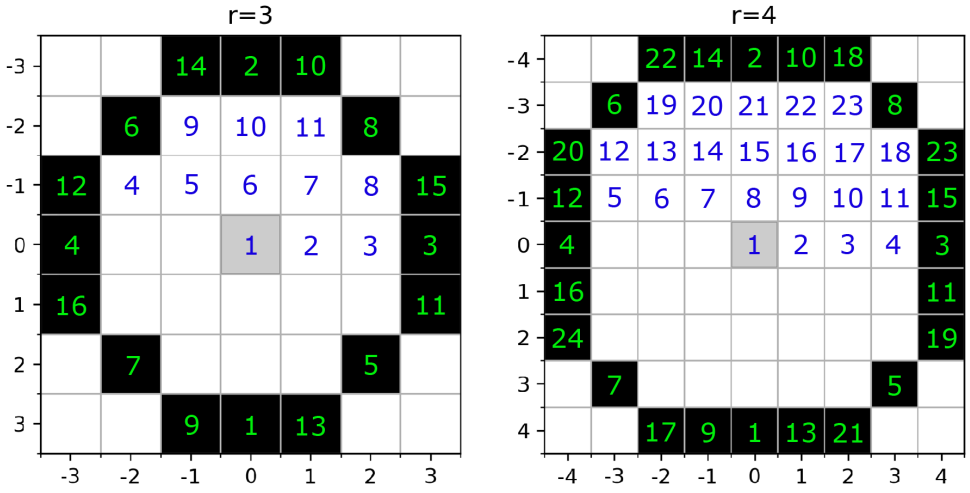
1: **for** $0 \leq x < W,\ 0 \leq y < H$ **do** ▷ *iterate over image coordinates (x,y)*
2: **if** $I_{x,y} > T$ **then** ▷ *if the point's intensity $I_{x,y}$ is large enough*
3: $S \leftarrow False$ ▷ *initially it does not have a potential to be a sun point*
4: **if** $I_{x,y} > 2T$ **then** ▷ *if the point is very bright*
5: $S \leftarrow True$ ▷ *then has a potential to be a sun point*
6: **end if**
7: $M \leftarrow True$ ▷ *initially it has a potential to be a marker*
8: $s \leftarrow 0$ ▷ *keep a count of very bright neighbourhood pixels*
9: **for** $(i,j) \in N$ **do** ▷ *iterate over the neighbourhood*
10: **if** $(I_{x,y} - I_{x+i,y+j}) < T/2$ **then** ▷ *if the difference is too small*
11: $M \leftarrow False$ ▷ *then it cannot be a marker point*
12: **if** S is *True* **then** ▷ *if the point could be a part of the sun*
13: $s \leftarrow s + 1$ ▷ *then the counter is incremented*
14: **else**
15: **break** ▷ *otherwise the point is not relevant anymore*
16: **end if**
17: **else** ▷ *if the difference is too large*
18: $S \leftarrow False$ ▷ *then it cannot be a sun point*
19: **end if**
20: **end for**
21: **if** M is *True* **then** ▷ *if the marker potential is preserved*
22: Store argmax of the neighbourhood's interior as a marker.
23: **else if** S is *True* **and** $s = \text{length}(N)$ **then**
▷ *if the sun potential is preserved and all neighbours are very bright*
24: Store (x,y) as a sun point.
25: **end if**
26: **end if**
27: **end for**

Circle approximations

A Bresenham's circle [12] is usually used as an approximation of the circular neighbourhood but arbitrary approximations are possible. The circle approximations used by the UVDAR system are compared to the Bresenham's approximations in fig. 1.3. Only the circles with the radius of 4 pixels differ in their shapes.



(a) : Bresenham's circles of radii 3 and 4.



(b) : Used circles of radii 3 and 4 with a denoted order of pixels for the algorithm.

Figure 1.3: Comparison of the used and Bresenham's circle approximations.

The speed of the segment test can be increased by a smarter choice of the order of the neighbourhood pixels. With the prior assumption of the size of the bright spots to be detected by this algorithm and the respective pixel intensity gradients present in the image, the UVDAR implementation minimises the number of required test steps by a selection of the mutually furthermost pixels in the circular neighbourhood. The order of the pixels used for the segment test (green) and the order of the interior pixels used for the search of the maximum value (blue) are shown in fig. 1.3b.

An equivalent conditional approach

The FAST-like algorithm can be also described using the following equations:

1. The candidate point (x, y) cannot be a marker point if the following condition is met:

$$\begin{aligned} &\exists (i,j) \in N : \left(I_{x,y} - I_{x+i,y+j}\right) < \frac{T}{2} \\ &\Leftrightarrow \min_{(i,j)} \left(I_{x,y} - I_{x+i,y+j}\right) < \frac{T}{2} \\ &\Leftrightarrow I_{x,y} - \max_{(i,j)} \left(I_{x+i,y+j}\right) < \frac{T}{2} \end{aligned} \tag{1.1}$$

2. The candidate point (x, y) retains its potential to be a marker point if the following condition is met:

$$\begin{aligned} &\forall (i,j) \in N : \left(I_{x,y} - I_{x+i,y+j}\right) \geq \frac{T}{2} \\ &\Leftrightarrow \min_{(i,j)} \left(I_{x,y} - I_{x+i,y+j}\right) \geq \frac{T}{2} \\ &\Leftrightarrow I_{x,y} - \max_{(i,j)} \left(I_{x+i,y+j}\right) \geq \frac{T}{2} \end{aligned} \tag{1.2}$$

3. The candidate point (x, y) cannot be a sun point if the following condition is met:

$$\begin{aligned} &\exists (i,j) \in N : \left(I_{x,y} - I_{x+i,y+j}\right) \geq \frac{T}{2} \\ &\Leftrightarrow \max_{(i,j)} \left(I_{x,y} - I_{x+i,y+j}\right) \geq \frac{T}{2} \\ &\Leftrightarrow I_{x,y} - \min_{(i,j)} \left(I_{x+i,y+j}\right) \geq \frac{T}{2} \end{aligned} \tag{1.3}$$

4. The candidate point (x, y) retains its potential to be a sun point if the following condition is met:

$$\begin{aligned} &\forall (i,j) \in N : \left(I_{x,y} - I_{x+i,y+j}\right) < \frac{T}{2} \\ &\Leftrightarrow \max_{(i,j)} \left(I_{x,y} - I_{x+i,y+j}\right) < \frac{T}{2} \\ &\Leftrightarrow I_{x,y} - \min_{(i,j)} \left(I_{x+i,y+j}\right) < \frac{T}{2} \end{aligned} \tag{1.4}$$

It is obvious that the condition (1.1) is the negation of the condition (1.2), the same holds for conditions (1.3) and (1.4). Also, the condition (1.2) directly implies the condition (1.3) and (1.4) implies (1.1). When subtracting unsigned integer types it is also desirable to check their values for possible underflow.

Using the conditional expressions, the FAST feature detection can be separated into two steps performed for each image patch (i.e., overlapping patches of size 7x7 pixels for the radius of 3 or 9x9 pixels for the radius of 4) containing the whole circular neighbourhood:

1. For each image patch, find and store:
   a. the maximum and minimum values of the neighbourhood pixels (i.e., max and min of $I_{x+i,y+j}$).
   b. the pixel intensity of the central candidate point (i.e., $I_{x,y}$).
   c. the maximum value of the pixels in the interior together with its coordinates (i.e., max and argmax).
2. If $I_{x,y} > T$, then evaluate the candidate point's potential to be:
   a. a marker point using the condition (1.2).
   b. a sun point using a condition $I_{x,y} > 2T$ together with (1.4).

This separation of the computation steps inherently leads to a minimal FPGA implementation which is described later.

1.2.2 Hough Transform (HT)

The Hough Transform is a computer vision method of detecting complex patterns of points in binary images. The patterns are characterised using their specific parameters so that spatially extended patterns can be transformed into compact features in the resulting parameter space (referred to as *Hough space*). This way, a pattern detection problem in the image space is translated into a local peak detection problem in the Hough space. [13]

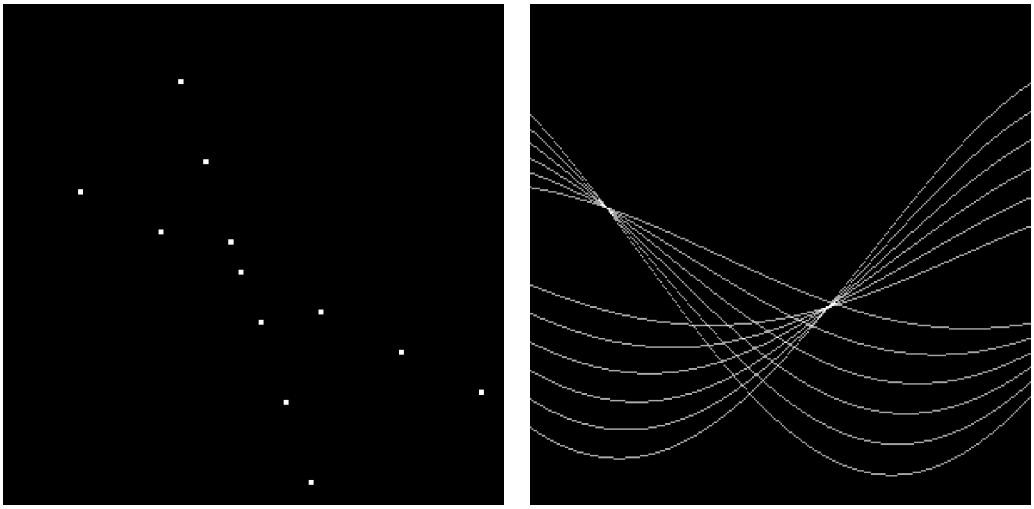
A 2D lines search example

The simplest application of the Hough Transform is the search for straight lines made up of the points of interest. Every line in (x, y) plane can be expressed as $y - mx - c = 0$ using the slope parameter m and the intercept parameter c, however the parameter $m \to \infty$ for lines approaching the vertical y axis.

More suitable line parameterisation in the form $\rho = x\cos(\theta) + y\sin(\theta)$ is usually used, where ρ and θ are the length and orientation of the normal vector to the line from the image origin. This way the parameter θ can be limited to a range $< 0,\ \pi)$ and the parameter ρ's magnitude is bounded approximately by the L^1-norm of the point vector.

Then each image point maps to a sinusoidal curve representing all lines the point is incident on in the resulting parameter plane $(\rho,\ \theta)$ (i.e., the Hough space), so the lines joining multiple points are found as the intersections of the matching sinusoidal curves (i.e., as the local maxima after summation of the curves' graphs).

Visual explanation is shown in fig. 1.4.



(a) : Points in the image plane: $p_i = (x_i, y_i)$

(b) : Sinusoidal curves in the Hough space: $\rho = x_i \cos(\theta) + y_i \sin(\theta)$ The two rifest intersections correspond to the two expected straight lines in the image plane.

Figure 1.4: Application of the Hough Transform for the straight line search. [14]

An important note is the points in the discrete image plane (i.e., the pixel raster) do not exactly lie on the incident lines in most cases which causes that the resulting curve intersections in the discrete Hough space also may not be exact, which is also determined by the resolution of discretisation of the parameters. The local maxima obtained by the summation of the discretised curves therefore lead to an approximation of the incident lines.

Similarly, more complicated analytic patterns such as circles [14] or ellipses [15] can be detected using the Hough Transform, but for the price of increasing number of dimensions of the Hough space resulting in higher computational and memory requirements.

Generalised Hough Transform (GHT)

Generalised Hough Transform (GHT) must be used when a detection of non-analytic patterns (i.e., without a general shape formula) is needed. [16]

To perform the detection of arbitrary shapes using the GHT, the shape at first has to be described using its edge points together with the angles between the edge normal vectors originating in that points and the x-axis, as explained in fig. 1.5 where two different edge points with the same normal vector direction Φ_i are shown. Also a fixed reference point (x_c, y_c) of that shape must be selected, allowing to compute the relative distances r_j^i and the angles α_j^i related to a j-th edge point. The parameters of the shape are usually referred to as a so-called *object template* (or a *R-table*).

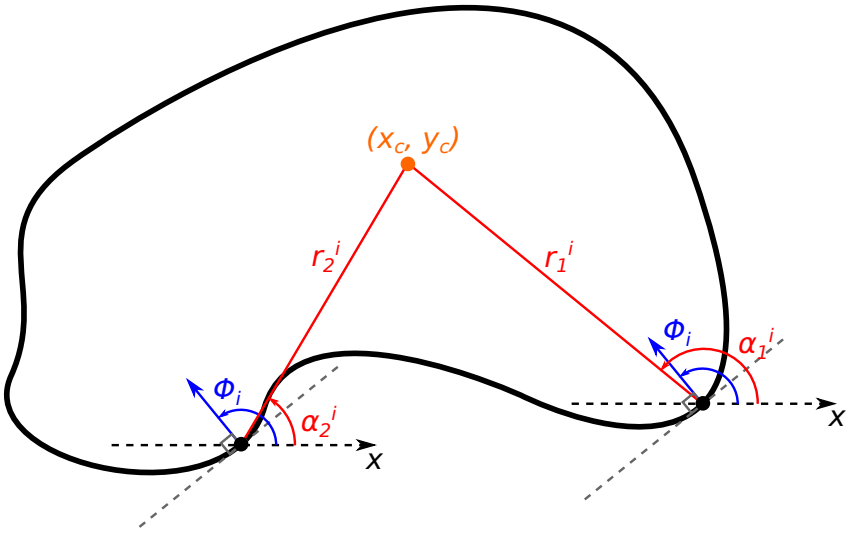


Figure 1.5: Explanation of parameters needed for arbitrary shape detection using the GHT.

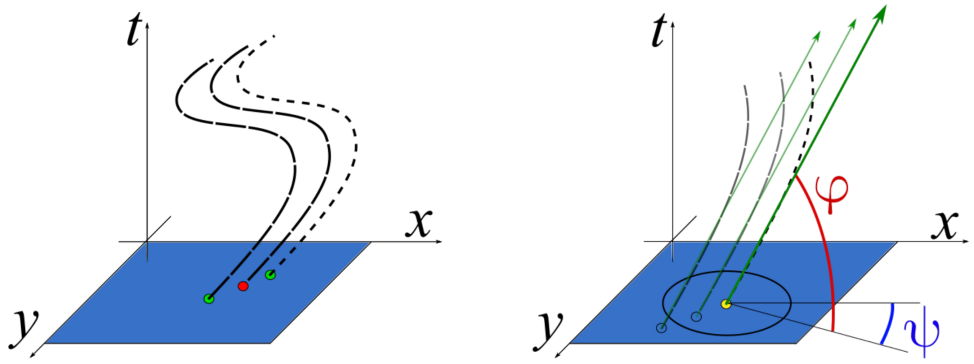
The search for the object in an image is simplified by carrying out a *voting procedure* in the Hough space. The voting process refers to the maxima (peak) search in the Hough space (an example of the voting procedure is also the search for sinusoidal curve intersections in the aforementioned detection of 2D lines).

The GHT also allows adding more 2D shape parameters like its rotation θ or a scale factor s, leading to a dense 4D parameter space. The resulting computational complexity is defined by four nested loops, in particular by iterating over all image points (points of interest) and by applying the object template (i.e., iterating over all 2D edge point coordinates defined for the desired shape, all possible rotations and all possible scale factors), generating every possible parameter combination that is used to increment the corresponding positions in the Hough space. [17]

The main principles behind the GHT (template generating and application on the points of interest, the voting procedure) are the same as for a 3D line search used by the UVDAR system.

A 3D line pattern detection (HT4D)

The UVDAR system utilises the Hough Transform to retrieve separate t-lines for each t-set of the t-points, more specifically to retrieve the corresponding origin points and the parameters of the t-lines, thus allowing retrieval of the blinking sequences along them. Visual explanation of the t-points and the t-lines is in fig. 1.6.



(a) : Moving along smooth curves (the green points are visible - ON, the red one is temporarily invisible - OFF)

(b) : Straight line approximation (the yellow point is the origin point in the current frame - the blue plane)

Figure 1.6: Explanation of the t-points (the moving markers w.r.t. time). [7]

In order to detect a straight line pattern in a 3D space (i.e., the t-lines in the (x, y, t) space), at least 4 parameters are required [18] which leads to a dense discretisation of a 4D Hough space. The memory and/or computational resources on most embedded UAV solutions usually do not suffice for a local maxima search in such space, hence the novel UVDAR approach [7] performs several simplifications to reduce the computational complexity:

- Relaxation of the t-lines reconstruction by discretising the t-line parameters with a conveniently large steps $\Delta\phi$ and $\Delta\psi$:

$$\begin{aligned} \phi_i &= i\frac{0.5\pi}{\Delta\phi},\ i \in \left\langle \frac{\phi_{\min}}{\Delta\phi}, \frac{\phi_{\max}}{\Delta\phi} \right\rangle \subset \mathbb{N} \\ \psi_j &= j\frac{2\pi}{\Delta\psi},\ j \in \left\langle \frac{\psi_{\min}}{\Delta\psi}, \frac{\psi_{\max}}{\Delta\psi} \right\rangle \subset \mathbb{N} \end{aligned} \tag{1.5}$$

  That also ensures lower susceptibility to small errors emerging from the preceding pipeline stages. Realistic t-lines are also constrained by the physics of the swarm system, permitting additional reduction of the Hough space size.

- To avoid direct application of the 4D Hough Transform to the t-sets, the algorithm uses a simpler 4D space of indices for permuted (ϕ, ψ) parameters. The Hough Transform transforms the t-points into the Hough space in a form of voxelated surfaces of pre-computed masks.

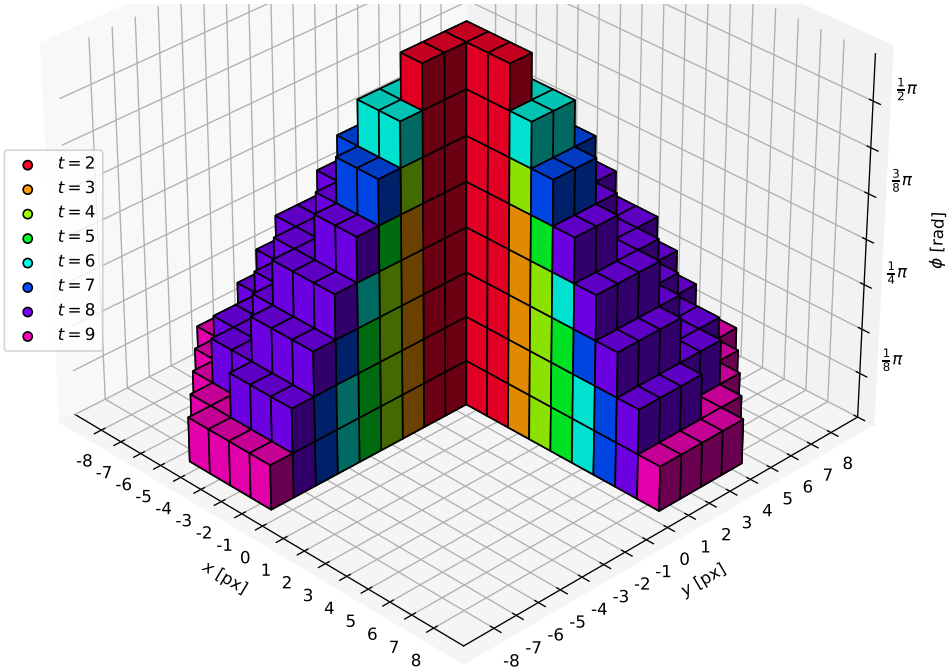


Figure 1.7: Generated masks for the pitch ϕ parameter (8 pitch and 10 time steps) - with a cut for a more informative visualisation.

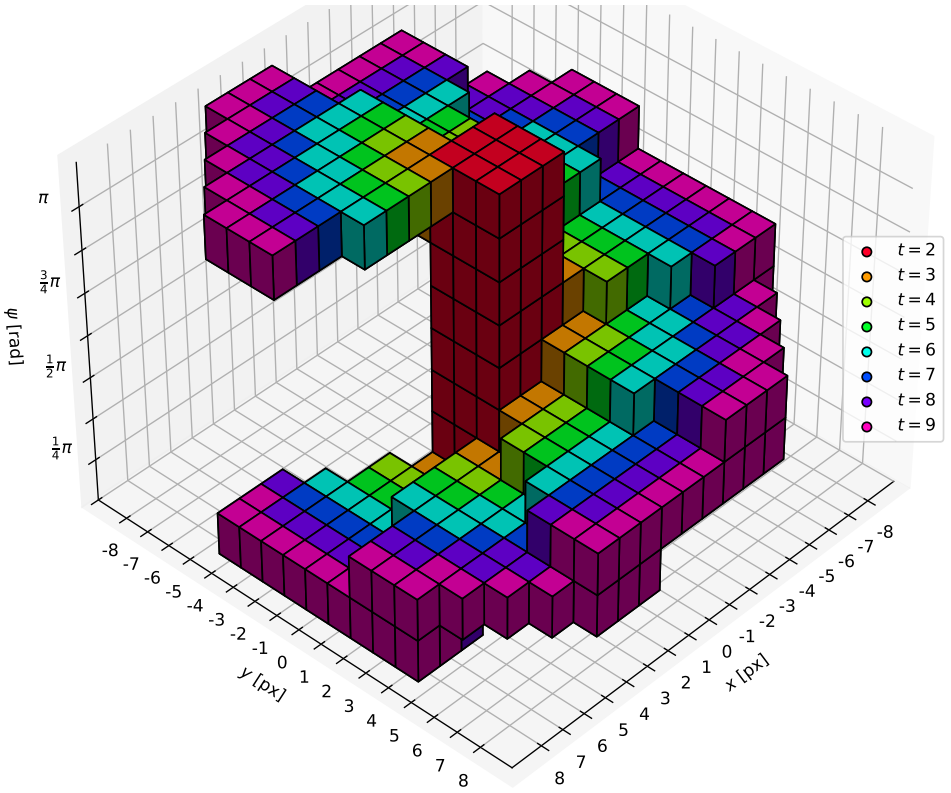


Figure 1.8: Generated masks for the yaw ψ parameter (8 yaw and 10 time steps).

Visualisations of generated voxel masks with sample parameter steps and time steps are shown in figures 1.7 and 1.8. The voxel surfaces belonging to consecutive time steps are overlapping to prevent discontinuities in the masks, which is not visible in the visualisations (new colours belong to the new voxels added at subsequent time steps).

The permutation of the t pairs of the generated 3D masks for the pitch (i.e., (x, y, ϕ)) and the yaw (i.e., (x, y, ψ)) parameters into t 3D arrays for the *4D hybrid masks* is done by representing the third dimension by the parameters' indices combined in a yaw-pitch order (i.e., $(x, y, N_{\phi \text{ steps}} \cdot j + i)$). Projection of the permuted hybrid masks to the x-y plane is shown in fig. 1.9.

Figure 1.9: Projection of sample hybrid masks to the x-y plane.

If multiple t-points belong to the same marker, their images in the Hough space intersect (i.e., the voxel values are summed up) at voxels corresponding to the parameters of the t-line which they are incident to. [7]

- The search for local maxima is conducted in a 2D space (referred to as *maxima array*). The maxima array is obtained by flattening the aforementioned 3D Hough space in the sense of selecting maxima in the t-axis direction. The

indices of permuted (ϕ, ψ) parameters matching the maxima values are stored in another 2D array (referred to as *angle array*) of the same shape.

Then the maxima found in the maxima array correspond to the expected origin points of the t-lines (valid for both ON and OFF states of the markers) and the matching t-line parameters are found using the angle array.

A summary of the t-lines retrieval algorithm

The process of retrieving t-lines is summarised in the following steps:

1. Initialisation:
   a. Allocate memory resources to store the 3D Hough space, the maxima array, the angle array and an accumulator array for t-sets of t-points (with a length equal to the number of time steps).
   b. Generate the steps for pitch and yaw parameters and store them in two separate arrays.
   c. For each time step, generate the voxel masks for pitch and yaw, permute them to 3D arrays for the 4D hybrid masks with combinations of the parameter array indices.
2. Insert a new t-set into the accumulator's first position ($t = 0$), drop the eldest t-set if the accumulator is full. The length of the most recent t-set is the number of visible markers. The number of expected markers (i.e., the markers currently in the OFF state) is a difference between the maximum of lengths of all accumulated t-sets and the number of the visible markers.
3. Apply the Hough Transform on the updated accumulator:
   a. Reset the Hough space and the maxima array.
   b. Apply the hybrid masks to the accumulated t-sets (i.e., increment the voxels of the Hough space belonging to the masks centred at the coordinates of the t-points).
   c. Flatten the Hough space to obtain the maxima array (i.e., store maxima values from the third axis direction) and the angle array.
4. Obtain the results:
   a. Nullify the maxima array around the visible markers to limit the search for the peaks only to the expected markers.
   b. Find the local maxima points in the maxima array and the corresponding permuted indices in the angle array. The search can stop when the number of found peaks is equal to the number of expected markers.
   c. Retrieve the blinking patterns, average yaw angles and average pitch angles for the origin points that are stored in an array made of the visible markers and the found peaks.
5. Repeat from the step 2.

Blinking pattern retrieval

The blinking frequencies (or the blinking patterns) are retrieved by clustering all t-points close to the identified t-lines. This is done by considering small relative distances of origin points of the relevant t-lines from the centre of a *t-cylinder* which is generated around a *t*-axis of the retrieved t-lines (shown as yellow dashed line) passing through the origin points (yellow) as visualised in fig. 1.10. The green dashed line presents the real non-linear trajectory of the t-points.

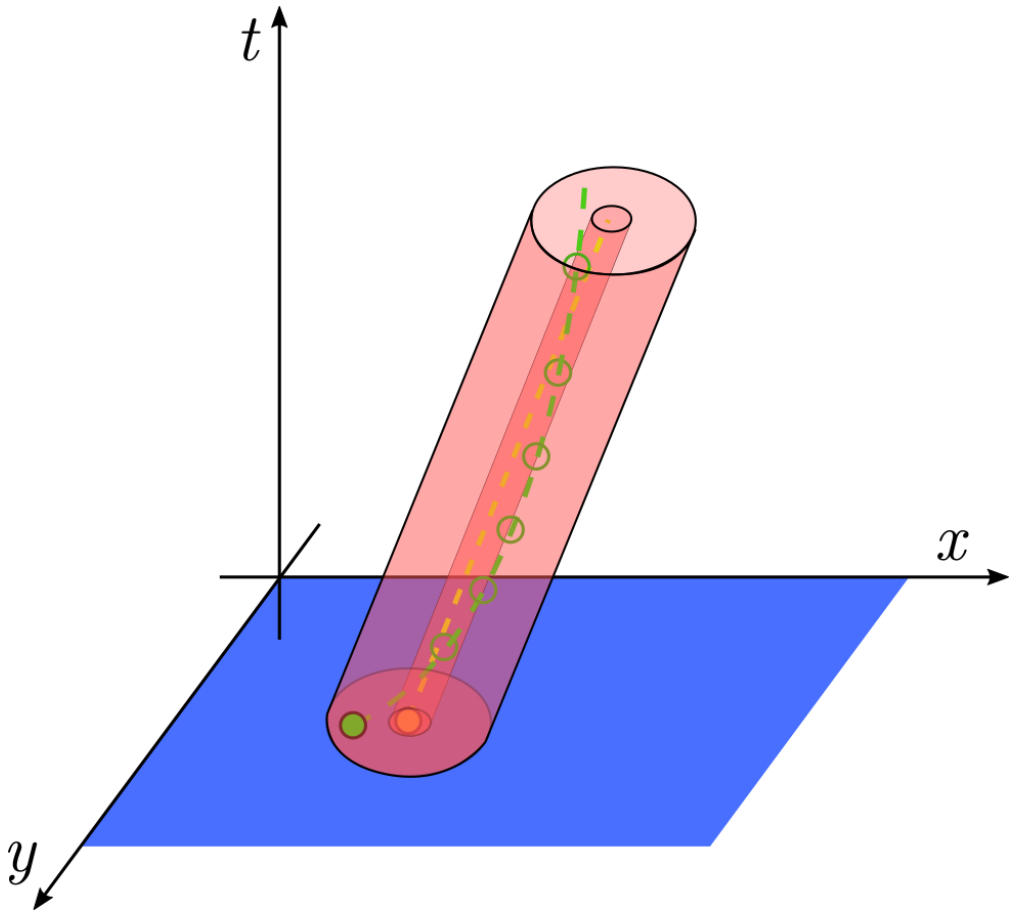


Figure 1.10: Cylinder shell defined around the retrieved t-line. [7]

The expected radius range of the t-cylinder shell at a certain time step is calculated using the pitch parameter already obtained from the angle matrix for the estimated origin point. All visible t-points in the corresponding t-set in the accumulator are then filtered by their distance from the estimated origin point compared with the expected radius range, the pitch values of the valid visible matches are used in a next iteration for an average pitch calculation. The visible matches are filtered again using a t-cylinder generated using the estimated average pitch. If no visible matches remain present in the filtered array at a certain time step, the state of the blinking marker is considered to be OFF, otherwise ON. This way the blinking pattern is reconstructed.

1.3 Current Implementation Details

The current implementation [19] is written in the C++ language as a Robot Operating System (ROS) package containing nodes and nodelets for the individual pipeline stages.

The main nodes and nodelets with their description are shown in table 1.1.

Node / Nodelet	Description
Detector	Detects bright points from the UV camera image.
BlinkProcessor	Extracts blinking signals and image positions of the markers detected previously.
PoseCalculator	Calculates approximate pose and error covariance of the MAVs carrying the UV LED markers.
Kalman	Filters out sets of detected poses with covariance based on positions or the included identities.
BluefoxEmulator	Generates an image stream similar to the output of the Bluefox cameras with UV band-pass filters.
LedManager	Sends commands to the controller boards that set the signals of the blinking UV LEDs on the current MAV using the Baca Protocol [20].

Table 1.1: UVDAR ROS nodes/nodelets and their description.

The software requirements (except the ROS distribution) include several ROS packages maintained by the MRS group, containing utility libraries, message type definitions and serial communication interface, a ROS package for accessing mvBlueFOX cameras and several libraries used for testing of the implementation in simulation.

1.3.1 Hardware Overview

The cameras used by the UVDAR system are Matrix Vision mvBlueFOX-MLC200wG [21]. The photo of the camera is in fig. 1.11a. This camera model has a 0.4MP resolution (752x480 pixels) and a frame rate of 60 FPS is used. It uses a monochrome global-shutter 1/3" CMOS imaging sensor MT9V034 from ON Semiconductor (formerly Aptina) with a quantum efficiency of about 38% at the wavelength of 395 nm [22]. This wavelength is produced by the active LED markers placed on the UAVs. [1]

The lens mounted on the camera modules are Sunex DSL215 fisheye lens with approximately 180° of the horizontal FOV, depicted in fig. 1.11b. Also near-UV band-pass filters MidOpt BP365-R6 are placed between the CMOS sensors and the lens.

(a) : Matrix Vision mvBlueFOX-MLC200wG camera module.

(b) : Sunex DSL215 fisheye lens.

Figure 1.11: Photos of the camera module and the lens used by the UVDAR system.

The custom hardware developed specifically for the UVDAR system is a LED controller board controlled by a STM32 microcontroller. [23] The LEDs driven by the controller board are ProLight Opto PM2B-1LLE with an emission angle of 130° and a maximum optical power of 315mW.

A custom camera board that was developed for this thesis uses the same CMOS imaging sensor as the mvBlueFOX camera modules. This way the input stage of the pipeline remains unchanged and the results can be easily validated against the current implementation. The custom board design is presented in the following chapter.

Chapter 2

FPGA Use Case

This chapter determines the memory and computational requirements of the two image processing algorithms considered for an implementation in the FPGA, i.e., of the FAST-like algorithm and of the 4D Hough Transform. Related approaches to FPGA implementations of the FAST algorithm and of the Hough Transform are also reviewed.

2.1 Initial Pipeline Stage

At first, the camera's parameters are defined in table 2.1 with their default values used by the current UVDAR implementation. The requirements of the input state of the pipeline are in table 2.2.

Name	Description	Value
CAM_{IW}	Image width	752 px
CAM_{IH}	Image height	480 px
CAM_{IA}	Image area (i.e., $CAM_{IW} \cdot CAM_{IH}$)	360960 px^2
CAM_{FR}	Frame rate	60 FPS
CAM_{CLK}	Pixel clock frequency	26.67 MHz

Table 2.1: Definition of the parameters of the camera.

The camera's CMOS imaging sensor MT9V034 provides the image data through a DCMI (Digital Camera Interface) running at the CAM_{CLK} frequency and the interface is described in detail in the next chapters of this work.

Requirement description	Value
Memory requirements	
Image (frame) (`uint8_t[]`)	$CAM_{IA} = 360960$ B

Table 2.2: The pipeline input stage requirements.

It is shown in the following section that it is not necessary to store complete camera frame for the following image processing algorithms, thus reducing the memory requirements of the initial stage of the pipeline of the UVDAR system.

2.2 The FAST-like Algorithm

The FAST-like algorithm implementation uses parameters shown in table 2.3 for both radii of 3 and 4 pixels.

Name	Description	Value
$FAST_{T}$	Static threshold (8-bit unsigned)	105
$FAST_{DP}$	Maximum number of detected points	30
$FAST_{SD}$	Minimum distance of the markers from the sun	25 px
Radius of 3 pixels		
$FAST_{R3PS}$	Image patch (square) size	7 px
$FAST_{R3B}$	Size of the circular boundary	16 px
$FAST_{R3UI}$	Size of the used neighbourhood's interior	11 px
$FAST_{R3TI}$	Total size of the neighbourhood's interior	21 px
Radius of 4 pixels		
$FAST_{R4PS}$	Image patch (square) size	9 px
$FAST_{R4B}$	Size of the circular boundary	24 px
$FAST_{R4UI}$	Size of the used neighbourhood's interior	23 px
$FAST_{R4TI}$	Total size of the neighbourhood's interior	45 px

Table 2.3: Parameters of the FAST-like algorithm.

The pseudocode of the algorithm described in the previous chapter implies the requirements shown in table 2.4.

The worst-case time complexities are theoretical upper bounds of usual time complexities of the algorithm. The validation of a potential of a point to be a marker or a part of the sun ends in a majority of cases after only a few of iterations over boundary pixels of the neighbourhood, as there is usually a low number of the visible markers in the image (usually up to $FAST_{DP}$) and the sun's projection is also limited in its size. The usual time complexity of the sun-to-marker distance check depends on the number of the detected sun points.

From the memory requirements perspective, it is disadvantageous to store the sun points as an array of their coordinates because of their usual abundance in the images. It is favourable to store the sun points in a binary mask array (i.e., a binary image with bit values set to 1 when the binary coordinates correspond to the detected sun points). Then the sun-to-marker distance checks can be performed e.g., using a repeated dilation of the sun mask followed by a bit value checking at the corresponding marker coordinates.

Requirement description	Value
Computational requirements	
Worst-case complexity of potentials' validation	$\mathcal{O}\left(\mathrm{CAM_{IA}} \cdot \mathrm{FAST_{RxB}}\right)$
Usual complexity of the interior search	$\mathcal{O}\left(\mathrm{CAM_{IA}} \cdot \mathrm{FAST_{RxUI}}\right)$
Worst-case complexity of the sun distance check	$\mathcal{O}\left(\mathrm{CAM_{IA}} \cdot \mathrm{FAST_{DP}}\right)$
Memory requirements	
Markers (coordinates) (`uint16_t[]`)	$4 \cdot \mathrm{FAST_{DP}} = 120$ B
Sun points (coordinates) (`uint16_t[]`)	$4 \cdot \mathrm{CAM_{IA}} = 1443840$ B
Sun points (binary mask) (`uint8_t[]`)	$\mathrm{CAM_{IA}}/8 = 45120$ B
Memory requirements - radius of 3 pixels	
Boundary points (coordinates) (`int8_t[]`)	$2 \cdot \mathrm{FAST_{R3B}} = 32$ B
Interior points (coordinates) (`int8_t[]`)	$2 \cdot \mathrm{FAST_{R3UI}} = 22$ B
Memory requirements - radius of 4 pixels	
Boundary points (coordinates) (`int8_t[]`)	$2 \cdot \mathrm{FAST_{R4B}} = 48$ B
Interior points (coordinates) (`int8_t[]`)	$2 \cdot \mathrm{FAST_{R4UI}} = 46$ B

Table 2.4: Requirements of the FAST-like algorithm.

2.2.1 Related FPGA Implementations

Many image co-processor (or accelerator) FPGA architectures can be found in the literature [24] [25] [26] [27] [28] [29] for many different feature detection algorithms. All of them possess an ability to process the image data stream with a minimal processing latency (i.e., they are *streaming architectures*).

As the results of the FAST algorithm can be evaluated based on the neighbourhood of the pixels, which is completely contained in the corresponding image patches, it is suitable to provide a way to access the image patches without unnecessary delays and also to minimise the requirements for storing intermediate results. Usually, the sources of the image frames (e.g., cameras) provide the pixel data in a row-first order, meaning all the data in the first row must be read and handled before the second row's data is available.

This fact enforces such co-processors to implement so-called *line buffers* to store whole rows of the images. The buffers are filled sequentially with the incoming pixels from the video source. When all the line buffers required for the main image processing part are filled up, then the data then can be read from them in parallel, i.e., whole columns of the image patches are provided to the main processing part at once. [10]

The line buffers introduce an initial delay before the main image processing can take place. There is also an additional delay caused by the processing part itself before the first result is available at the output. After the two delays pass, the processing results are synchronised with the new incoming pixel data (i.e., with the pixel clock frequency). [30]

This streaming approach reduces the needed amount of memory resources of the FPGA chip as the image does not have to be stored in its memory completely.

2.2.2 Proposed FPGA Architecture

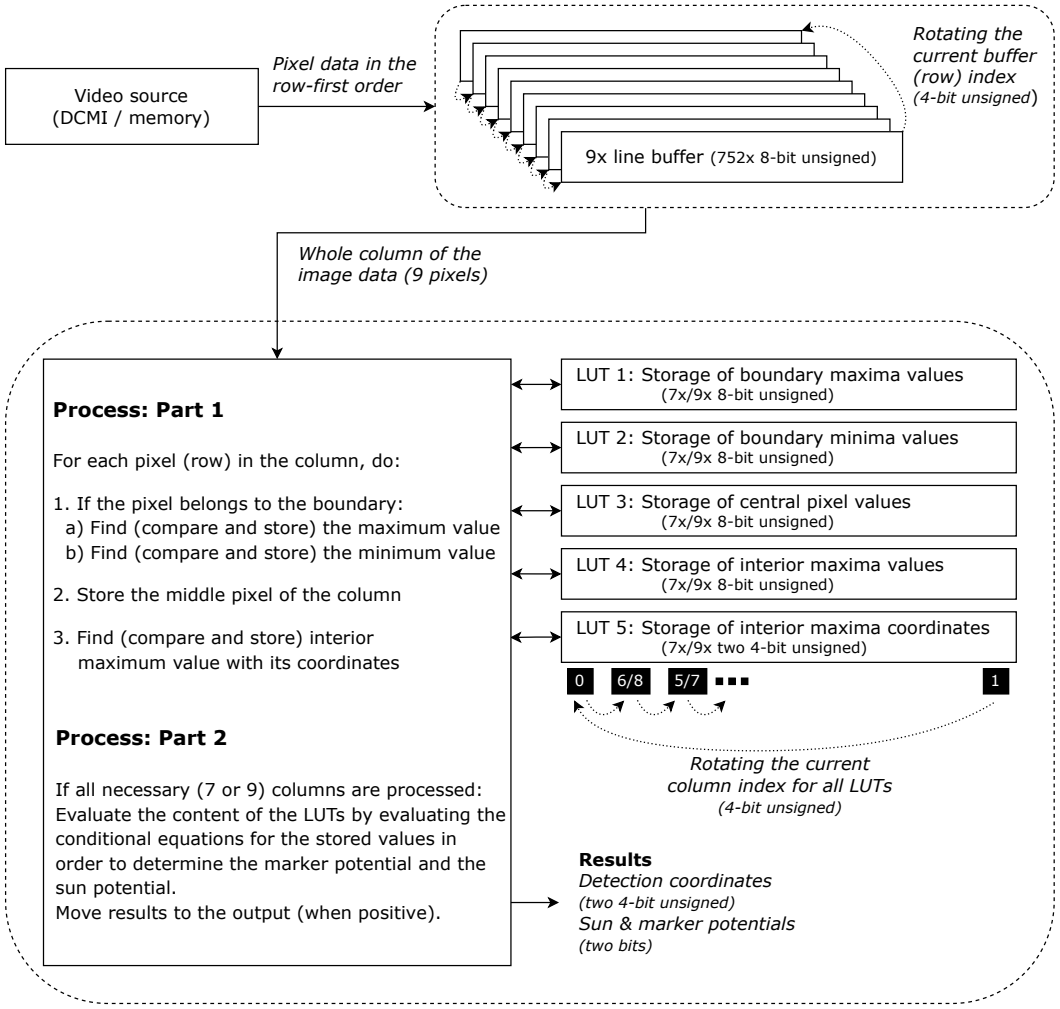


Figure 2.1: Proposed FPGA architecture for the FAST-like algorithm.

The proposed FPGA architecture for the implementation of the FAST-like algorithm is shown in a form of a block diagram in fig. 2.1.

- It utilises 9 line buffers (or FIFOs) to store the consecutive image rows (i.e., with sizes equal to $\mathrm{CAM}_{\mathrm{IW}}$ bytes).

  The resulting initial delay is thus equal to the height of the largest image patch times the image width, i.e., $\mathrm{CAM}_{\mathrm{IW}} \cdot \mathrm{FAST}_{\mathrm{R4PS}}$, in terms of the $\mathrm{CAM}_{\mathrm{CLK}}$ clock cycles. The process which handles memory access to the line buffers uses a `row_shift` signal to keep track of the currently filled line buffer. When the 9th

row (i.e., the `row_shift` is equal to 8) is about to be written to the last buffer (i.e., the initial delay passes), the new value is also read from the buffer (i.e., the memory is used in a write-first mode) together with the other 8 pixel values in the preceding buffers and the whole column is provided to the main FAST process (including the `row_shift` signal). After the end of every 9th row, the `row_shift` resets to 0 and the first line buffer gets overwritten as its old content is not needed anymore.

- The main process is split into 2 parts and uses the conditional approach as described in the previous chapter. Five LUTs (Look Up Tables) are utilised to store the local information about boundary minima and maxima values, central pixel values and the interior maxima with their coordinates. Each of the LUTs contain 7 (or 9) elements used by both of the steps (depending on the selected circular radius).

  The `row_shift` signal is used for a calculation of a `row_index` variable that is used as an index of the r-th pixel in the column array, so the pixel with $r = 0$ always lies in the topmost row of the image patch. The equation for its calculation is shown in equation (2.1).

$$\texttt{row\_index} = (\texttt{row\_shift} + r) \bmod \text{FAST}_{\text{R4PS}};\ r = 0, ..., \text{FAST}_{\text{R4PS}} - 1 \quad (2.1)$$

- The first part of the sequential process implements the comparisons of the maxima and minima values stored in the 5 LUTs with the pixel values in the current column. Hence each column is processed 7 (or 9) times on the same rising clock edge for the 7 (or 9) overlapping image patches and the results are stored into the LUT at the position corresponding to a `column_index` variable. When the first 7 (or 9) columns for each row are processed (i.e., the additional delay passes and all columns and rows of the first image patch are evaluated), the second part of the process takes place.

  The `column_index` variable rotates in such a way that the different pixel comparisons for the current column c are saved to the correct LUT positions for the overlapping patches. An auxiliary `column_shift` variable is initially set to 0 and gets incremented by 1 up to 6 (or 8) after each completion of the first part, then rewinds to value 0. The `column_index` for the currently processed column c of the image patch is computed as shown in equation (2.2), hence the pixels with $c = 0$ always lie in the leftmost column of the image patch.

$$\texttt{col\_index} = (\texttt{col\_shift} - c) \bmod \text{FAST}_{\text{RxPS}};\ c = 0, ..., \text{FAST}_{\text{RxPS}} - 1 \quad (2.2)$$

- The second part of the process is a straightforward implementation of the conditional equations. The data is read from the 5 LUTs (using an incremented `column_shift` variable as an index matching the currently finished image patch) and the final comparisons with the static threshold value are performed. Eventually the local coordinates of the detected marker or sun point inside the

image patch are moved to the output signals, including two bits indicating the designation of the point.

- At the moment when the detection results are read from the output, the final detection coordinates inside the original image frame are computed as the detection coordinates in the image patch plus the image coordinates of the current pixel being written to the line buffer and with each coordinate decremented by the processing delay (equal to the size of the image patch, $\text{FAST}_{\text{RxPS}}$).

The comparisons in the first part of the process can be implemented with several different approaches, depending on how the boundary and interior coordinates are accessed or derived:

1. The relative coordinates of the circular neighbourhood can be stored in another two LUTs (for the boundary and the interior coordinates respectively), in similar fashion as they are stored in arrays used within the current implementation. This approach requires additional iterating over the LUTs and comparing the stored coordinates with the current row and column indices used in the process.
2. The affiliation of a pixel inside the image patch to the circular boundary (or to the interior) can be evaluated using its relative centred coordinates (x, y) with equations (2.3).

$$\begin{aligned} (10r-3)^2 < 100(x^2+y^2) < (10r+5)^2 \\ 100(x^2+y^2) \leq (10r-3)^2 \end{aligned} \tag{2.3}$$

   The first equation produces the same circular boundary in a discrete pixel raster as the one used by the current implementation for both radii $r = 3$ and $r = 4$, the second equation refers to the interior.
3. The comparisons for the particular pixels in the columns can be hard-wired inside the VHDL process. No additional memory resources or loops are needed. Arbitrary shapes of the neighbourhood can be used, but at the cost of worse modifiability of the VHDL code.

For the proposed architecture, I opted for the third approach above as the VHDL code can be easily generated using a simple and parameterisable Python script.

The data types used by the architecture are 8-bit unsigned integers for all pixel-related values and 4-bit unsigned integers for all values related to coordinates, shifts and indices (limited by the image patch size).

2.3 The 4D Hough Transform

The parameters relevant to the implementation of the 4D Hough Transform with their default values are listed in table 2.5.

Name	Description	Value
$HT4D_{MS}$	Number of memory (time) steps	14
$HT4D_{PS}$	Number of pitch steps for the masks generation	16
$HT4D_{YS}$	Number of yaw steps for the masks generation	16
$HT4D_{NR}$	Nullify radius for resetting the maxima array	5
$HT4D_{RR}$	Reasonable radius for the blinking pattern retrieval	6
$HT4D_{MPS}$	Max. pixel shift for the mask width calculation	1
$HT4D_{FS}$	Frame scale for the masks width calc. $(0.3 \cdot CAM_{FR})$	18
$HT4D_{MW}$	Mask width $(1 + 2 \cdot HT4D_{MPS} \cdot (HT4D_{FS} - 1))$	35

Table 2.5: Parameters of the 4D Hough transform algorithm.

The Hough Transform implementation requires the pre-generated hybrid masks (i.e., the arrays of 3D coordinates) stored in the memory so that they are readily accessible by the algorithm to be applied onto the accumulated t-points. For simplicity of the visualisation of their sizes in fig. 2.2, an equal number of the pitch steps and of the yaw steps ranging from 2 up to 32 is assumed (i.e., $HT4D_{PS} = HT4D_{YS}$), for time step t ranging from 2 to 20.

The x and y point coordinates stored in the mask array for a time step t are in practise limited to a range $[-(t-1), (t-1)]$ and it is obvious from the visualisation in fig. 1.9 in the previous chapter, the permuted yaw-pitch index (the z coordinate) is limited to range $[0, HT4D_{PS} \cdot HT4D_{YS} - 1]$. It should be noted that the x, y range limits arise from a maximum pixel shift parameter (i.e., the largest allowed pixel distance between two t-points belonging to the same t-line in subsequent time steps) which is equal to 1 for all assumptions stated in this thesis. In turn, the minimal pitch angle of a t-line equals to 45°.

With the assumption of the range limits, a 6-bit signed integer (range -32 to 31) can be used to store both the x and y coordinates and 10-bit unsigned integer (range 0 to 1023) to store the z coordinate (for all the masks assumed in the aforementioned visualisation).

For the default values listed in the table 2.5 ($HT4D_{MS} = 14$, $HT4D_{PS} = 16$, $HT4D_{YS} = 16$), sufficient data types are 5-bit signed integers for x and y, and 8-bit unsigned integer for z (i.e., each 3D point takes 18 bits of the memory).

The sizes of the masks for the default parameter values are depicted in detail in fig. 2.3. All the mask arrays together contain 30556 points, occupying 550008 bits of the memory (or equally 68751 bytes, when the suggested data types are used).

As a side note, the mask sizes (and also their cumulative sums) form a surface in a 3D space with dimensions $(HT4D_{PS}, HT4D_{YS}, \text{mask\_size})$ that can be approximated by a function (2.4) with score $R^2 > 0.99$ (the coefficient of determination) for all numbers of steps in range 2 to 32 and $t \leq 20$, as can be seen in fig. 2.4.

$$\text{mask\_size} = a \cdot (HT4D_{PS})^b \cdot (HT4D_{YS})^c;\ a, b, c \in \mathbb{R} \tag{2.4}$$

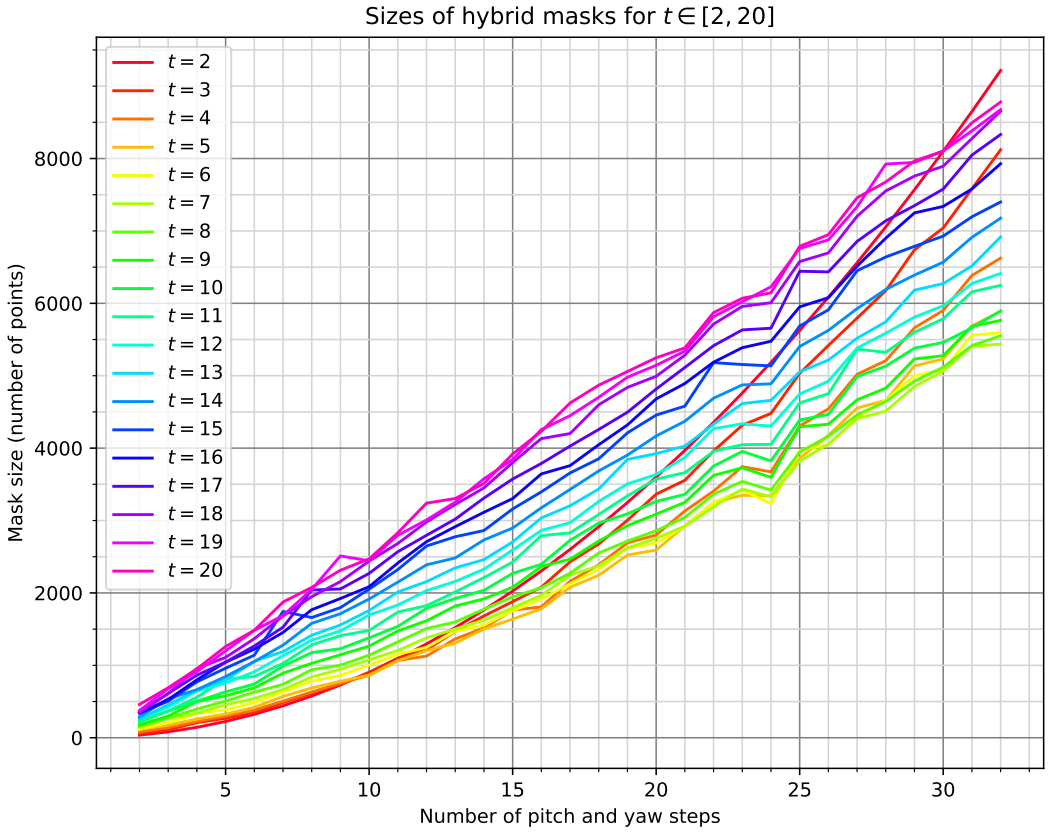


Figure 2.2: Sizes of the hybrid masks for equal numbers of pitch and yaw steps.

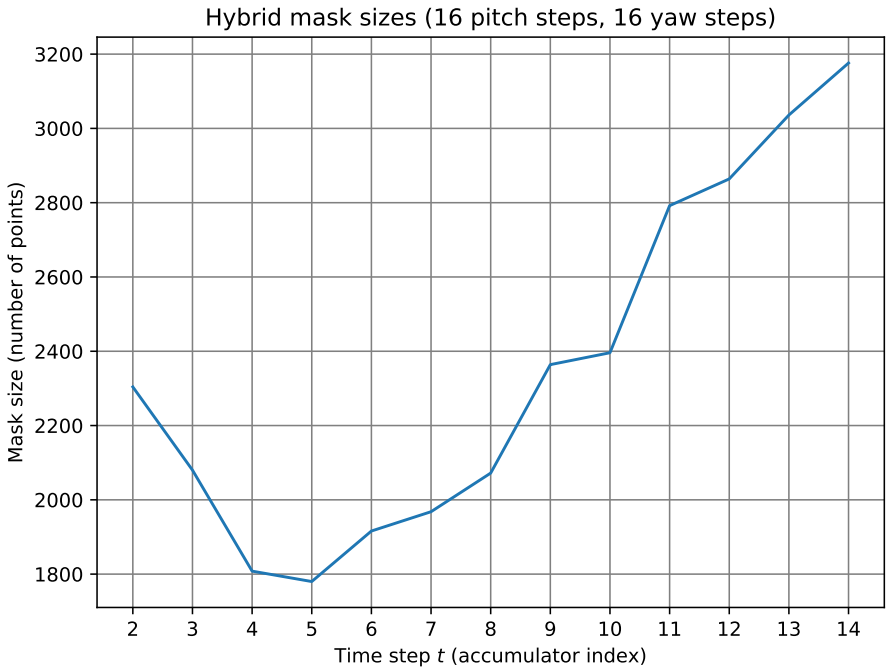


Figure 2.3: Hybrid mask sizes for the default parameter values.

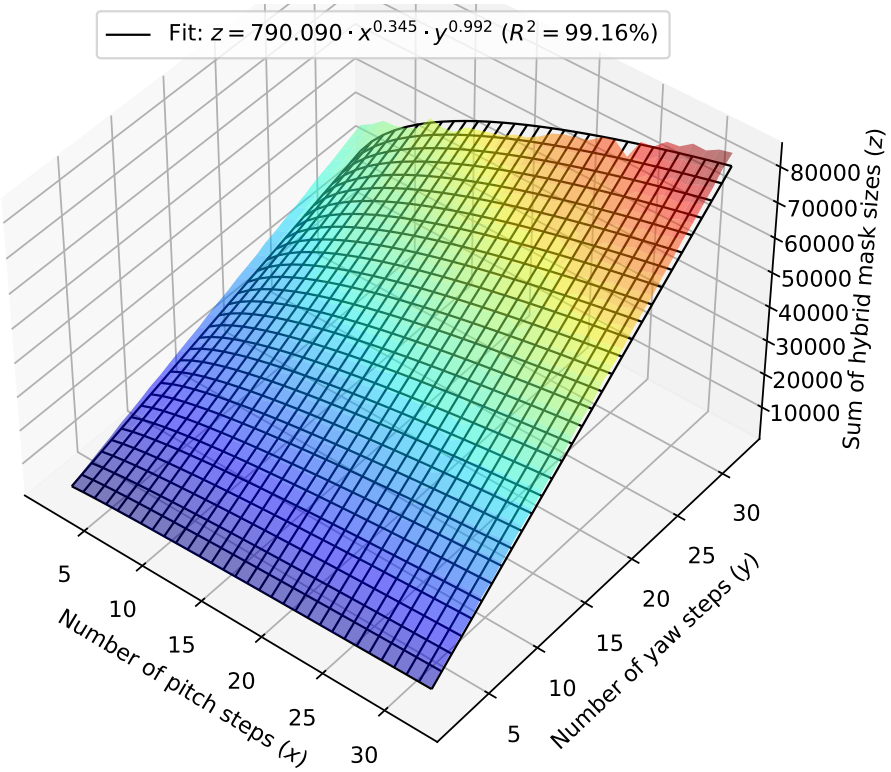


Figure 2.4: Sums of hybrid mask sizes for $t \in [2, 14]$ - 3D surface approximation.

Furthermore, the parameters a, b, c can be approximated as well, as functions of the time step t, leading to a formula (2.5) for computing the mask size (its number of points) for a particular time step t.

$$\text{mask\_size} = \left(\alpha_a + \frac{\beta_a}{1 + \gamma_a^{-t+\delta_a}}\right) \cdot (\text{HT4D}_{\text{PS}})^{\left(\alpha_b + \beta_b^{-t+\gamma_b}\right)} \cdot (\text{HT4D}_{\text{YS}})^c \quad (2.5)$$

Parameter	Optimal value
α_a	-14.4478
β_a	283.172
γ_a	1.27306
δ_a	12.0956
α_b	0.03493
β_b	1.25669
γ_b	1.98181
c	0.99576

Table 2.6: Optimal parameters of the formula approximating the computed sizes of the hybrid masks.

The formula (2.5) can be used to determine the upper bound of the necessary memory utilisation when the hybrid masks are generated for new parameters. The formula's 8 real parameters have optimal values listed in table 2.6, with $R^2 \approx 0.996$ for all numbers of steps in range 2 to 32 and time steps $t \leq 20$.

The computational and memory requirements of the current implementation of the 4D Hough Transform are summarised in tables 2.7 and 2.8.

Apart from the main memory requirements listed in the table 2.8, there are also several auxiliary arrays needed, e.g., for storing the detected blinking patterns and the averaged pitch and yaw angles.

Comp. complexity description	**Value**
A t-set insertion	$\mathcal{O}\left(\mathrm{FAST}_{\mathrm{DP}}\right)$
The Hough space reset	$\mathcal{O}\left(\mathrm{CAM}_{\mathrm{IA}} \cdot \mathrm{HT4D}_{\mathrm{PS}} \cdot \mathrm{HT4D}_{\mathrm{YS}}\right)$
The maxima array reset	$\mathcal{O}\left(\mathrm{CAM}_{\mathrm{IA}}\right)$
The hybrid mask application	$\mathcal{O}\left(\text{max. mask size} \cdot \mathrm{FAST}_{\mathrm{DP}} \cdot \mathrm{HT4D}_{\mathrm{MS}}\right)$
The Hough space flattening	$\mathcal{O}\left(\mathrm{CAM}_{\mathrm{IA}} \cdot \mathrm{HT4D}_{\mathrm{PS}} \cdot \mathrm{HT4D}_{\mathrm{YS}}\right)$
The maxima array nullification	$\mathcal{O}\left(\mathrm{FAST}_{\mathrm{DP}}\right)$
The maxima search	$\mathcal{O}\left(\mathrm{CAM}_{\mathrm{IA}}\right)$
A blinking pattern retrieval	$\mathcal{O}\left(\mathrm{FAST}_{\mathrm{DP}} \cdot \mathrm{HT4D}_{\mathrm{MS}}\right)$

Table 2.7: Computational requirements of the HT4D algorithm.

Memory requirement description	**Value**
Pitch steps (`float32_t[]`)	$4 \cdot \mathrm{HT4D}_{\mathrm{PS}} = 64$ B
Yaw steps (`float32_t[]`)	$4 \cdot \mathrm{HT4D}_{\mathrm{YS}} = 64$ B
Cotangent maxima (`float32_t[]`)	$4 \cdot \mathrm{HT4D}_{\mathrm{PS}} = 64$ B
Cotangent minima (`float32_t[]`)	$4 \cdot \mathrm{HT4D}_{\mathrm{PS}} = 64$ B
Accumulator for t-sets (`uint16_t[]`)	$6 \cdot \mathrm{FAST}_{\mathrm{DP}} \cdot \mathrm{HT4D}_{\mathrm{MS}} = 2520$ B
Hough space (`uint8_t[]`)	$\mathrm{CAM}_{\mathrm{IA}} \cdot \mathrm{HT4D}_{\mathrm{PS}} \cdot \mathrm{HT4D}_{\mathrm{YS}} \approx 92$ MB
Maxima array (`uint8_t[]`)	$\mathrm{CAM}_{\mathrm{IA}} = 360960$ B
Angle array (`uint8_t[]`)	$\mathrm{CAM}_{\mathrm{IA}} = 360960$ B
Hybrid masks (`(u)int8_t[]`)	$3 \cdot \mathrm{sum}(\text{mask sizes}) = 91668$ B

Table 2.8: Memory requirements of the HT4D algorithm.

It is obvious that unlike the memory requirements of the FAST-like algorithm, the memory utilisation of the 4D Hough Transform is too large in comparison with common internal memory resources inside embedded microcontrollers or FPGA chips. For a practical embedded application of this algorithm in its current form, an external memory resource has to be used.

2.3.1 Related FPGA Implementations

One of the earliest implementations of the Hough Transform on FPGA for 2D line pattern detection is presented in [31]. More specifically, the Fast Incremental Hough Transform (FIHT2) [32] is used as it does not require any trigonometric operations and only addition operations are needed. Their architecture also consists of constant value multipliers needed for a correction of the values retrieved via incrementation. This incremental approach, however, applies only to 2D line patterns.

More recent implementations for 2D line detection are discussed for instance in [33] and [34] focusing on the its use for LDWS (Lane Departure Warning System) application. The first paper uses the (ρ, θ) line parameterisation (as described in the first chapter) and their implementation limits the θ parameter values to a range of ± 20 degrees with a step of 1 degree. Therefore 41 parallel units for individual degrees are proposed in the system, each one processing only a 1D space for the ρ parameter. In the second paper, the (ρ, θ) line parameterisation is simplified to (b, θ) parameterisation at the cost of discarding horizontal lines and the required cotangent trigonometric values are stored in a single LUT. This simplification is also possible due to the intended use for a road lane detection because the horizon (i.e., a line passing through all vanishing points in the projective space) can be omitted.

Other FPGA implementations of the Hough Transform for detection of 2D lines and/or circles are discussed in [35], [36], [37], [38] and [39]. Few of them involve CORDIC (COordinate Rotation DIgital Computer) algorithm for simplification of mathematical operations (e.g., trigonometric functions) by transforming them into simple operators, such as adders and shift registers.

In [17], a complete FPGA architecture of Generalised Hough Transform (GHT) for detection of arbitrary 2D shapes (with arbitrary rotation and scale, i.e., using 4D parameter space) from detected edges in images is presented. A hierarchical system is proposed, splitting the architecture into a parameterisable number of so-called GHT cells (accessing single global R-table). Each GHT cell contains an edge buffer, a rotation unit processing only a part of all possible rotation angles (which are divided among all GHT cells) and a set of scale units taking the rotated points as inputs. Each scale units again processes only a portion of all possible scale values and is connected to a voter unit with its own memory for a portion of the parameter space. Thus each voter contains a portion of the total voting result that is fetched into a maximum search tree at the cell's output. Maximal votes and the corresponding addresses from all GHT cells are then combined and analysed by a software.

Moreover, the paper [17] also suggests the use of iterative refinement and cropping, i.e., increasing a resolution of the parameter space and reducing the ranges of parameter values after each iteration of the GHT, leading to a significant reduction of the required memory resources as the coarse results are sequentially refined. Similar approach might be considered for the UVDAR's HT4D implementation on FPGA.

A HLS (High Level Synthesis) for the Hough Transform acceleration on FPGA is discussed in [40]. The HLS is an automated design process that relies on programming languages (such as C/C++) in which the system design is prepared first as a

functional software. The automation resides in the ability to directly transfer that design into a RTL (Register-Transfer Level) architecture synthetisable for the target FPGA chip, while using various memory and performance optimisation techniques such as array partitioning, loop pipelining and loop unrolling. Therefore the HLS is a tool preferred for a fast evaluation of new algorithms on FPGAs instead of time consuming development using ordinary Hardware Description Languages (HDLs).

Chapter 3

Hardware Selection & Design

3.1 FPGA Boards

An appropriate FPGA development board had to be selected for the experimental part of this work, while addressing the computational and memory demands of the image processing algorithms. Due to the current low availability of the integrated components on the market, two different in-stock low-cost boards have been considered for the task, namely the Microchip (Microsemi) Hello FPGA board and the Terasic DE10-Nano board.

Both of the boards offer the conventional Arduino shield headers which were selected as the input interface to a custom camera expansion board with a hardware design described later in this chapter.

3.1.1 Microchip Hello FPGA Board

The main board of the Hello FPGA kit [41] is shown in fig. 3.1. The board offers Microsemi M2S010-1VFG256 SmartFusion2 SoC chip connected to Micron MT41K1G8RKB-107:N DDR3 8x1Gbit (1GB) SDRAM memory.

The SoC chip is programmed as a whole, i.e., the FPGA part together with the single-core ARM Cortex-M3 MSS (Microcontroler SubSystem), using the USB 2.0 port and an on-board USB-UART bridge connected to a Microchip PIC32MX795F512L microcontroller, which accesses a 8MB SPI flash memory to store compiled FPGA and MSS designs, or via an external Microchip FlashPro4/5/6 programmer connected to the JTAG interface of the SoC chip.

This kit also comes with a camera board containing a CMOS imaging chip OmniVision OV7725 with a VGA resolution of 640x480 pixels and with a 24-bit RGB pixel format. The last part of the kit is a LCD display board also offering a VGA resolution.

Unfortunately, the two expansion boards included in the kit are attached to the main board using the 20-pin expansion headers in a way that makes the Arduino headers inaccessible, especially when the LCD board is placed on the top side of the main board, thus prohibiting to attach the custom camera board. The included cam-

era board has also no further use for the UVDAR system because of the insufficient quantum efficiency of the RGB CMOS chips in the UV spectrum range.

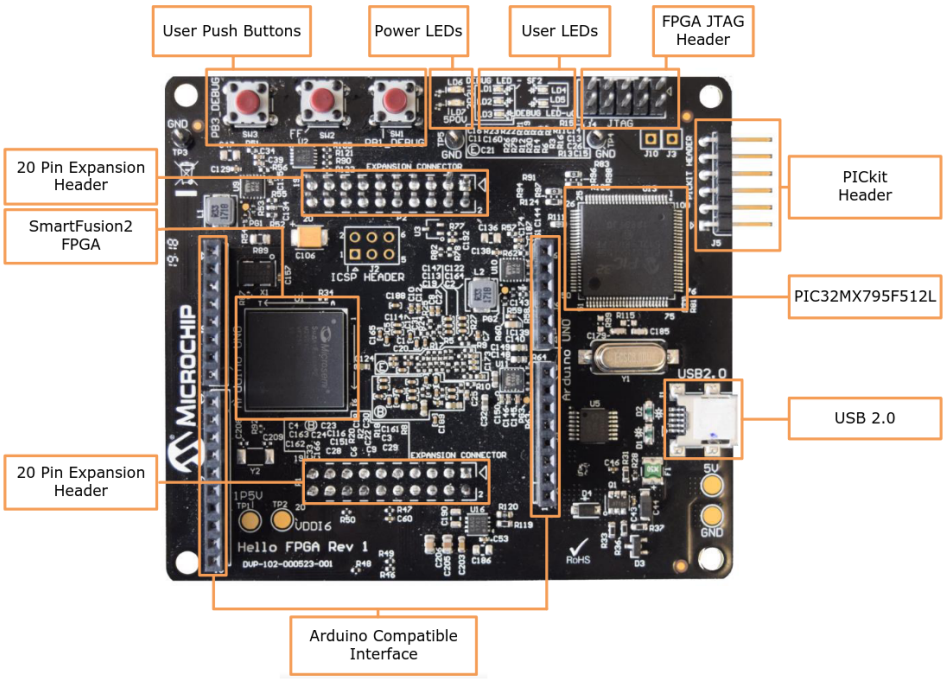


Figure 3.1: Microchip Hello FPGA Board Overview.

3.1.2 Terasic DE10-Nano Board

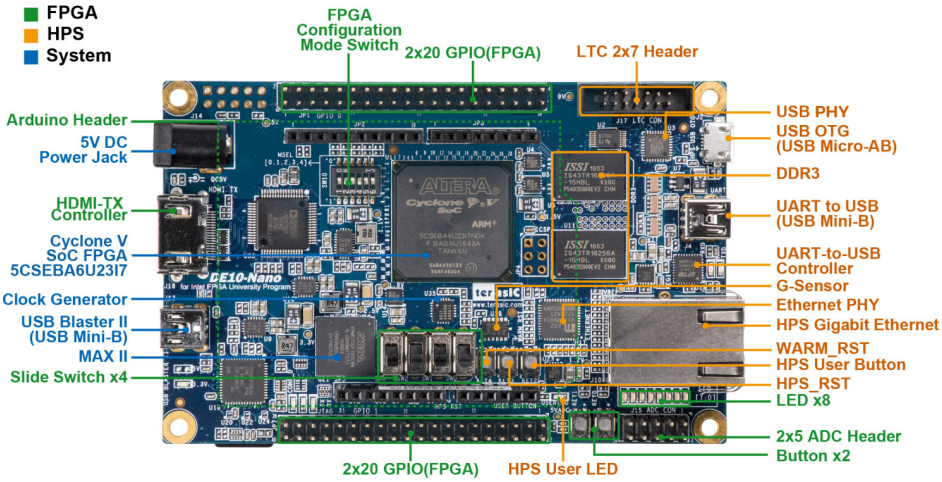


Figure 3.2: Terasic DE10-Nano Board Overview.

The main board of the Terasic DE10-Nano Kit [42] is shown in fig. 3.2. It offers Intel Cyclone V SE 5CSEBA6U23I7 SoC chip, which is connected to two ISSI IS43TR16256A-15HBL DDR3 16x256Mbit SDRAM memories (oferring 1GB in total).

The programming of the FPGA part of the SoC chip is done either using a USB 2.0 port connected to an on-board Intel EPM570GF100C5N CPLD chip emulating a JTAG interface (referred to as USB-Blaster II), or the compiled FPGA designs can be loaded via SDMMC interface from a microSD card, which also contains a firmware for the dual-core ARM Cortex-A9 HPS (Hard Processor System), during the boot of the HPS.

3.1.3 Boards Comparison

The two boards considered for the UVDAR image processing implementation on FPGA are compared by the most important parameters in table 3.1.

	Terasic DE10-Nano [42]	**Microchip Hello FPGA** [41]
SoC family	Intel Cyclone V SE [43]	Microsemi SmartFusion2 [44]
SoC chip	5CSEBA6U23I7	M2S010-1VFG256
SoC package	UBGA-672	VFPBGA-256
Kit price (Dec 2022)	∼ $223 (academic $189)	∼ $255
Built-in ARM processor parameters		
Core @ Max. freq.	(2x) Cortex-A9 @ 800MHz	Cortex-M3 @ 166MHz
Floating Point Unit	Yes	No
FPGA part parameters		
Logic Elements	110000	12084
Logic Array Blocks	4191	1007
Internal mem. blocks (BRAM)	553x M10K (10Kbit) 994x MLAB (640bit)	21x LSRAM (18Kbit) 22x uSRAM (1Kbit)
Total int. memory	6151Kbit (∼ 770KB)	400Kbit (∼ 50KB)
Multipliers (18x18)	224	22
Number of PLLs	6	2
External memory parameters		
Chip family	ISSI IS43TR16256A	Micron MT41K1G8RKB
Type @ Max. freq.	DDR3 @ 400MHz	DDR3 @ 333MHz
Configuration	(2x) 256M x 16bit	1G x 8bit
Total memory size	1GB (32bit data bus)	1GB (8bit data bus)
System development		
FPGA IDE	Intel Quartus Prime [45]	Microsemi Libero SoC [46]
JTAG Flashing Tool	USB-Blaster II (built-in)	FlashPro4/5/6 (external)

Table 3.1: Comparison of the considered FPGA boards.

The Microsemi SmartFusion2 family is comparable to the older Intel Cyclone II

family in the parameters of the FPGA part. A large disadvantage of the FPGA kit from Microchip is the lack of available documentation, examples and tutorials, while the DE10-Nano kit from Terasic is well documented and many design examples can be found online. The DE10-Nano kit also offers a much better performance for a lesser price, the development software (Intel Quartus Prime) is offered for free in a Lite version without any usage limits [45] (in contrast, the Microsemi Libero SoC is offered in a Silver version for free only for 1 year of usage [46]) and last but not least, the flashing of the Hello FPGA kit through the built-in PIC32 microcontroller is limited to an undocumented Windows GUI application (without a possession of the expensive external FlashPro programmer to use the JTAG interface).

For the reasons mentioned above, I eventually opted for the Terasic DE10-Nano kit and used it for the conduction of the experimental part of this work.

3.2 The MCU Board

For the possibility to evaluate the performance of the UVDAR's image processing algorithms on an embedded microcontroller, the STMicroelectronics Nucleo-H745ZI-Q development board was chosen. This board is shown in fig. 3.3 and includes STMicroelectronics STM32H745ZIT6U MCU [47] containing two 32-bit ARM cores, one ARM Cortex-M7 core able to run at a frequency of 480MHz and one ARM Cortex-M4 core able to run at 240MHz.

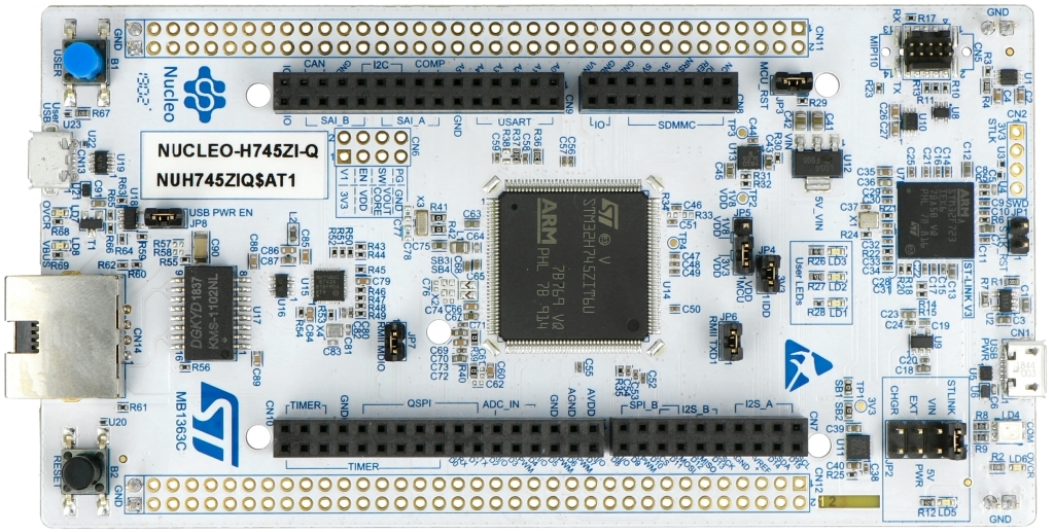


Figure 3.3: ST Nucleo-H745ZI-Q Board Overview.

This MCU chip offers a 2MB of flash memory for the compiled firmware and about 1MB of internal RAM memory (split to 512KB of AXI-SRAM on domain D1, 3 SRAM blocks of 128KB, 128KB and 32KB on domain D2, one SRAM block of 64KB on domain D3, 4KB of backup SRAM, 64KB of instruction RAM and 128KB of DTCM-RAM for critical real-time data), thus it is able to store a complete 8-bit camera image to the AXI-SRAM block or a 16-bit camera image split between

multiple blocks. The main asset of the chip relevant for this work is its built-in DCMI peripheral interface fully accessible through the board's Zio connectors, which are backwards-compatible extensions of the ordinary Arduino headers.

3.3 The Camera Board

The custom camera board contains On Semiconductor MT9V034 CMOS imaging sensor [22]. This sensor provides 10-bit monochromatic video output with a WVGA resolution of 752x480 pixels at 60FPS. It is available in an easily solderable CLCC-48 package (11.43mm x 11.43mm) with active imager size of 4.51mm x 2.88mm, resulting in a real pixel size of 6.0μm x 6.0μm.

The CMOS sensor requires 3.3V supply voltage and 26.67MHz master clock input for a proper operation. The sensor has a parallel DCMI output interface consisting of a pixel clock (PIXCLK) signal (i.e., a delayed master clock input), horizontal (HSYNC) and vertical (VSYNC) synchronisation signals and 10 data (D0 to D9) signals.

A general frame timing diagram of the DCMI signals is shown in fig. 3.4. By default, the MT9V034 provides HSYNC and VSYNC signals with active low state instead, with data to be sampled on the falling edge of the pixel clock.

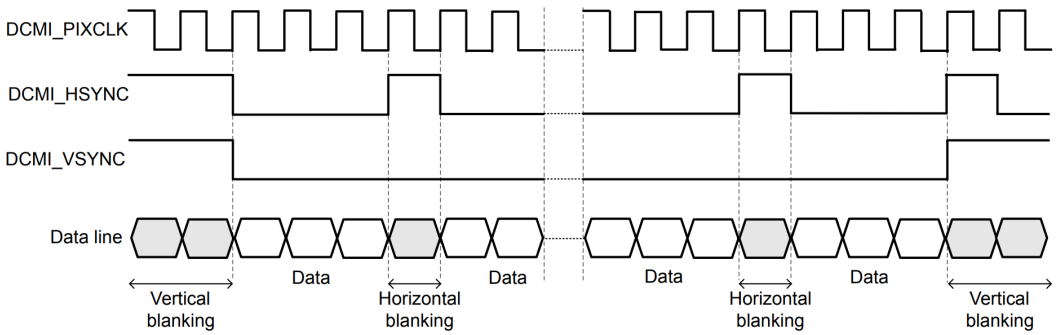


Figure 3.4: DCMI frame structure in hardware synchronisation mode. [48]

At the pixel clock frequency of 26.67MHz, the total frame time is 16.67 milliseconds (i.e., 60FPS). When the synchronisation blanking intervals are taken into account, the total frame resolution corresponds to 846x525 pixels.

The sensor is configured via a standard I2C (i.e., two-wire serial) interface. For instance, it allows to choose any smaller image format of the video output (i.e., window size cropping), to change an exposure rate of the sensor using the Automatic Exposure Control (AEC) or to set a gains for variable pixel regions of the sensor's internal ADC inputs using the Automatic Gain Control (AGC). The blanking times and synchronisation polarities can also be changed by writing appropriate internal registers of the sensor.

3.3.1 Hardware Design

The schematic showing connection of the CMOS camera sensor is in fig. 3.5.

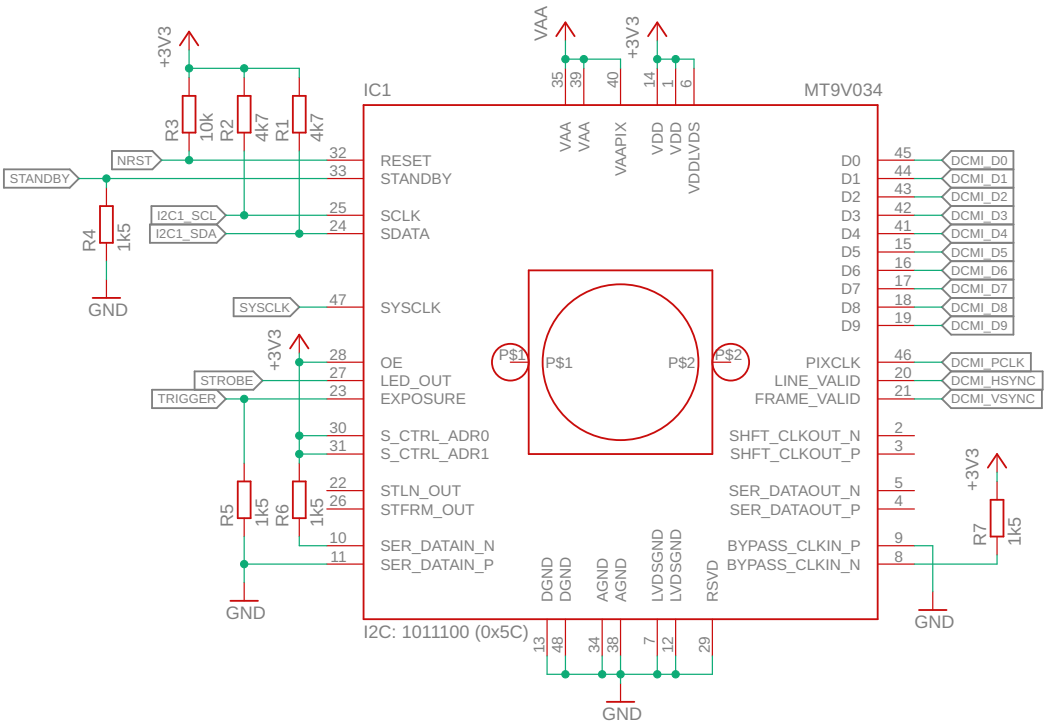


Figure 3.5: Schematic of the MT9V034 CMOS imaging sensor pin connections.

Except the CMOS sensor, there is also an auxiliary step-down converter STMicroelectronics ST1S40 with a 5V output voltage (with up to 3A capability) for unification of the supply source of the supported MCU and FPGA boards. The converter is fed by input voltages 6-18V from a common 5.5/2.1mm jack adapter connector. Optionally, an inertial measurement unit (IMU) sensor Bosch BMX160 can be soldered underneath the CMOS sensor for computation of the camera's orientation using an additional fusion algorithm.

The top layout of the camera board is shown in fig. 3.6. It has dimensions of 47x52 mm and supports a placement of a lens mount with distance of 22mm between the screw holes.

The table 3.2 shows the connection of the custom board headers to the pins of the supported boards. The star symbol next to the Nucleo pin names denotes their specific affiliation to the ST's Zio connector.

Pin	Camera signal	Nucleo	Hello FPGA	DE0/DE10 Nano
D13	DCMI_D0	PC6*	MSIO118NB4_R13	PIN_AH12
D11	DCMI_D1	PC7*	MSIO115NB4_R11	PIN_AG16
D10	DCMI_D2	PC8*	MSIO114NB4_M10	PIN_AF15
D7	DCMI_D3	PC9*	MSIO110PB4_T7	PIN_AH8
D4	DCMI_D4	PC11*	MSIO106NB4_R9	PIN_U14
D2	DCMI_D5	PD3*	MSIO103PB4_P6	PIN_AG10
D15	DCMI_D6	PB8	MSIO112PB4_R8	PIN_AG11
D14	DCMI_D7	PB9	MSIO112NB4_P8	PIN_AH9
D5	DCMI_D8	PC10*	MSIO109PB4_P10	PIN_U13
D3	DCMI_D9	PC12*	MSIO104NB4_N7	PIN_AG9
D12	DCMI_PIXCLK	PA6	MSIO115PB4_T11	PIN_AH11
D8	DCMI_VSYNC	PG9	MSIO113NB4_R10	PIN_AF17
D9	DCMI_HSYNC	PA4*	MSIO113PB4_P9	PIN_AE15
D6	MASTER_CLK	PA8	MSIO110NB4_T8	PIN_AG8
D0	I2C_SDA	PB7	MSIO78NB7_F3	PIN_AG13
D1	I2C_SCL	PB6	MSIO80PB7_G3	PIN_AF13

Table 3.2: Custom board signals connection to Nucleo and FPGA boards.

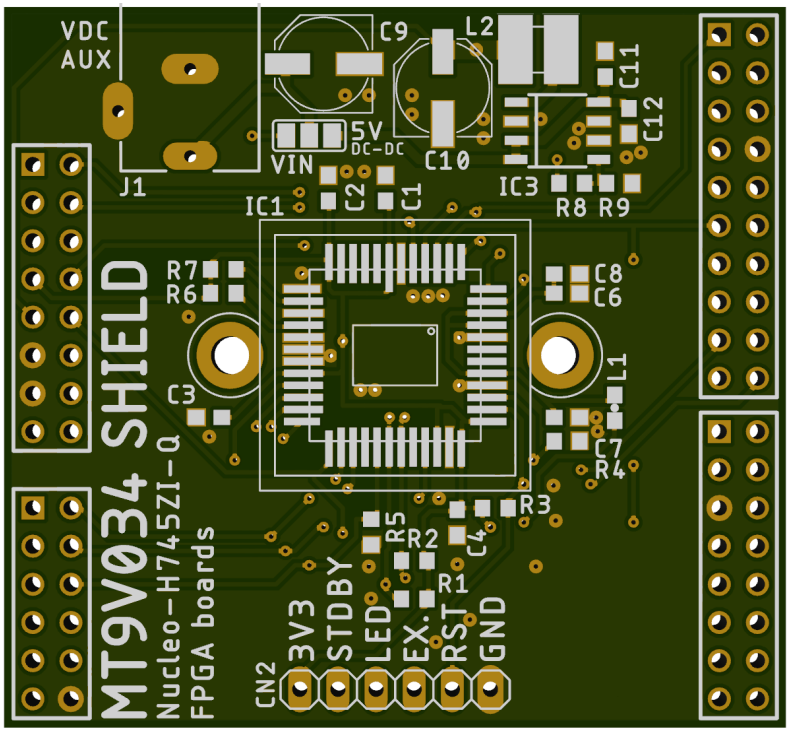


Figure 3.6: Top layout of the custom camera board.

3.3.2 Board Validation

A photo of the assembled camera shield board is in fig. 3.7. The board's functionality has been validated using the Nucleo-H745 board using a STMicroelectronics HAL (Hardware Access Library) in a STM32Cube IDE. The built-in hardware DCMI peripheral of the microcontroller was used for accessing raw camera data, together with an integrated DMA (Direct Memory Access) peripheral. The DMA peripheral is able to handle memory-to-peripheral, peripheral-to-memory and memory-to-memory transfers independently on the microprocessor's core instructions. It was configured to directly cooperate with the DCMI peripheral to transfer the image data to the available SRAM memory regions. Internal PLL has been configured to provide 26.67MHz clock output at the pin PA8 (alternate function MCO1). Pins PB6 and PB7 are used by the peripheral I2C1 (i.e., alternate functions SCL and SDA, respectively).

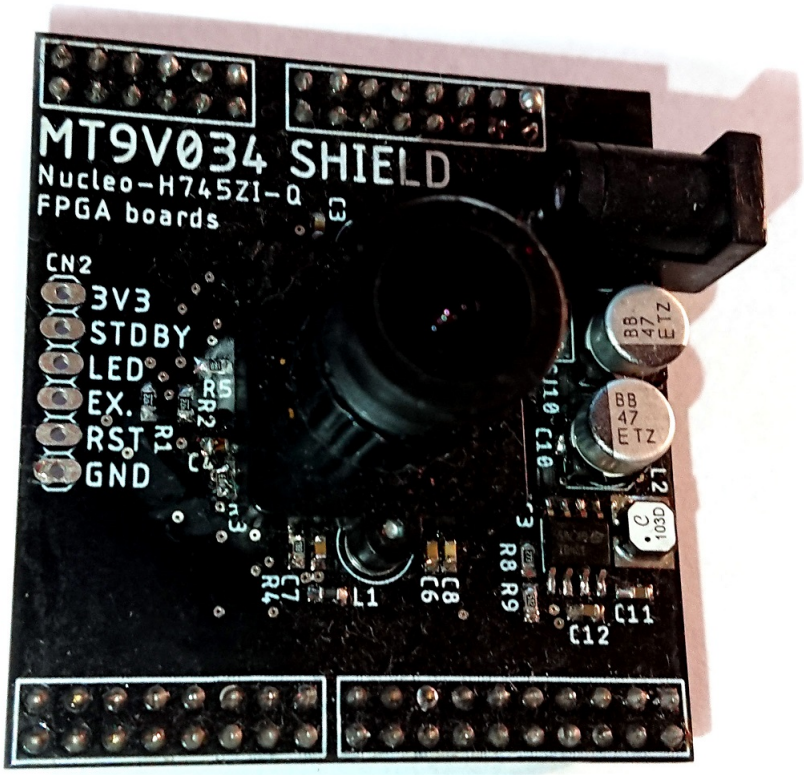


Figure 3.7: Photo of the assembled custom camera board with mounted lens.

The camera board is designed in a way that exactly 10-bit image data can be gathered using the built-in DCMI peripheral, because neither of the chips offer any hardware option to truncate the 2 most significant bits of the image data, resulting in a final data size of 16 bits per pixel in the memory. In turn, the complete image of 752x480 pixels results in $360960 \cdot 2 = 721920$ bytes of the memory and hence must be stored in multiple RAM sections. In particular, each third of the image

(i.e., 160 rows occupying 240640 bytes) has to be transferred separately by the DMA peripheral to two different sections of the internal RAM (i.e., on the domain D1 for two thirds and on D2 for the last third of the image).

A completeness and an integrity of the captured image has been validated visually after all three parts were transferred to a computer over a ST-Link programmer's UART interface. However, the camera frames cannot be transferred to the computer in real time over the USB-UART interface due to the speed limitations of the implemented Low-Speed USB interface (12Mbps) itself, the real data throughput required to transfer the frames at 60FPS equals approx. 43.3MBps (or 346.5Mbps). The MCU chip also offers an ULPI (UTMI+ Low Pin Interface) to access external High-Speed USB (480Mbps) physical layer which is not present on the Nucleo board. To use this feature, a fully custom hardware board would have to be developed involving the MCU and the physical layer chips.

Chapter 4

FPGA Experiments

This chapter describes all configuration and implementation details of the experimental part of this work which was conducted on the Terasic DE10-Nano board with Cyclone V SoC.

The VHDL (VHSIC Hardware Description Language) has been chosen as the default HDL for the entire project because of its preservation of a delta cycle determinism. Delta cycles are a HDL concept used to order events that occur in zero physical time. Unlike the Verilog language, the VHDL handles value update events (i.e., assignments of signals) and process evaluation events (i.e., a signal admission) in separate phases and thus explicitly preserves the determinism of synchronous architectures.

Prefix	Description
`i_`	Entity input port
`o_`	Entity output port
`io_`	Entity bidirectional (inout) port
`b_`	Entity buffer port
`g_`	Generic definition
`e_`	Entity (component) instance
`p_`	Process instance
`f_`	Function/procedure instance
`t_`	Type/subtype definition
`c_`	Constant instance
`v_`	Variable instance
`r_`	Signal instance (register) - driven by a sequential domain (i.e., process)
`w_`	Signal instance (wire) - driven by a parallel domain

Table 4.1: Prefix notation of the VHDL statements.

For the sake of the transparency of the VHDL code, the prefix notation listed in table 4.1 has been always observed, followed with uppercase instance names. No prefix with uppercase was only used for input, output and bidirectional ports of the top entity (i.e., when the ports refer to the assigned package pins), with lowercase was only used for function/procedure parameters.

Terasic provides a GHRD (Golden Hardware Reference Design) Quartus project for the DE10-Nano board, which was used as a base template for further development. It comes with correct pin assignments for all hardware parts (buttons, keys, DDR memory, etc.) on the board. It also contains important DDR memory timings in the HPS configuration and shows example usage of a few of the available IP blocks.

4.1 Clock Configuration

The board uses an on-board Texas Instruments CDCE937PWRG4 clock generator chip to provide three 50MHz clock signals fed to the FPGA fabric, also additional frequencies of 25MHz and 24MHz are generated for the HPS, USB and Ethernet peripherals. The clock generator may be configured via its I2C interface to source different clocks but it was left at the default configuration in this project.

The FPGA offers up to 6 PLL integer-N/fractional-N multipliers. For the purpose of this project, the clock frequencies listed in table 4.2 were chosen as they can be conveniently derived using a single PLL circuit and thus provide exact synchronous clock edges for all internal logic.

Name	Frequency	Purpose	Derivation
CLK1_50	50MHz	HPS, PLL, I2C	(External generator)
PLL_PRIM	320MHz	PLL primary freq.	CLK1_50 · (32/5)
CLK_BRAM	160MHz	BRAM memory clock	PLL_PRIM/2
CLK_VGA	40MHz	VGA pixel clock	PLL_PRIM/8
CLK_DCMI	26.67MHz	DCMI pixel clock	PLL_PRIM/12

Table 4.2: Summary of the FPGA clock signals.

The PLL Intel FPGA IP entity was used for the PLL circuit initialisation.

4.2 I2C Master Interface

The I2C interface is used for an initial configuration of multiple components including the CMOS imaging sensor and a VGA-to-HDMI interface. Two following approaches have been implemented for a component initialisation.

1. **HPS I2C peripheral** - The built-in HPS I2C peripheral has been routed through the FPGA fabric to the output pins associated with the MT9V034 imaging

sensor. The I2C pins of the HPS peripheral are not open-drain and are inverted by default, hence additional two ALTIOBUF IP Core bidirectional blocks for both lines (SDA and SCL) had to be added in the FPGA fabric, as shown in fig. 4.1. The I2C peripheral is memory mapped on the HPS side which runs the compiled configuration application and allows for easy changes of the camera settings.

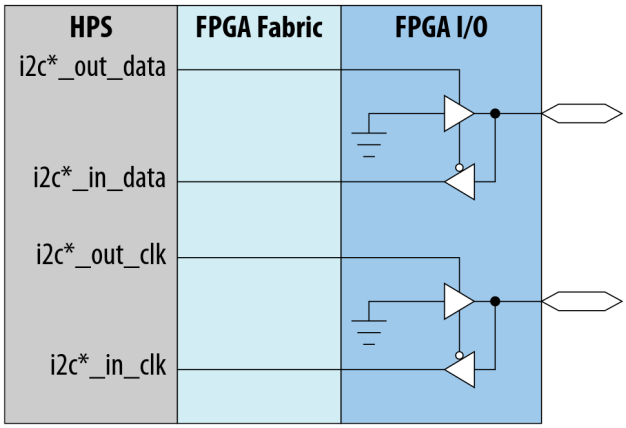


Figure 4.1: I2C wiring to the FPGA pins. [49]

2. **VHDL I2C Master interface** - The interface VHDL implementation as a finite state machine (FSM) has been taken from [50] and used for configuration of the on-board Analog Devices ADV7513 VGA-to-HDMI interface. This interface chip requires regular configuration updates whenever its interrupt output pin is triggered (e.g., when the HDMI monitor connection or resolution change occurs), thus it has to be handled instantaneously by the FPGA fabric.

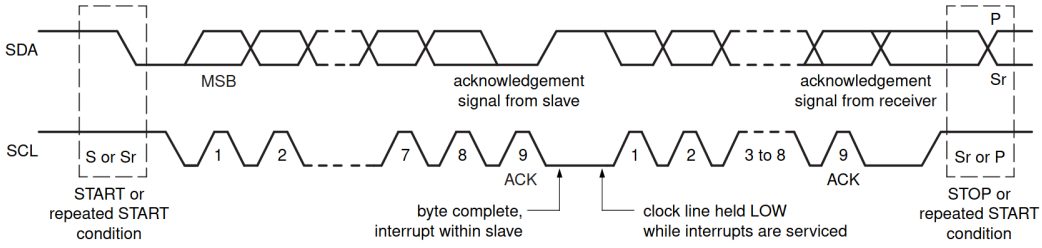


Figure 4.2: I2C data transfer diagram. [51]

The structure of the I2C data transfer is shown in fig. 4.2. The target SCL clock (400kHz) is obtained directly by dividing the input 50MHz clock by an integer prescaler value. All 9 required FSM states and their transitions are visualised in fig. 4.3. The state changes are performed on falling edges of the SCL clock, the SDA line is sampled on its rising edges. The FSM states are briefly described below:

a. State `ready` - The default state, waiting until `i_ENA` is high, then the FSM changes to the state `start`.

b. State `start` - The port `o_BUSY` is set high, the port `io_SDA` is set low (must surpass the `io_SCL` low state). Then the FSM continues to the state `command`.

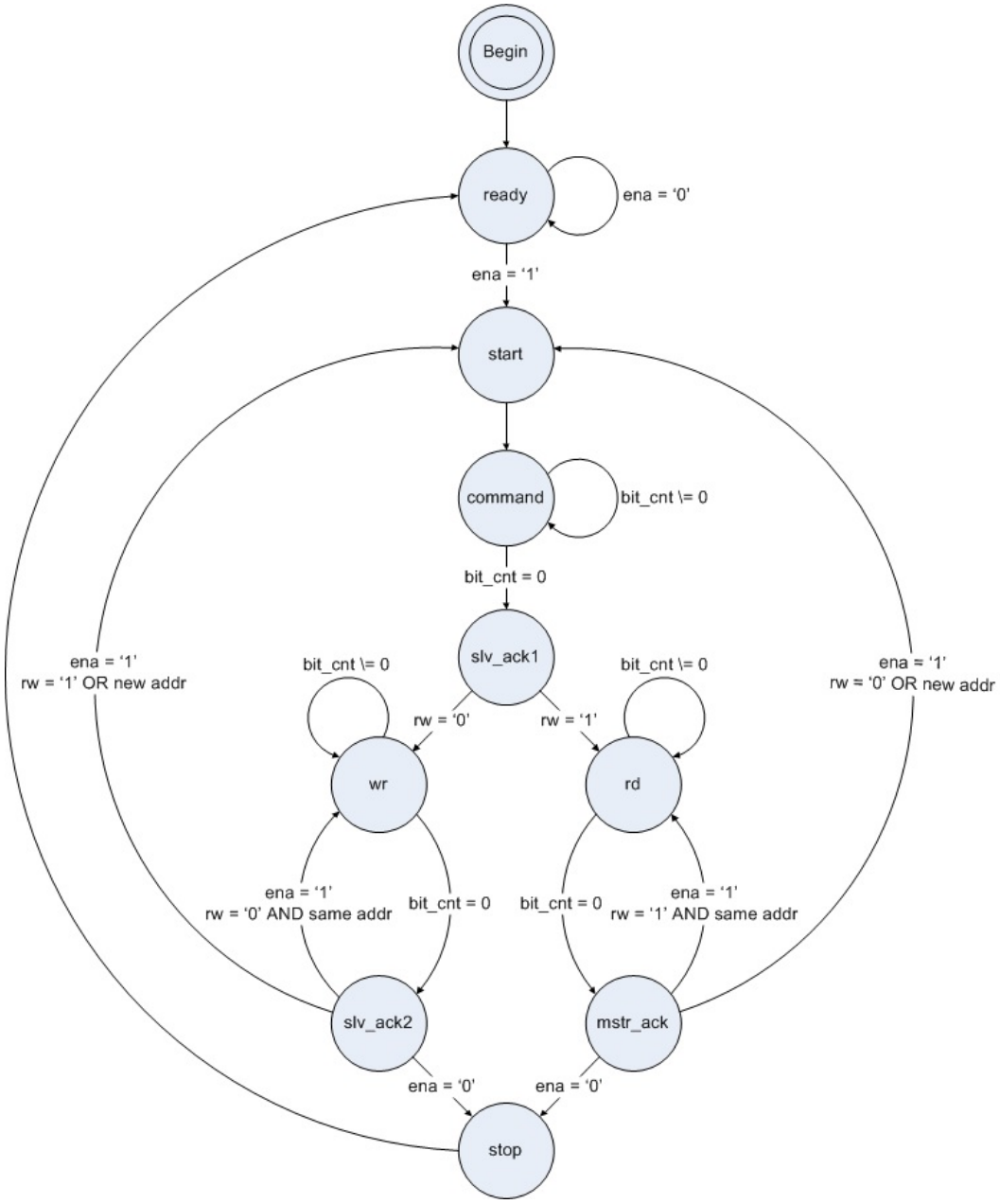


Figure 4.3: I2C master state diagram. [50]

c. State `command` - The 7 address bits plus the last read/write bit (`i_RW`) are passed to the `io_SDA` sequentially at each clock cycle (indexed by a `r_BIT_CNT` signal). Then the FSM continues to the state `slv_ack1`.

d. State `slv_ack1` - The `io_SDA` should be low when the slave device is successfully addressed. The state `wr` or the state `rd` then follows, based on

the `i_RW` signal.

e. State `wr` - All 8 bits of the `i_DATA_WR` port are sequentially passed to the `io_SDA`. Then the FSM changes to the state `slv_ack2`.

f. State `slv_ack2` - Similarly as in the `slv_ack1` state, the low state of `io_SDA` indicates a success. Also, the `o_BUSY` signal is set low. If `i_ENA` is kept high, the FSM goes back to the `start` state and next 8 bits transfer follows. Otherwise the FSM changes to the `stop` state.

g. State `rd` - All 8 bits are sequentially sampled from the `io_SDA` to the `o_DATA_RD` port. Then the FSM changes to the state `mstr_ack`.

h. State `mstr_ack` - The `o_BUSY` signal is set low and the `io_SDA` is set low to indicate that all 8 bits of data were successfully captured by the master interface. Then, similarly to the `slv_ack2` state, a next transfer may follow immediately when `i_ENA` is kept high and the FSM state goes back to `start`, otherwise the state `stop` follows.

i. State `stop` - The final FSM state. The `io_SCL` is released before the `io_SDA`. The FSM then changes to its default `ready` state.

Besides the I2C interface FSM presented above, another simple FSM had to be created to serve the interface with register addresses and values stored in a correct order in a LUT memory. The FSM states and transitions are visualised in fig. 4.4.

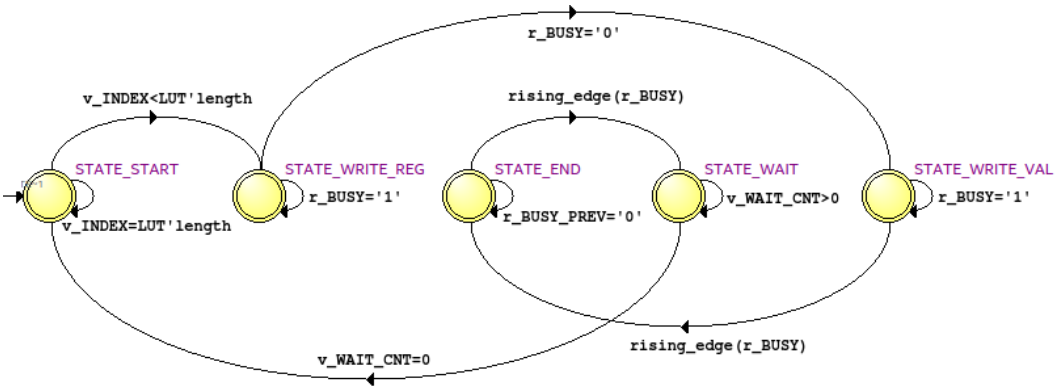


Figure 4.4: State diagram of the FSM serving the I2C interface.

The states are described as follows:

a. State `start` - The initial (default) state of the FSM. When an internal integer variable `v_INDEX` is equal to the size of the LUT and also the interrupt signal is triggered (i.e., `i_HDMI_TX_INT` is low), then the index variable is reset to zero, otherwise the index variable is incremented. If the updated variable is in a valid range, then register address and value are loaded from the LUT, and the FSM continues to the `write_reg` state.

b. State `write_reg` - The FSM waits until the I2C interface is not busy. Then the interface is enabled and the loaded register address is handed over. After that the state changes to `write_val`.

c. State `write_val` - This FSM state works similarly as in the previous case, but the interface's enable signal is kept high resulting in a continuous I2C transfer. The FSM changes to the `end` state on a falling edge of the interface's busy signal.

d. State `end` - The interface's enable signal is kept high until a falling edge of the interface's busy signal occurs. Then the FSM changes to its final `wait` state.

e. State `wait` - Because the FSM runs in a process at the 50MHz clock, additional delay cycles are needed to create sufficient time delays between the adjacent I2C transfers. When an internal decrement counter `v_WAIT_CNT` reaches zero from a predefined value, the FSM resets to the initial `start` state.

4.3 DCMI Interface

The pin assignments for the camera board were taken directly from the table 3.2 in the previous chapter. The structure of the frames transferred via the DCMI interface shown in fig. 3.4 inherently led to a design of the VHDL process treating the interface signals. The excerpt of the process code is shown below.

Code 4.1: Excerpt of the DCMI process (VHDL)

```
p_DCMI : PROCESS (w_HW_NRST, w_CLOCK_26) IS
BEGIN
 -- if hardware reset is active (i.e., key button is pressed)
 IF w_HW_NRST = '0' THEN
  -- [...] all relevant signals are reset to the default values
 ELSE
  IF rising_edge(w_CLOCK_26) THEN
   -- BRAM write enable is disabled by default
   r_BRAM_FRAME_WR_EN <= '0';

   -- simple 2-state FSM for the DCMI is handled
   CASE r_DCMI_STATE IS
    WHEN c_DCMI_STATE_IDLE =>
     -- image coordinate counters (UNSIGNED) are reset to 0
     r_DCMI_POS_X         <= (OTHERS => '0');
     r_DCMI_POS_Y         <= (OTHERS => '0');
     -- pixel buffer (STD_LOGIC_VECTOR) matching the BRAM data width
     r_DCMI_PIXELS        <= (OTHERS => '0');
     -- offset counter (UNSIGNED) for the pixel buffer
     r_DCMI_OFFSET        <= (OTHERS => '0');
     -- write address set to maximum for overflow to 0 at the start
     r_BRAM_FRAME_WR_ADDR <= (OTHERS => '1');

     -- wait for the end of the latest frame
     IF DCMI_VSYNC /= g_DCMI_VSYNC_ACTIVE THEN
      r_DCMI_STATE <= c_DCMI_STATE_SNAPSHOT;
     END IF;
```

```
28      WHEN c_DCMI_STATE_SNAPSHOT =>
29       -- wait for a rising edge of VSYNC (frame start), also crop the frame
    ↪  to its expected height (error prevention)
30       IF DCMI_VSYNC = g_DCMI_VSYNC_ACTIVE AND r_DCMI_POS_Y <
    ↪  g_DCMI_IMG_HEIGHT THEN
31         -- wait for a rising edge of HSYNC (row start), also crop the frame
    ↪  to its expected width (error prevention)
32         IF DCMI_HSYNC = g_DCMI_HSYNC_ACTIVE AND r_DCMI_POS_X <
    ↪  g_DCMI_IMG_WIDTH THEN
33           -- if the pixel buffer is almost full
34           IF r_DCMI_OFFSET = c_BRAM_FRAME_NUM_BYTES - 1 THEN
35            -- increment the BRAM address
36            r_BRAM_FRAME_WR_ADDR <=
    ↪  STD_LOGIC_VECTOR(UNSIGNED(r_BRAM_FRAME_WR_ADDR) + 1);
37            -- store a new pixel value including the buffer
38            r_BRAM_FRAME_WR_DATA <= DCMI_D(9 DOWNTO 2) &
    ↪  r_DCMI_PIXELS(r_DCMI_PIXELS'HIGH DOWNTO 8);
39            -- set BRAM write enable high
40            r_BRAM_FRAME_WR_EN <= '1';
41           ELSE
42            -- append a new pixel value to the buffer (rotate right)
43            r_DCMI_PIXELS <= DCMI_D(9 DOWNTO 2) &
    ↪  r_DCMI_PIXELS(r_DCMI_PIXELS'HIGH DOWNTO 8);
44           END IF;
45
46           -- increment x coordinate with each new pixel in the row
47           r_DCMI_POS_X  <= r_DCMI_POS_X + 1;
48           -- also increment the offset counter
49           r_DCMI_OFFSET <= r_DCMI_OFFSET + 1;
50          ELSE
51           -- check x coordinate to ensure only one execution
52           IF r_DCMI_POS_X > 0 THEN
53            -- increment the y coordinate when the row ends, reset x coord.
54            r_DCMI_POS_Y <= r_DCMI_POS_Y + 1;
55            r_DCMI_POS_X <= (OTHERS => '0');
56           END IF;
57          END IF;
58         ELSE
59           -- check y coordinate to ensure only one execution
60          IF r_DCMI_POS_Y > 0 THEN
61           -- reset the FSM state to IDLE
62           r_DCMI_STATE      <= c_DCMI_STATE_IDLE;
63           -- keep track of the number of the received image frames
64           r_DCMI_NUM_FRAMES <= r_DCMI_NUM_FRAMES + 1;
65          END IF;
66         END IF;
67       END CASE;
68      END IF;
69     END IF;
70   END PROCESS;
```

The following list summarises the key aspects of this DCMI implementation.

- Apart from the assigned signals in the excerpt above, FPGA-to-HPS interrupt request signals were also assigned to inform the HPS about changes of the FSM state (i.e., that the received image is completely stored in a BRAM memory). The HPS involvement is described in detail later in this chapter.
- The individual impedance of the DCMI signals on the custom camera board was not perfectly matched, which led to inexact match of the synchronisation signals with the data signals. Therefore additional cropping of the pixel data based on the coordinate counter values had to be added in order to ensure a correct number of stored pixels per row (and also rows per frame).
- For a similar reason, the DCMI process was driven by the rising edge of the generated pixel clock passed to the CMOS sensor instead of the `DCMI_PIXCLK` input signal.
- Only the 8 most significant bits of each 10-bit pixel intensity value were selected and stored in the BRAM memory (i.e., `DCMI_D(9 DOWNTO 2)`). The truncated pixel values were first buffered to be written in small groups, with a size matching the data width of the BRAM's write port in order to reduce the total number of write operations. This way, every complete image frame (i.e., 360960 pixels) was stored in the BRAM memory, overwriting the previous frame.

The compilation procedure in the Quartus environment requires proper definition of timing constraints stored in a Synopsys Design Constraints (SDC) file. Without the constraints, the setup and hold times of the input and output signals are not taken into account when the RTL architecture is mapped into particular parts of the FPGA fabric, which results in additional input and output signal delays and eventually in visual incompleteness and/or defects of the received images via the DCMI interface. The required setup time (TSU) and hold time (TH) were taken directly from the MT9V034 datasheet [22] and used in the SDC file as shown below.

Code 4.2: SDC timing constraints for the DCMI interface

```
1  set CLOCK_26
   ↪ {e_PLL|pll_inst|altera_pll_i|general[2].gpll~PLL_OUTPUT_COUNTER|divclk}
2
3  set DCMI_CLK_PERIOD [get_clock_info -period [get_clocks ${CLOCK_26}]]
4  set DCMI_TSU 16.0 # nanoseconds
5  set DCMI_TH  16.0 # nanoseconds
6  set DCMI_PAD 0.3  # nanoseconds
7  set DCMI_MAX [expr ${DCMI_CLK_PERIOD} - ${DCMI_TSU} - ${DCMI_PAD}]
8  set DCMI_MIN [expr ${DCMI_CLK_PERIOD} + ${DCMI_TH}  + ${DCMI_PAD}]
9
10 set_output_delay -clock [get_clocks ${CLOCK_26}] -max ${DCMI_MAX} -min
   ↪ ${DCMI_MIN} [get_ports {CAM_MASTER_CLK}]
11 set_input_delay  -clock [get_clocks ${CLOCK_26}] -max ${DCMI_MAX} -min
   ↪ ${DCMI_MIN} [get_ports {DCMI_D[*]}]
12 set_input_delay  -clock [get_clocks ${CLOCK_26}] -max ${DCMI_MAX} -min
   ↪ ${DCMI_MIN} [get_ports {DCMI_VSYNC}]
13 set_input_delay  -clock [get_clocks ${CLOCK_26}] -max ${DCMI_MAX} -min
   ↪ ${DCMI_MIN} [get_ports {DCMI_HSYNC}]
```

4.4 VGA-to-HDMI Interface

The DE10-Nano board contains Analog Devices ADV7513 HDMI transmitter [52]. The video input is provided to this chip in a standard VGA format, while the transmitter automatically recognises the resolution of the video stream based on the used VGA timing (i.e., the blanking times and lengths of the synchronisation pulses) and it supports a variety of HDTV formats. It also supports up to eight I2S audio lines for a 7.1 surround audio (up to 768kHz), however it this feature was not used in this project.

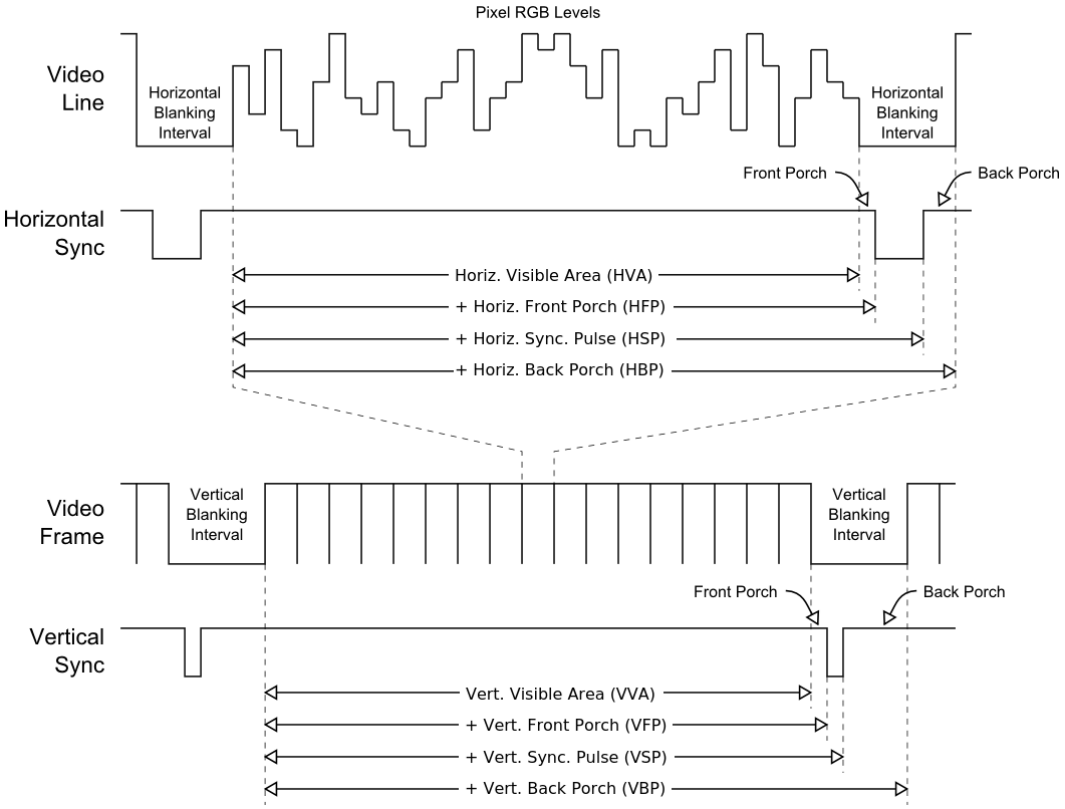


Figure 4.5: General VGA timing diagram.

The VGA timing diagram is visualised in fig. 4.5. Unlike the DCMI interface, the synchronisation signals are activated by the VGA to determine the end of every row (horizontal blanking) and of every frame (vertical blanking). The Front Porch (FP) and Back Porch (BP) delays surround the Synchronisation Pulse (SP) inside the blanking interval. The transmitter chip uses high logic state when the signals are active. It also uses Data Enable (DE) signal, which is active in the visible area of the image (both horizontal and vertical). The video line uses 24-bit RGB pixel format (i.e., 8 bits per colour).

As it was mentioned in the Clock Configuration section, the VGA pixel clock was set to 40MHz. Two different VGA timings could be used by the HDMI transmitter at this frequency as shown in table 4.3. The preferred timing was then chosen using

an on-board slide switch according to the aspect ratio of the connected display.

Visible Resolution	800x600 (SVGA)		960x540	
Visible Aspect Ratio	4:3		16:9	
Horiz. Visible Area (HVA)	800px	20μs	960px	24μs
Horiz. Front Porch (HFP)	40px	1μs	50px	1.25μs
Horiz. Sync. Pulse (HSP)	128px	3.2μs	100px	2.5μs
Horiz. Back Porch (HBP)	88px	2.2μs	90px	2.25μs
Total Frame Width	1056px	26.4μs	1200px	30μs
Vert. Visible Area (VVA)	600px	15.84ms	540px	16.2ms
Vert. Front Porch (VFP)	1px	26.4μs	2px	60μs
Vert. Sync. Pulse (VSP)	4px	105.6μs	10px	300μs
Vert. Back Porch (VBP)	23px	607.2μs	11px	330μs
Total Frame Height	628px	16.5792ms	563px	16.89ms
Real FPS value (rounded)	60.317		59.207	

Table 4.3: Selectable VGA timings for the HDMI transmitter.

The main part of the VGA process written in the VHDL language is shown below.

Code 4.3: Excerpt of the VGA process (VHDL)

```
p_VGA : PROCESS (w_HW_NRST, w_CLOCK_40) IS
BEGIN
 IF w_HW_NRST = '0' THEN
  -- [...] all relevant signals and variables are set to the default values
 ELSIF rising_edge(w_CLOCK_40) THEN
  -- HDMI transceiver signals are reset
  HDMI_TX_D  <= (OTHERS => '0');
  HDMI_TX_DE <= '0';
  HDMI_TX_HS <= '0';
  HDMI_TX_VS <= '0';
  -- BRAM read enable signal is disabled
  r_BRAM_FRAME_RD_EN <= '0';

  -- increment POS_X and POS_Y of VGA output
  IF r_VGA_POS_X < w_VGA_TOTAL_WIDTH - 1 THEN
   IF r_VGA_POS_X = 0 THEN
    IF r_VGA_POS_Y < w_VGA_TOTAL_HEIGHT - 1 THEN
     -- start of a new line (row) - increment POS_Y
     r_VGA_POS_Y <= r_VGA_POS_Y + 1;
    ELSE
     -- start of a new frame - reset POS_Y to zero
     r_VGA_POS_Y          <= (OTHERS => '0');
     r_BRAM_FRAME_RD_ADDR <= (OTHERS => '0');
    END IF;
   END IF;
```

```
26      r_VGA_POS_X <= r_VGA_POS_X + 1;
27     ELSE
28      r_VGA_POS_X <= (OTHERS => '0');
29     END IF;
30
31     -- inside the 480 lines (i.e., DCMI image height) in the frame centre
32     IF r_VGA_POS_Y >= w_VGA_MARGIN_TOP AND r_VGA_POS_Y < w_VGA_MARGIN_BOTTOM
    ↪  THEN
33      -- first read in the line
34      IF r_VGA_POS_X = w_VGA_MARGIN_LEFT - c_BRAM_FRAME_NUM_BYTES THEN
35       r_BRAM_FRAME_RD_EN <= '1';
36      END IF;
37      -- inside the 752 columns (i.e., DCMI image width) in the frame centre
38      IF r_VGA_POS_X >= w_VGA_MARGIN_LEFT AND r_VGA_POS_X < w_VGA_MARGIN_RIGHT
    ↪  THEN
39       -- load BRAM output to the variable, request next read
40       IF r_VGA_OFFSET = 0 THEN
41        v_PIXELS := r_BRAM_FRAME_RD_DATA;
42        r_BRAM_FRAME_RD_ADDR <=
    ↪   STD_LOGIC_VECTOR(UNSIGNED(r_BRAM_FRAME_RD_ADDR) + 1);
43        r_BRAM_FRAME_RD_EN   <= '1';
44       END IF;
45       -- set the lowest byte (greyscale pixel) to output RGB channels
46       HDMI_TX_D <= v_PIXELS(7 DOWNTO 0) & v_PIXELS(7 DOWNTO 0) & v_PIXELS(7
    ↪   DOWNTO 0);
47       -- [...] also, output current pixel to the FAST-like alg. signals
48       -- shift the pixel buffer right by 1 element
49       v_PIXELS(v_PIXELS'HIGH - 8 DOWNTO 0) := v_PIXELS(v_PIXELS'HIGH DOWNTO
    ↪   8);
50       -- increment current offset within the loaded bytes
51       r_VGA_OFFSET <= r_VGA_OFFSET + 1;
52      ELSE
53       r_VGA_OFFSET <= (OTHERS => '0');
54      END IF;
55     END IF;
56
57     IF r_VGA_POS_X >= 0 AND r_VGA_POS_X < w_VGA_ACTIVE_WIDTH AND r_VGA_POS_Y
    ↪   >= 0 AND r_VGA_POS_Y < w_VGA_ACTIVE_HEIGHT THEN
58      HDMI_TX_DE <= '1'; -- set the data enable signal
59     END IF;
60     IF r_VGA_POS_X >= (w_VGA_ACTIVE_WIDTH + w_VGA_H_FRONT_PORCH) AND
    ↪   r_VGA_POS_X < (w_VGA_ACTIVE_WIDTH + w_VGA_H_FRONT_PORCH +
    ↪   w_VGA_H_SYNC_PULSE) THEN
61      HDMI_TX_HS <= '1'; -- set the horizontal sync signal
62     END IF;
63     IF r_VGA_POS_Y >= (w_VGA_ACTIVE_HEIGHT + w_VGA_V_FRONT_PORCH) AND
    ↪   r_VGA_POS_Y < (w_VGA_ACTIVE_HEIGHT + w_VGA_V_FRONT_PORCH +
    ↪   w_VGA_V_SYNC_PULSE) THEN
64      HDMI_TX_VS <= '1'; -- set the vertical sync signal
65     END IF;
66    END IF;
67  END PROCESS;
```

The following list summarises the key aspects of the VGA implementation. Most of the implementation parts are not included in the code excerpt above because of the code length.

- The VHDL implementation also loads the binary values for the corresponding image pixels from two binary masks updated by the FAST process (i.e., from two additional BRAM memory instances). The masks are used to label the pixels belonging to the sun projection (i.e., the points with the sun potential) and to highlight detected marker points. The sun pixels are visualised by a yellow colour and the markers are surrounded by blue rectangles.
- The BRAM memory instance holding the DCMI frames was accessed through the same port of its dual-port interface. For this purpose, the single port was split into one read interface and one write interface. An additional process was used for organisation of the write requests (DCMI) and read requests (VGA) with a capability to remember single pending request when another request is currently being handled.
- The second port of the BRAM's dual-port interface was used by the HPS. This way, the HPS was able read the DCMI image frames and store them in real time on a SD card, it was also able to write frames loaded from stored recordings to the BRAM memory. The VGA process was then able to work the same way for both of the video sources.
- The implementation also sets FPGA-to-HPS interrupt signals to notify the HPS about the last read request of the image frame from the BRAM.
- The video frames could be paused using the second on-board key button (i.e., showing a constant image frame). If the video images were sourced by the HPS, the HPS effectively stopped/restored loading of the stored recordings from the SD card.
- The VGA process was used as a source of the pixel data for the FAST-like VHDL process, thus the FAST-like implementation was not dependent on the DCMI interface (and the camera board) and could be used entirely just with the HPS.
- For visualisation purposes, a raster ASCII font was stored in additional ROM memory, so additional information could displayed. This feature is described in the following section.

Displaying a raster font in the video output

A raster font with symbol sizes of 16x12 pixels was taken from [53]. The symbols were transformed from the C arrays to a binary representation and stored in a Memory Initialisation File (MIF), that was used for initialisation of a ROM memory. Each symbol took exactly 256 bits (32 bytes) of the memory and there were 97 symbols in total (starting with a 32th ASCII symbol used for white space), resulting

in total utilisation of 3104 bytes of the memory. The ROM memory data port width was selected as 256-bit to match the symbol size, then each ROM address corresponded exactly to one symbol and every symbol could be read during a single VGA clock cycle.

Below is a VHDL procedure used in the VGA process for drawing strings to the output video stream.

Code 4.4: VHDL procedure used for displaying fonts in the VGA output.

```
VARIABLE v_SYM_BIT : STD_LOGIC             := '0';
VARIABLE v_SYM_IDX : INTEGER RANGE 0 TO 63 := 0;
VARIABLE v_BIT_IDX : INTEGER RANGE 0 TO 12 := 0;

PROCEDURE f_DRAW_STRING (CONSTANT str : IN STRING; CONSTANT x : IN INTEGER;
↪  CONSTANT y : IN INTEGER; CONSTANT rgb : IN INTEGER) IS
BEGIN
 IF r_VGA_POS_X >= x AND r_VGA_POS_X < (x + (v_BIT_IDX'HIGH * str'LENGTH))
↪  AND r_VGA_POS_Y >= y AND r_VGA_POS_Y < (y + 16) THEN
  IF r_VGA_POS_X = x THEN
   v_SYM_IDX := 0;
   v_BIT_IDX := 0;
  END IF;
  IF v_SYM_IDX < str'LENGTH THEN
   r_ROM_FONT_ADDR <=
↪  STD_LOGIC_VECTOR(to_unsigned(CHARACTER'POS(str(v_SYM_IDX + 1)) - 32,
↪  r_ROM_FONT_ADDR'LENGTH));
   v_SYM_BIT := r_ROM_FONT_DATA(v_BIT_IDX + 16 * (to_integer(r_VGA_POS_Y) -
↪  y));
   IF v_SYM_BIT = '1' THEN
    HDMI_TX_D <= STD_LOGIC_VECTOR(to_unsigned(rgb, HDMI_TX_D'LENGTH));
   END IF;
   v_BIT_IDX := v_BIT_IDX + 1;
   IF v_BIT_IDX = v_BIT_IDX'HIGH THEN
    v_SYM_IDX := v_SYM_IDX + 1;
    v_BIT_IDX := 0;
   END IF;
  END IF;
 END IF;
END PROCEDURE;
```

The procedure must have been called inside the VGA process body. One of the calls of the VHDL procedure is shown in the code excerpt below.

Code 4.5: Call of the VHDL procedure for a string constant visualisation.

```
IF r_VIDEO_SRC_RUNNING = '1' THEN
 f_DRAW_STRING("Running", w_VGA_MARGIN_LEFT, w_VGA_MARGIN_TOP - 20,
↪  16#00FF00#);
ELSE
 f_DRAW_STRING("Stopped", w_VGA_MARGIN_LEFT, w_VGA_MARGIN_TOP - 20,
↪  16#FF0000#);
END IF;
```

Timing constraints for the HDMI transmitter signals

Similarly as for the DCMI interface, the signal inputs and outputs of the HDMI transmitter must have been constrained. The setup and hold times were taken from the ADV7513 datasheet [52] and added to the SDC file as shown below.

Code 4.6: SDC timing constraints for the HDMI transmitter (VGA input) interface

```
set CLOCK_40
 ↪ {e_PLL|pll_inst|altera_pll_i|general[1].gpll~PLL_OUTPUT_COUNTER|divclk}

set HDMI_CLK_PERIOD [get_clock_info -period [get_clocks ${CLOCK_40}]]
set HDMI_TSU 1.8 # nanoseconds
set HDMI_TH  1.3 # nanoseconds
set HDMI_PAD 0.3 # nanoseconds
set HDMI_MAX [expr ${HDMI_CLK_PERIOD} - ${HDMI_TSU} - ${HDMI_PAD}]
set HDMI_MIN [expr ${HDMI_CLK_PERIOD} + ${HDMI_TH}  + ${HDMI_PAD}]

set_output_delay -clock [get_clocks ${CLOCK_40}] -max ${HDMI_MAX}
 ↪ [get_ports {HDMI_TX_CLK}]
set_output_delay -clock [get_clocks ${CLOCK_40}] -min ${HDMI_MIN}
 ↪ [get_ports {HDMI_TX_CLK}]
set_output_delay -clock [get_clocks ${CLOCK_40}] -max ${HDMI_MAX}
 ↪ [get_ports {HDMI_TX_D[*]}]
set_output_delay -clock [get_clocks ${CLOCK_40}] -min ${HDMI_MIN}
 ↪ [get_ports {HDMI_TX_D[*]}]
set_output_delay -clock [get_clocks ${CLOCK_40}] -max ${HDMI_MAX}
 ↪ [get_ports {HDMI_TX_DE}]
set_output_delay -clock [get_clocks ${CLOCK_40}] -min ${HDMI_MIN}
 ↪ [get_ports {HDMI_TX_DE}]
set_output_delay -clock [get_clocks ${CLOCK_40}] -max ${HDMI_MAX}
 ↪ [get_ports {HDMI_TX_HS}]
set_output_delay -clock [get_clocks ${CLOCK_40}] -min ${HDMI_MIN}
 ↪ [get_ports {HDMI_TX_HS}]
set_output_delay -clock [get_clocks ${CLOCK_40}] -max ${HDMI_MAX}
 ↪ [get_ports {HDMI_TX_VS}]
set_output_delay -clock [get_clocks ${CLOCK_40}] -min ${HDMI_MIN}
 ↪ [get_ports {HDMI_TX_VS}]
set_input_delay  -clock [get_clocks ${CLOCK_40}] -max ${HDMI_MAX}
 ↪ [get_ports {HDMI_TX_INT}]
set_input_delay  -clock [get_clocks ${CLOCK_40}] -min ${HDMI_MIN}
 ↪ [get_ports {HDMI_TX_INT}]
```

Final visualisation sample

A sample frame of the HDMI video output (800x600 pixels) is shown in fig. 4.6. The image frames were sourced by the HPS from a recording stored in the SD card. The visualisation shows the current video frame colourised using the binary masks from the FAST-like algorithm process and a following textual information: a running/stopped state of the video source, the current frame number, an average HDMI output FPS value calculated by the VGA process, an elapsed time since the

board power-up, a number of the detected markers and a number of the detected sun points (ordered from the top-left image corner).

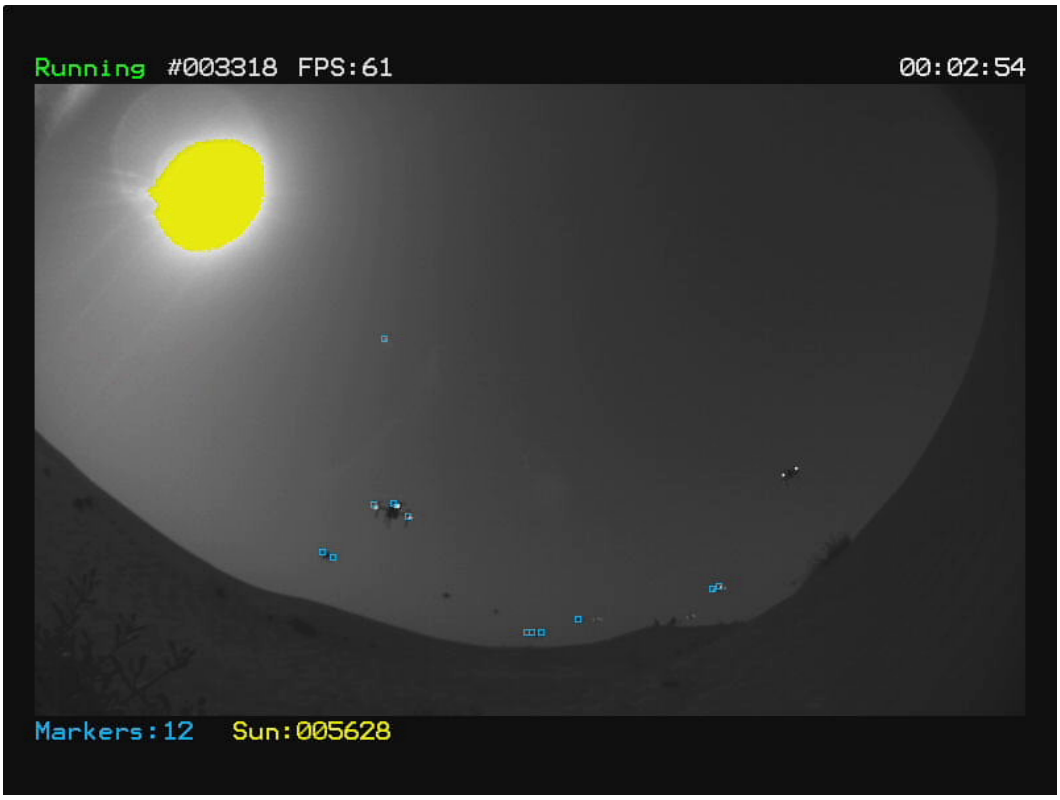


Figure 4.6: A sample frame captured from the HDMI video output.

4.5 FAST-like Algorithm

The architecture for the VHDL implementation of the FAST-like algorithm has been already presented in the chapter 2.

VHDL subtypes `t_PIX_INT`, `t_SHIFT_INT` and `t_COORD_INT` are defined as integers with ranges sufficient for the contained values. Subtype `t_PIX_VEC` is defined as a `STD_LOGIC_VECTOR` with a size matching its integer counterpart. VHDL types with `t_ARR_` prefix are defined as arrays of the subtypes (variable lengths).

Two similar VHDL entities were defined for both radii of 3 and 4 pixels of the algorithm's neighbourhood. The entity definition for the radius of 3 pixels is shown in the code excerpt below.

Code 4.7: VHDL entity of the FAST-like algorithm using radius of 3 pixels.

```
1 ENTITY fast_r3 IS
2  GENERIC (
3   g_RAM_NUM : t_SHIFT_INT := 9; -- defined by the largest used radius
4   g_SIZE    : t_SHIFT_INT := 7  -- (or 9 for the radius of 4 pixels)
5  );
```

```
 PORT (
  i_CLOCK : IN STD_LOGIC;                 -- clock signal input
  i_NRST  : IN STD_LOGIC;                 -- reset signal input
  i_THRESHOLD  : IN t_PIX_INT    := 120; -- threshold value signal input
  i_DATA       : IN t_ARR_PIX_VEC(0 TO g_RAM_NUM - 1) := (OTHERS => (OTHERS
 ↪  => '0')); -- array input containing the pixels of the current column
  i_ROW_SHIFT  : IN t_SHIFT_INT  := 0;    -- row shift signal input
  o_DET_COL    : OUT t_SHIFT_INT := 0;    -- detection column output
  o_DET_ROW    : OUT t_SHIFT_INT := 0;    -- detection row output
  o_SUN_POT    : OUT STD_LOGIC   := '0'; -- sun potential output
  o_MARKER_POT : OUT STD_LOGIC   := '0'; -- marker potential output
  o_VALID      : OUT STD_LOGIC   := '0'   -- valid signal output
);
END ENTITY;
```

The RTL architecture of this entity implements the following auxiliary signals.

Code 4.8: VHDL RTL architecture signals of the FAST-like implementation.

```
-- internal LUTs used to store the intermediate results
SIGNAL r_CENTER_ARR      : t_ARR_PIX_INT(0 TO g_SIZE-1)   := (OTHERS => 0);
SIGNAL r_MAX_BOUND_ARR   : t_ARR_PIX_INT(0 TO g_SIZE-1)   := (OTHERS => 0);
SIGNAL r_MIN_BOUND_ARR   : t_ARR_PIX_INT(0 TO g_SIZE-1)   := (OTHERS =>
 ↪  t_PIX_INT'HIGH);
SIGNAL r_MAX_INT_ARR     : t_ARR_PIX_INT(0 TO g_SIZE-1)   := (OTHERS => 0);
SIGNAL r_MAX_INT_COL_ARR : t_ARR_SHIFT_INT(0 TO g_SIZE-1) := (OTHERS => 0);
SIGNAL r_MAX_INT_ROW_ARR : t_ARR_SHIFT_INT(0 TO g_SIZE-1) := (OTHERS => 0);
-- signals containing threshold-related and coordinate-related values
SIGNAL r_THR_HALF       : t_PIX_INT   := 0; -- half of the threshold input
SIGNAL r_THR_X2         : t_PIX_INT   := 0; -- double of the threshold
 ↪  input
SIGNAL r_ROW_SHIFT_PREV : t_SHIFT_INT := g_SIZE;
CONSTANT c_CENTER_POS   : t_SHIFT_INT := g_SIZE / 2;
```

The following procedures are used in the VHDL process for calculation of the column and row indices.

Code 4.9: VHDL procedures used in the FAST-like implementation.

```
-- procedure for calculation of the column index
PROCEDURE f_GET_COL_INDEX (
 VARIABLE shift     : IN t_SHIFT_INT;
 CONSTANT column    : IN t_SHIFT_INT;
 VARIABLE col_index : OUT t_SHIFT_INT
) IS
 VARIABLE i : t_SHIFT_INT;
BEGIN
 i := shift;
 IF column > shift THEN
  i := i + g_SIZE;
 END IF;
 col_index := i - column;
END PROCEDURE;
```

```
15  -- procedure for calculation of the row index
16  PROCEDURE f_GET_ROW_INDEX (
17   SIGNAL shift       : IN t_SHIFT_INT;
18   CONSTANT row       : IN t_SHIFT_INT;
19   VARIABLE row_index : OUT t_SHIFT_INT
20  ) IS
21   VARIABLE i : t_SHIFT_INT;
22  BEGIN
23   i := shift + row;
24   IF i >= g_RAM_NUM THEN
25    i := i - g_RAM_NUM;
26   END IF;
27   row_index := i;
28  END PROCEDURE;
```

The main VHDL process also uses several auxiliary variables to store intermediate results. The process runs synchronously at `rising_edge(i_CLOCK)` and the VHDL code can be separated into the following parts based on their purpose.

1. **The initial part**. Most importantly, the pixels from the column array are organised into an auxiliary LUT variable `v_PIXELS` in the correct order.

Code 4.10: Initial part of the main VHDL process.

```
1   -- column shift reset on a new row
2   IF r_ROW_SHIFT_PREV /= i_ROW_SHIFT THEN
3    v_PROCESS_RES := '0';
4    v_COL_SHIFT   := 0;
5   END IF;
6   r_ROW_SHIFT_PREV <= i_ROW_SHIFT;
7
8   -- reorder the column pixels
9   FOR r IN 0 TO g_SIZE - 1 LOOP
10   f_GET_ROW_INDEX(i_ROW_SHIFT, r, v_ROW_INDEX);
11   v_PIXELS(r) := to_integer(UNSIGNED(i_DATA(v_ROW_INDEX)));
12  END LOOP;
```

2. **The comparison part.** In the following excerpt, only the comparisons related to the second column of the FAST image patch are shown. All columns are processed in a similar manner.

Code 4.11: Comparison part (excerpt) of the main VHDL process.

```
1   -- get correct array column index for column 1
2   f_GET_COL_INDEX(v_COL_SHIFT, 1, v_COL_INDEX);
3
4   -- load current LUT values to auxiliary variables
5   v_MAX_BOUND_VAL := r_MAX_BOUND_ARR(v_COL_INDEX);
6   v_MIN_BOUND_VAL := r_MIN_BOUND_ARR(v_COL_INDEX);
7   v_MAX_INT_VAL   := r_MAX_INT_ARR(v_COL_INDEX);
8   v_MAX_INT_COL   := r_MAX_INT_COL_ARR(v_COL_INDEX);
9   v_MAX_INT_ROW   := r_MAX_INT_ROW_ARR(v_COL_INDEX);
10
```

```
-- obtain current column comparison results
IF v_PIXELS(1) > v_MAX_BOUND_VAL THEN
    v_MAX_BOUND_VAL := v_PIXELS(1);
END IF;
IF v_PIXELS(1) < v_MIN_BOUND_VAL THEN
    v_MIN_BOUND_VAL := v_PIXELS(1);
END IF;
IF v_PIXELS(2) > v_MAX_INT_VAL THEN
    v_MAX_INT_VAL := v_PIXELS(2);
    v_MAX_INT_COL := 1;
    v_MAX_INT_ROW := 2;
END IF;
IF v_PIXELS(3) > v_MAX_INT_VAL THEN
    v_MAX_INT_VAL := v_PIXELS(3);
    v_MAX_INT_COL := 1;
    v_MAX_INT_ROW := 3;
END IF;
IF v_PIXELS(4) > v_MAX_INT_VAL THEN
    v_MAX_INT_VAL := v_PIXELS(4);
    v_MAX_INT_COL := 1;
    v_MAX_INT_ROW := 4;
END IF;
IF v_PIXELS(5) > v_MAX_BOUND_VAL THEN
    v_MAX_BOUND_VAL := v_PIXELS(5);
END IF;
IF v_PIXELS(5) < v_MIN_BOUND_VAL THEN
    v_MIN_BOUND_VAL := v_PIXELS(5);
END IF;

-- store column comparison results to the LUTs
r_MAX_BOUND_ARR(v_COL_INDEX)   <= v_MAX_BOUND_VAL;
r_MIN_BOUND_ARR(v_COL_INDEX)   <= v_MIN_BOUND_VAL;
r_MAX_INT_ARR(v_COL_INDEX)     <= v_MAX_INT_VAL;
r_MAX_INT_COL_ARR(v_COL_INDEX) <= v_MAX_INT_COL;
r_MAX_INT_ROW_ARR(v_COL_INDEX) <= v_MAX_INT_ROW;
```

3. **The final (result processing) part.** When one full cycle of columns (of the size of the image patch) passes, the final decisions about the candidate point potentials are made.

Code 4.12: Final evaluation part of the main VHDL process.

```
-- increase column shift (modulo the patch size)
v_COL_SHIFT := v_COL_SHIFT + 1;
IF v_COL_SHIFT = g_SIZE THEN
 v_COL_SHIFT   := 0;
 v_PROCESS_RES := '1'; -- enable result processing
END IF;

-- process results if the LUT values are ready
IF v_PROCESS_RES = '1' THEN
 -- load LUT values to the auxiliary variables
 v_CENTER_VAL    := r_CENTER_ARR(v_COL_SHIFT);
```

```
12    v_MAX_BOUND_VAL := r_MAX_BOUND_ARR(v_COL_SHIFT);
13    v_MIN_BOUND_VAL := r_MIN_BOUND_ARR(v_COL_SHIFT);
14    v_MAX_INT_COL   := r_MAX_INT_COL_ARR(v_COL_SHIFT);
15    v_MAX_INT_ROW   := r_MAX_INT_ROW_ARR(v_COL_SHIFT);
16
17    -- evaluate the loaded values
18    IF v_CENTER_VAL > i_THRESHOLD THEN
19     IF v_MAX_BOUND_VAL <= v_CENTER_VAL AND (v_CENTER_VAL -
      ↪  v_MAX_BOUND_VAL) >= r_THR_HALF THEN
20      -- the marker potential was preserved
21      r_MARKER_POT <= '1';
22      r_SUN_POT    <= '0';
23      r_DET_COL    <= v_MAX_INT_COL;
24      r_DET_ROW    <= v_MAX_INT_ROW;
25      r_VALID      <= '1';
26     ELSE
27      r_MARKER_POT <= '0';
28      IF v_CENTER_VAL > r_THR_X2 AND (v_MIN_BOUND_VAL >= v_CENTER_VAL OR
      ↪  (v_CENTER_VAL - v_MIN_BOUND_VAL) <= r_THR_HALF) THEN
29       -- the sun potential was preserved
30       r_DET_COL <= c_CENTER_POS;
31       r_DET_ROW <= c_CENTER_POS;
32       r_SUN_POT <= '1';
33       r_VALID   <= '1';
34      ELSE
35       r_VALID <= '0';
36      END IF;
37     END IF;
38    ELSE
39     r_MARKER_POT <= '0';
40     r_SUN_POT    <= '0';
41     r_VALID      <= '0';
42    END IF;
43
44    -- reset the LUT values to defaults
45    r_MAX_BOUND_ARR(v_COL_SHIFT)   <= 0;
46    r_MIN_BOUND_ARR(v_COL_SHIFT)   <= t_PIX_INT'HIGH;
47    r_MAX_INT_ARR(v_COL_SHIFT)     <= 0;
48    r_MAX_INT_COL_ARR(v_COL_SHIFT) <= 0;
49    r_MAX_INT_ROW_ARR(v_COL_SHIFT) <= 0;
50   END IF;
```

The VHDL code of the middle (comparison) part was generated using a parameterisable Python script taking the neighbourhood radius as one of its parameters. This is the only part where the architectures of the both entities (for the two radii) differ, the other parts of the process remain the same.

As mentioned in the previous section, both entities are driven synchronously with the VGA process and use the 9 line buffers (LUTs) that are filled by pixels obtained from the BRAM memory containing the full image frame. Thus one more process running at the VGA clock is defined inside the top entity, which takes care of:

- Storing the pixel values provided by the VGA process into the 9 line buffers (i.e., 9 single-port BRAM instances), keeping the FAST-like entities at reset until a sufficient number of line buffers is ready.
- Keeping track of the current pixel coordinates in order to compute the image frame coordinates of the valid detected points.
- Filtering the valid detected points based on their mutual distances using the previous detected point coordinates stored in an additional LUT. This step removes possible detection duplicates emerging from single or both of the entities.
- Providing the filtered detection results to the HPS and notifying it using FPGA-to-HPS interrupt signals.
- Drawing the filtered detection results into the two binary masks used for visualisation (as described in the previous section).

4.6 Hard Processor System (HPS)

The Cyclone V SoC HPS primarily contains a MPU (Microprocessor Unit) subsystem with two ARM Cortex-A9 cores, subsystems for memory controllers (SDRAM, FLASH and on-chip RAM), PLLs, interface peripherals and HPS-FPGA interfaces, as shown in fig. 4.7. Both HPS and FPGA portions of the device have separate power supplies, thus the HPS can run independently on the power state of the FPGA (however the FPGA can be used only when the HPS is running). [54]

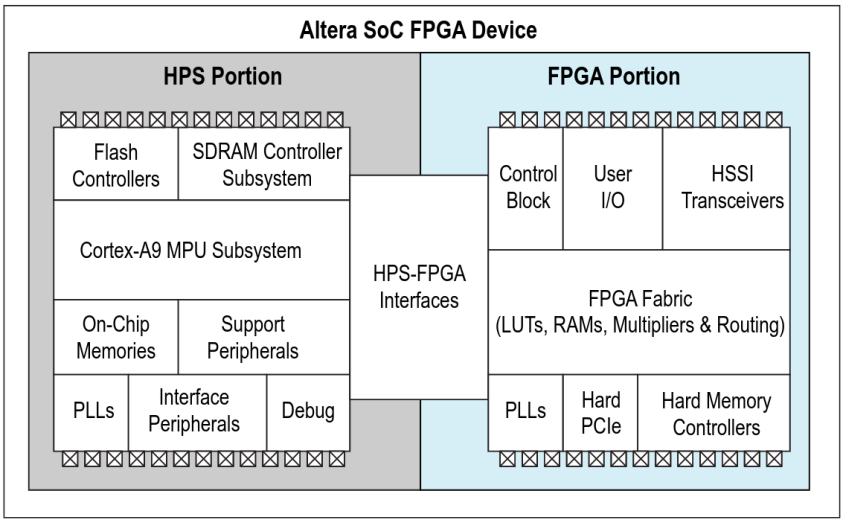


Figure 4.7: Altera SoC FPGA Device Block Diagram. [54]

The FPGA configuration scheme is determined by a 6-pin DIP switch placed on the DE10-Nano board. By factory default, the FPGA is configured to be flashed with a pre-programmed FPGA design image from an EPCS (a flash memory) device,

that can be changed using the USB-Blaster on-board programmer. For this project, I decided to set all positions of the DIP switch to "enabled" state, which means the FPGA is configured from the HPS software, i.e., an compiled FPGA design image is loaded from the SD card during boot time of the HPS. Also, the FPGA can be still reprogrammed via JTAG using the USB-Blaster programmer during run time.

4.6.1 Initial Setup

The firmware for the MPU subsystem can be built in the two most common ways:

1. As a "Bare Metal" - This approach starts with compiling a Minimal Preloader (MPL) provided in the Altera SoC Embedded Software and Tools (EDS) Suite using an included cross-compiler (which is a port of `arm-altera-eabi-gcc`). The EDS also contains Altera's hardware libraries enabling a direct management of the HPS peripherals using the memory-mapped registers, which can be compiled together with the user application source code.

2. As a Linux Operating System - The approach is similar to the Bare Metal option, it consists of compiling a U-Boot Linux boot loader [55], a Linux kernel for SoC-FPGA devices [56] and the preferred Linux distribution for the root file system (e.g., Debian or Arch Linux) using a common ARM cross compiler with enabled hardware floating point support (i.e., `arm-linux-gnueabihf-gcc`).

For both options, all the cross-compiled parts must be placed into correctly formatted partitions of the selected boot media (e.g., the SD card or the FLASH memory).

I chose the latter option because of the rich set of peripheral drivers in the Linux kernel (e.g., for the Ethernet or I2C) and also because of the simplicity of compilation of the user space applications directly in the device without a need for the cross-compiler. Additional applications can be installed easily using the distribution's package manager (such as `apt`) while accessing the OS through, for example, a local SSH connection. These benefits, however, come at the cost of leaving direct control over the computational resources and over several of the memory-mapped peripherals to the Linux kernel, mainly to its unpredictable scheduler [57] and its memory manager.

A simple boot media image-building script was written, that performed the following steps:

1. Cloning of the latest versions of the repositories, [55] and [56] (i.e., U-Boot *v2022.10* and kernel *socfpga-5.19* at the time of the build).

2. Performance of all required modifications to the cloned files, e.g., a new device MAC address was generated, the boot command was edited to process a boot script which loads the FPGA design during each HPS boot, etc.

3. Creation of a Debian root file system of the latest *bullseye* distribution version using a `debootstrap` utility. Using a `chroot` session inside the created Debian

file system, all needed modifications (an Ethernet configuration, a SSH access, etc.) were made, including an installation of the `apt` packages.

4. Using a new `makefile`, both cloned repositories together with a custom boot script and with a kernel Device Tree Source (DTS) file were cross-compiled. The primary purpose of Device Tree in Linux is to provide a way to describe non-discoverable hardware and its configuration, such as the HPS peripherals, the PLLs or the FPGA-HPS bridges. Along with the boot loader, a secondary program loader (SPL) was also compiled. Unlike the primary program loader, which loads a code from the ROM memory, the SPL loads the actual boot loader and then the kernel from the boot media partitions.

5. A new virtual boot media image file was created and mounted, containing 3 partitions with the following boot order:

   a. U-Boot and SPL partition with a size of 1MiB and a custom Altera file system type. The SPL in a binary form was placed into this partition.

   b. Kernel and Device Tree partition with a size of 254MiB and a `fat32` file system type. Apart from the compiled DTS file and the Linux kernel files, the compiled boot script and the compiled FPGA design raw binary file (RBF) from Quartus were copied into this partition.

   c. Root file system partition consuming the remaining space of the image file and with a `ext4` file system type. The complete pre-configured Debian distribution was copied into this partition.

The boot media image file created by the script was flashed to a microSD card. This concluded the initial HPS setup.

4.6.2 FPGA-HPS Communication using AXI/Avalon Interfaces

In the Quartus software, the VHDL entity for the HPS was created by the Platform Designer (QSYS) tool. The input and output ports of the entity are given by the used intellectual property (IP) blocks and by the Arria V/Cyclone V Hard Processor System IP block configuration. The HPS block was configured with the correct DDR memory timings (taken from the GHRD project). The HPS-FPGA bridges and the HPS peripherals were enabled and/or disabled for the needs of this project using this IP block.

All interfaces for a direct FPGA-HPS communication are built upon the AXI and the Avalon interfaces. Both of the interfaces are compatible in their design, they use a master-slave bus topology and offer variable address and data bus sizes. The AXI interface is used as a universal bus to connect any utilised IP blocks to several unidirectional AXI bridges, namely HPS-to-FPGA (32-/64-128-bit), FPGA-to-HPS (32-/64-128-bit), lightweight 32-bit HPS-to-FPGA and up to six SDRAM FPGA-to-HPS (i.e., directly to the DDR memory, 32-/64-/128-/256-bit). All the FPGA-HPS interfaces are memory-mapped on the HPS side, i.e., the components connected to the preferred AXI bridge can be accessed from the HPS software using the bridge's fixed memory address and the component's address offset.

4.6.3 Input/Output IP blocks

The PIO (Parallel Input/Output) Intel FPGA IP blocks were used for data sharing between the FPGA and the HPS. One input and one output 32-bit blocks were used to share basic configuration data (such as the running/stopped state of the video source or the HPS software states), one output 32-bit block was used by the HPS to set the threshold value of the FAST-like algorithm FPGA implementation from the software and one input 32-bit block was used to transfer detection coordinates and potential bits to the HPS software. All the blocks were connected to the lightweight 32-bit HPS-to-FPGA bridge which is accessible at a memory address `0xFF200000`.

4.6.4 Handling FPGA-to-HPS Interrupts

Apart from the aforementioned AXI bridges, the HPS also offers two 32-bit FPGA-to-HPS interrupt vectors to be triggered from the FPGA fabric (i.e., 64 `f2h_irqX` VHDL signals). In the HPS partition, these vectors are connected to the ARM Generic Interrupt Controller (GIC). The GIC is a part of the MPU and also processes interrupt signals from the HPS peripherals. It enables the HPS firmware code to be interrupted by asynchronous hardware events without a need for any CPU-blocking mechanism (for checking of state changes).

In this project, the Linux kernel was used and a kernel module had to be developed and inserted during the boot time to make the interrupts available to the user space software. The GIC and the associated interrupts had to be enabled in the kernel DTS file.

The software then used the common `poll` method to catch the asynchronous interrupts from the FPGA fabric, which was especially useful, e.g., when registering the states of the VHDL process for the VGA output, because the next image frame of the stored video recording could be immediately transferred using a DMA controller to the BRAM memory as soon as the VGA process processed the previous image frame.

4.6.5 Sharing Memory Resources with the FPGA

Three different ways of accessing the memory resources of the FPGA from the HPS (such as the BRAM) and the HPS memory resources from the FPGA (such as the DDR memory) in the Cyclone V SoC are presented below.

1. Via a DMA Controller FPGA IP block - In the Platform Designer, this IP block has a control Avalon slave port, a write Avalon master port and a read Avalon master port. The control port may be connected to the HPS-to-FPGA bridge or to the lightweight bridge, and is configured through its memory-mapped registers at the corresponding address by the HPS software. The read port is connected either to the FPGA-to-HPS AXI slave or to the SDRAM AXI slave, the write port is connected to a slave port of the BRAM instance initialised by a On-Chip Memory (RAM or ROM) Intel FPGA IP block in the Platform Designer. This way, the content of a DDR memory is transferred to the BRAM memory

on a request from the HPS software, the opposite data transfer direction can be performed by switching the read and write ports. There is no way to asynchronously initiate a DMA transfer from the FPGA side.

2. Via an ARM DMA-330 Controller inside the MPU - The On-Chip Memory block's slave port is connected directly to the HPS-to-FPGA bridge. The DMA-330 controller is configured through its own registers by the HPS software and can access the connected BRAM memory directly at a memory address `0xFFFF0000`. A DMA transfer can be initiated from the FPGA side using a WFP (Wait For Peripheral) instruction of the controller that must be stored in the controller's microcode, which is prepared by the HPS software beforehand. The DMA interrupt is triggered from the FPGA side using DMA peripheral requests enabled in the HPS block (i.e., up to 8 `f2h_dma_reqX` VHDL signals) and the DMA transfers work bidirectionally.

3. Via a custom Platform Designer component written in a SystemVerilog language - A custom AXI master/slave interface needs to be written and all the AXI interface signals need to be handled manually. The FPGA side has a full control over the SDRAM memory transfers in this case, the custom block is invisible to the HPS without additional control circuitry.

After trying all the aforementioned options, I decided to use the ARM DMA-330 controller and I used a control hardware library included in the Altera EDS Suite. Because I was not able to get the DMA request peripherals working, I used the FPGA-to-HPS interrupts to start the execution of the DMA microcode from the HPS software.

To be able to use any DMA controller to access the DDR memory, a coherent page-aligned buffer in the kernel space had to be allocated. The Linux OS used a page size of 4kB (0x1000 in hexadecimal) on 32-bit systems, which can be also determined by the type of the DDR3 memory chip used. To make the allocated buffer memory-mappable by the user space software, a publicly available kernel module u-dma-buf [58] was utilised.

The On-Chip RAM (BRAM) memory was initialised with a 128-bit (16B) data width, which is also the maximum data width of the HPS-to-FPGA interface. The size of 360960 bytes was expanded to a page-aligned size to achieve the fastest DMA transfers, thus 364544 bytes (0x59000 in hexadecimal) of the BRAM resources were seized. The u-dma-buf kernel driver was configured by the kernel DTS file to allocate a buffer of the same size in the DDR RAM memory during the boot time.

4.6.6 Software Description

The user space software was written completely in C language and it was configured to be started by a `systemd` service in a detached `tmux` session after the OS boots. When the `tmux` application is used, the terminal task in which the software is executed can be accessed from any SSH connection and does not terminate automatically when detached.

Several terminal commands were defined to control the application's mode:

1. Grabbing the camera images - When the application was put into this mode, the FPGA-to-HPS interrupts from the DCMI process were registered as soon as the camera image was completely stored in the BRAM memory. Then the DMA transfer was executed to transfer the camera image from the on-chip BRAM to the image buffer in the DDR RAM memory. The images could be stored into the file system in a raw binary form or compressed using a JPEG library.

2. Playing the stored camera recordings - The application was able to read the camera images from the file system to the image buffer, that was transferred to the on-chip BRAM memory using a DMA transfer as soon as an interrupt from the VGA process was registered. It also provided the frame number via the PIO output component to the VGA process and the control output PIO component was used to temporarily disable the DCMI process.

The application also stored the incoming detection results for their later comparison with the results from the current ROS implementation.

4.6.7 Complete Platform Designer Project

The complete Platform Designer project is shown in fig. 4.8. Most of the components are described in the previous sections.

The On-Chip Memory component's Avalon slave port is routed to the FPGA fabric via an External Bus to Avalon Bridge IP block. The external bus signals are the address bus, write and read buses, read and write enable signals, byte enable signals and an acknowledge signal. Except the byte enable signals and the acknowledge signal, the interface can be operated in the same way as a port of a BRAM instance initialised by an Altera 1-port/2-port RAM IP block in Quartus. All bytes were enabled as all 16 bytes were always read/written at once, the acknowledge bit is used by the external bus interface to notify the FPGA fabric when read/write operations are finished and the data is available on the data bus.

All the memory-related components were driven by the 160MHz clock signal from the PLL circuit, the HPS and the PIO components were driven by the default 50MHz clock.

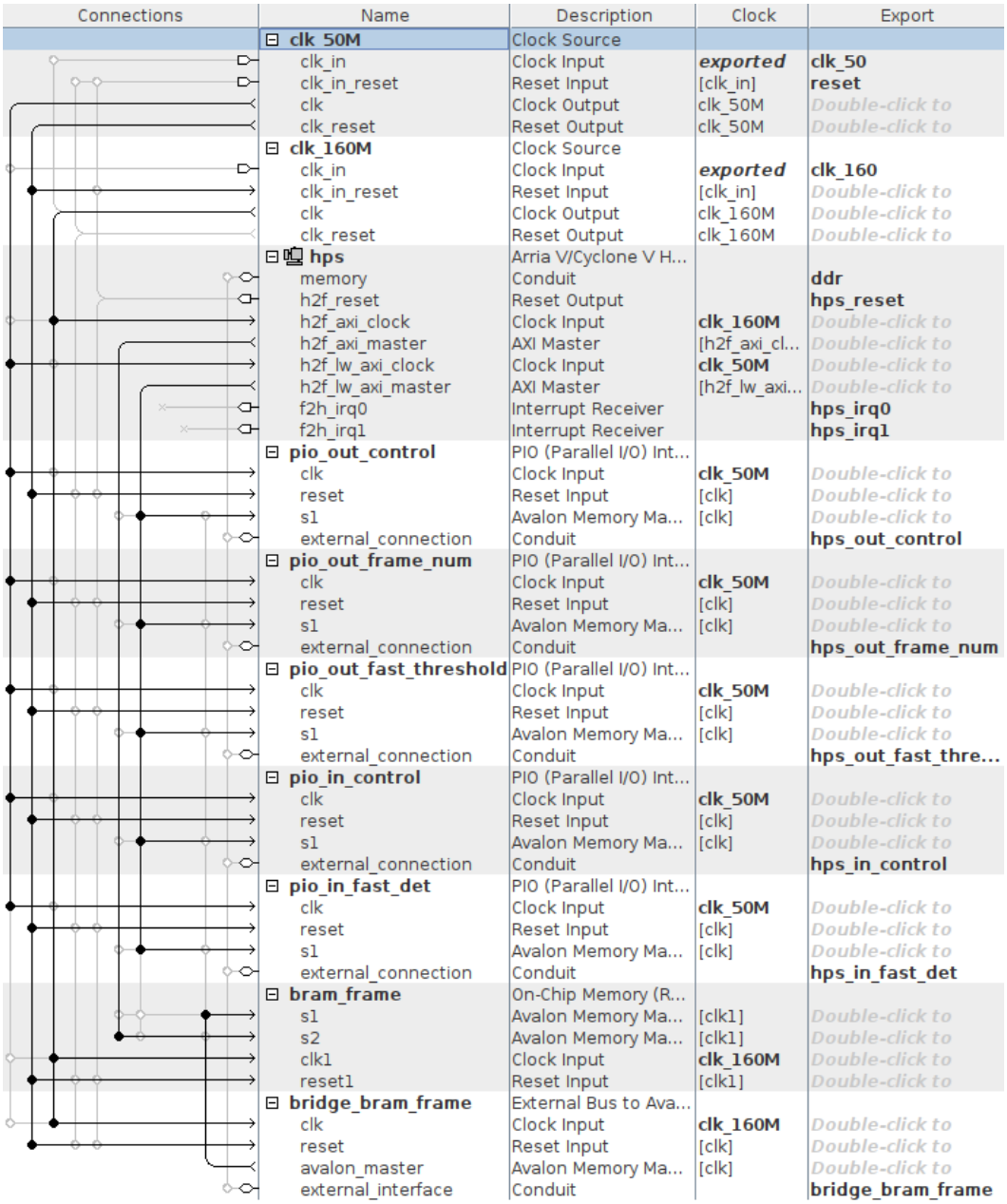


Figure 4.8: Complete platform designer project.

4.7 Total Resources Utilisation

The total utilisation of the FPGA resources was taken from the Fitter summary in the Quartus compilation report and is shown in table 4.4.

Resource name	Total usage
Logic utilization (in ALMs)	16479 / 41910 (39%)
Total registers	8686
Total pins	138 / 314 (44%)
Total block memory bits	3717248 / 5662720 (66%)
Total RAM Blocks (M10K)	496 / 553 (90%)
Total DSP Blocks	102 / 112 (91%)
Total PLLs	1 / 6 (17%)
Total DLLs	1 / 4 (25%)

Table 4.4: Total FPGA resources utilisation.

4.8 Complete Project Structure

A block diagram for the complete FPGA and HPS project structure is shown in fig. 4.9. Several less important interfaces and entities were omitted for simplification of the block diagram.

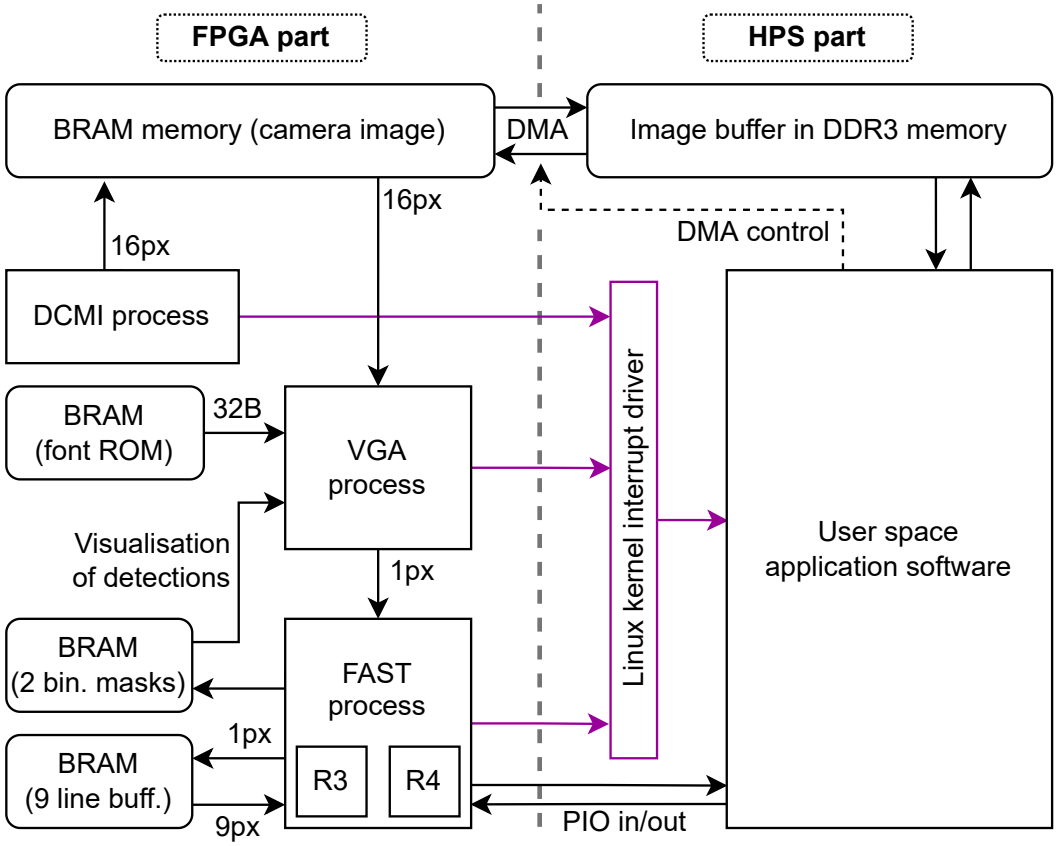


Figure 4.9: Complete project structure block diagram.

Chapter 5

Discussion & Future Work

5.1 Shortcomings of the FPGA Implementation

As it can be seen in the table 4.4, the FPGA implementation almost depleted all BRAM resources due to the storage of whole camera images, which are however needed for the visualisation by the VGA process.

At first, this project was designed without the HPS part in mind and thus only minimal design changes were then performed when the HPS was added to the project. Nonetheless, if a custom component interfacing the SDRAM Controller of the HPS was implemented, the DMA controller could be omitted from the project completely and the DDR RAM memory could be accessed from the FPGA fabric the same way as the BRAM memory. This would release 384 M10K blocks (about 69% of the total BRAM size). Thus, this implementation change is considered as an important step in the future.

5.2 Feasible HT4D FPGA Implementation

The current 4D Hough Transform algorithm cannot be directly used for a FPGA implementation due to its untractable memory requirements. However, another feasible approach was partially tested with a Python implementation, which does not require the Hough space to be stored in the memory. The hybrid mask sizes are limited in their size (as described in chapter 2 and as shown in fig. 1.9) and when applied to overly distinct t-points, they cannot overlap in the Hough space. In turn, this approach also does not require pre-generated hybrid masks but conducts the voting process locally by evaluating the overlapping hybrid mask columns in parallel for all t-points in a reasonable radius. Thus, instead of inserting all the t-points to one global accumulator, the t-points were inserted to multiple smaller accumulators (clusters) based on the possible overlaps in the Hough space (i.e., based on the mutual distances of all stored t-points). However, due to lack of time during development of this project, this approach was not tested in VHDL simulation, but the significant reduction of memory requirements makes this approach a promising idea to be tested properly on the FPGA in future.

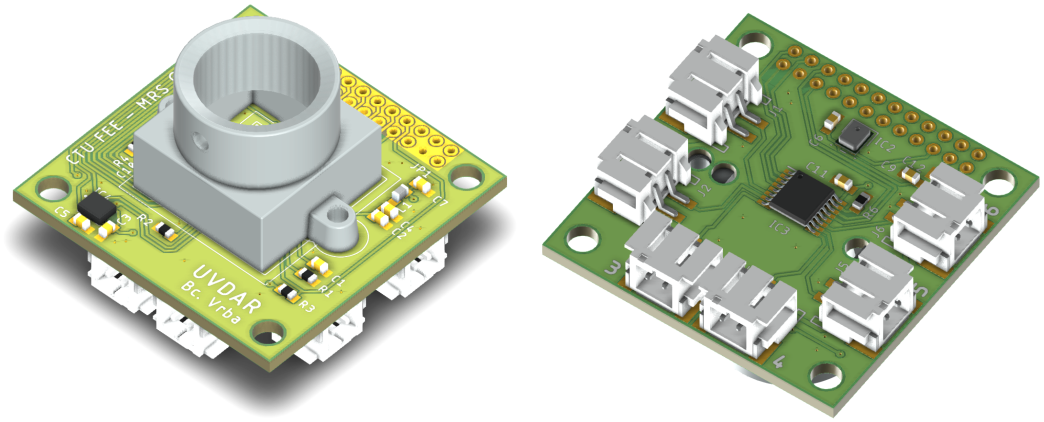
5.3 Running ROS Distribution for Comparison of Results

Another feature which is planned to be implemented in future is running a ROS distribution directly on the HPS in the Linux user space. This would allow for a quick and simple comparison of the results obtained from the FPGA architecture with the current ROS implementation of the UVDAR system. However, I was unable to compile the ROS distribution under the Debian distribution due to a large number of unresolved dependencies on the ARM architecture.

5.4 Embedded Application of the UVDAR System

The ultimate goal of a follow-up work on this project is embedding the UVDAR system into an independent device. Such device should contain a CMOS imaging chip and computational resources. Due to the current low availability of discrete FPGA chips on the market, an embedded MCU can be considered for this task. The choice of the MCU also simplifies the final board design of the device as MCUs are usually available in a quad flat package (QFP), unlike most of the FPGA chips available only in a ball grid array (BGA) packages.

Proposed board designs for a two-part embedded device are visualised in figures 5.1 (a camera board) and 5.2 (a MCU board). The boards are designed with same dimensions and locations of mounting holes, a standard 2mm 24-pin header is used for connection of the two parts.



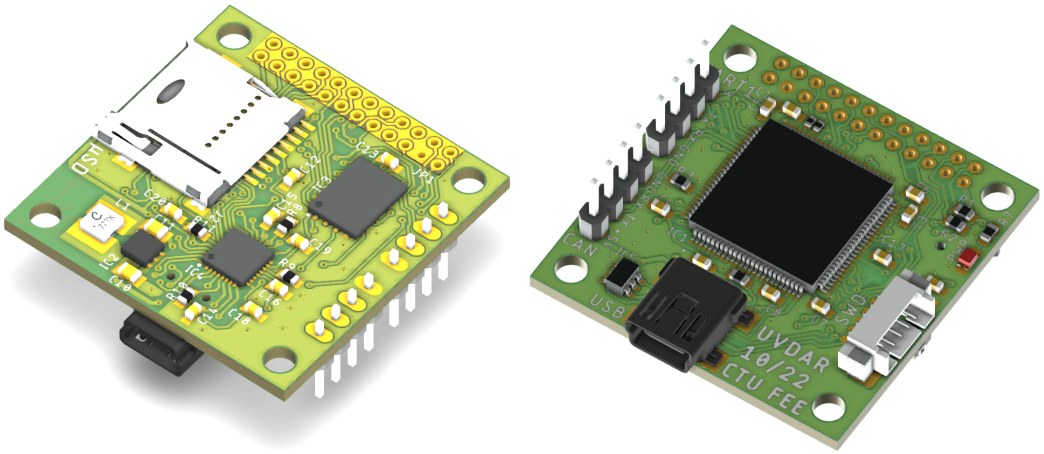
(a) : 3D top view. **(b) :** 3D bottom view.

Figure 5.1: Proposed design for a camera board of the embedded UVDAR system.

The key components incorporated in the camera board design are:

- ON Semiconductor MT9V034 - CMOS imaging sensor.
- ST ALED8102S - Constant current LED driver with up to 6 LEDs to be connected to the board.

- (Optionally) ST IMP23ABSU - Analog bottom port microphone with frequency response up to 80kHz.
- (Optionally) Bosch Sensortec BMX160 - Absolute orientation MEMS sensor with 9-DOF (3-axis accelerometer, 3-axis gyroscope, 3-axis magnetometer).



(a) : 3D top view. **(b) :** 3D bottom view.

Figure 5.2: Proposed design for a MCU board of the embedded UVDAR system.

The key components of the MCU board are:

- ST STM32H750VB - Arm Cortex-M7 MCU with 128 Kbytes of Flash memory, 1MB RAM and 480 MHz (1027 DMIPS) CPU.
- ST ST1S06 - Step-down DC-DC converter to provide 3.3V from USB 5V input.
- ON Semiconductor FUSB2805 - USB 2.0 High-Speed OTG Transceiver with ULPI Interface.
- MicroSD card connector, UART and CAN interfaces exposed to pin headers.
- (Optionally) AP Memory APS6404L - 64Mbit (8MB) PSRAM memory.

These designs, however, are not being elaborated on in this text any further, the boards are presented as an alternative approach to a FPGA-based design. Moreover, the image processing algorithms (especially the 4D Hough Transform) require several modifications to be able to process real time data at 60 FPS on the selected MCU, which is assumed to be a next step in a future evaluation of the proposed MCU approach.

Chapter 6

Conclusion

At first, the pipeline of the UVDAR system was presented with a focus on its two main image processing algorithms, the FAST-like feature detection algorithm for extraction of bright markers from camera images and the 4D Hough Transform for extraction of lines approximating trajectories of the detected markers in an image-time 3D space.

A custom camera board was designed in a way to be compatible with multiple development boards, primarily with the selected DE10-Nano FPGA development board. Its functionality was validated using a Nucleo MCU board at first, a VHDL implementation of the DCMI camera interface was then tested with the camera board using the FPGA board.

The FPGA architecture of the FAST-like algorithm was derived from the pseudocode of the algorithm to ensure consistency in the functionality. The VHDL implementation was validated against the current C++ software implementation, first in a simulation and then on the real FPGA hardware. The parity of the results from both implementations was verified.

The FPGA project also contained a VGA process written in VHDL which was used for a real time visualisation of the camera images with an overlay made of detection results from the FAST-like algorithm implementation via the HDMI interface on the board.

The processor part of the FPGA chip was utilised for storing of camera images and detection results, also as an alternative source of camera images loaded from recordings stored on a microSD card. The main software was written as a Linux user space terminal application in C, running in a fully custom Linux OS build. Also, a custom Linux kernel driver was written to add support for external interrupts driven by the FPGA fabric.

Unfortunately, due to insufficient time resources, FPGA architecture of the 4D Hough Transform was discussed only in theory and no FPGA implementation was tested. The total utilisation of the FPGA resources by the proposed VHDL implementations was discussed, together with ideas for future development of the FPGA project and for an alternative approach with an embedded microcontroller.

References

1. WALTER, Viktor; SASKA, Martin; FRANCHI, Antonio. Fast Mutual Relative Localization of UAVs using Ultraviolet LED Markers. In: *2018 International Conference on Unmanned Aircraft Systems (ICUAS)*. IEEE, 2018. Available from: doi:10.1109/icuas.2018.8453331.
2. LOPEZ, Martin Andreoni; BADDELEY, Michael; LUNARDI, Willian T.; PANDEY, Anshul; GIACALONE, Jean-Pierre. *Towards Secure Wireless Mesh Networks for UAV Swarm Connectivity: Current Threats, Research, and Opportunities*. arXiv, 2021. Available from: doi:10.48550/ARXIV.2108.13154.
3. ABDELKADER, Mohamed; GULER, Samet; JALEEL, Hassan; SHAMMA, Jeff S. Aerial Swarms: Recent Applications and Challenges. *Current Robotics Reports*. 2021, vol. 2, no. 3, pp. 309–320. Available from: doi:10.1007/s43154-021-00063-4.
4. GOEL, Salil. A Distributed Cooperative UAV Swarm Localization System: Development and Analysis. In: *Proceedings of the 30th International Technical Meeting of The Satellite Division of the Institute of Navigation (ION GNSS+ 2017)*. Institute of Navigation, 2017. Available from: doi:10.33012/2017.15217.
5. GOEL, Salil; KEALY, Allison; RETSCHER, Guenther; LOHANI, Bharat. Cooperative P2I localization using UWB and Wi-Fi. In: 2016.
6. COPPOLA, Mario; MCGUIRE, Kimberly; SCHEPER, Kirk Y. W.; CROON, Guido C. H. E. de. *On-board Communication-based Relative Localization for Collision Avoidance in Micro Air Vehicle teams*. arXiv, 2016. Available from: doi:10.48550/ARXIV.1609.08811.
7. WALTER, Viktor; STAUB, Nicolas; SASKA, Martin; FRANCHI, Antonio. Mutual Localization of UAVs based on Blinking Ultraviolet Markers and 3D Time-Position Hough Transform. In: *2018 IEEE 14th International Conference on Automation Science and Engineering (CASE)*. IEEE, 2018. Available from: doi:10.1109/coase.2018.8560384.
8. WALTER, Viktor; STAUB, Nicolas; FRANCHI, Antonio; SASKA, Martin. UVDAR System for Visual Relative Localization With Application to Leader–Follower Formations of Multirotor UAVs. *IEEE Robotics and Automation Letters*. 2019, vol. 4, no. 3, pp. 2637–2644. Available from: doi:10.1109/lra.2019.2901683.

9. WALTER, Viktor; VRBA, Matouš; SASKA, Martin. On training datasets for machine learning-based visual relative localization of micro-scale UAVs. In: *2020 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2020. Available from: doi:10.1109/icra40945.2020.9196947.
10. FULARZ, Michal; KRAFT, Marek; SCHMIDT, Adam; KASIŃSKI, Andrzej. A High-Performance FPGA-Based Image Feature Detector and Matcher Based on the FAST and BRIEF Algorithms. *International Journal of Advanced Robotic Systems*. 2015, vol. 12, no. 10, p. 141. Available from: doi:10.5772/61434.
11. ROSTEN, E.; DRUMMOND, T. Fusing points and lines for high performance tracking. In: *Tenth IEEE International Conference on Computer Vision (ICCV'05) Volume 1*. IEEE, 2005. Available from: doi:10.1109/iccv.2005.104.
12. BRESENHAM, Jack. A linear algorithm for incremental digital display of circular arcs. *Communications of the ACM*. 1977, vol. 20, no. 2, pp. 100–106. Available from: doi:10.1145/359423.359432.
13. ILLINGWORTH, J.; KITTLER, J. A survey of the hough transform. *Computer Vision, Graphics, and Image Processing*. 1988, vol. 44, no. 1, pp. 87–116. Available from: doi:10.1016/s0734-189x(88)80033-1.
14. BARBOSA, W.O.; VIEIRA, Antonio W. On the Improvement of Multiple Circles Detection from Images using Hough Transform. *TEMA (São Carlos)*. 2019, vol. 20, no. 2, pp. 331–342. Available from: doi:10.5540/tema.2019.020.02.0331.
15. YUEN, H. K.; ILLINGWORTH, J.; KITTLER, J. Ellipse Detection using the Hough Transform. In: *Procedings of the Alvey Vision Conference 1988*. Alvey Vision Club, 1988. Available from: doi:10.5244/c.2.41.
16. BALLARD, D.H. Generalizing the Hough transform to detect arbitrary shapes. *Pattern Recognition*. 1981, vol. 13, no. 2, pp. 111–122. Available from: doi:10.1016/0031-3203(81)90009-1.
17. KIEFER, Gundolf; VAHL, Matthias; SARCHER, Julian; SCHAEFERLING, Michael. A configurable architecture for the generalized hough transform applied to the analysis of huge aerial images and to traffic sign detection. In: *2016 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE, 2016. Available from: doi:10.1109/reconfig.2016.7857143.
18. DALITZ, Christoph; SCHRAMKE, Tilman; JELTSCH, Manuel. Iterative Hough Transform for Line Detection in 3D Point Clouds. *Image Processing On Line*. 2017, vol. 7, pp. 184–196. Available from: doi:10.5201/ipol.2017.208.
19. WALTER, Viktor. *UVDAR Drivers and Processing - GitHub Repository*. 2019. Also available from: https://github.com/ctu-mrs/uvdar_core. (accessed in Dec 2022).
20. BÁČA, Tomáš. *MRS Serial Protocol (Baca Protocol) - GitHub Repository*. 2017. Also available from: https://github.com/ctu-mrs/mrs_serial. (accessed in Dec 2022).

21. MATRIX VISION. *mvBlueFOX-MLC Series Datasheet*. USB 2.0 Board-level Industrial Cameras. Also available from: https://publications.balluff.com/pdfengine/pdf?id=MP11708010&language=en&type=kmat. (accessed in Dec 2022).
22. ON SEMICONDUCTOR. *MT9V034 Datasheet*. 1/3-Inch Wide-VGA CMOS Digital Image Sensor. Also available from: https://www.onsemi.com/pdf/datasheet/mt9v034-d.pdf. (accessed in Dec 2022).
23. ŽAITLÍK, David. *UVDAR Hardware - GitHub Repository*. 2021. Also available from: https://github.com/ctu-mrs/mrs_hw_uvdar. (accessed in Dec 2022).
24. HUANG, Jingjin; ZHOU, Guoqing; ZHOU, Xiang; ZHANG, Rongting. A New FPGA Architecture of FAST and BRIEF Algorithm for On-Board Corner Detection and Matching. *Sensors*. 2018, vol. 18, no. 4, p. 1014. Available from: doi:10.3390/s18041014.
25. HAIJUN, Lei; ZHANFU, Chen; ZHANG, Yang. Design and FPGA Implementation of the Fast Algorithm for Extracting Laser Line. In: *2010 International Conference on Communications and Mobile Computing*. IEEE, 2010. Available from: doi:10.1109/cmc.2010.315.
26. HEO, Hoon; LEE, Jung-yong; LEE, Kwang-yeob; LEE, Chan-ho. FPGA based implementation of FAST and BRIEF algorithm for object recognition. In: *2013 IEEE International Conference of IEEE Region 10 (TENCON 2013)*. IEEE, 2013. Available from: doi:10.1109/tencon.2013.6718842.
27. ČÍŽEK, Petr; FAIGL, Jan. Real-Time FPGA-Based Detection of Speeded-Up Robust Features Using Separable Convolution. *IEEE Transactions on Industrial Informatics*. 2018, vol. 14, no. 3, pp. 1155–1163. Available from: doi:10.1109/tii.2017.2764485.
28. ČÍŽEK, Petr; FAIGL, Jan. On FPGA Based Acceleration Of Image Processing In Mobile Robotics. *Acta Polytechnica CTU Proceedings*. 2015, vol. 2, no. 2, pp. 8–14. Available from: doi:10.14311/app.2015.1.0008.
29. DOHI, Keisuke; YORITA, Yuji; SHIBATA, Yuichiro; OGURI, Kiyoshi. Pattern Compression of FAST Corner Detection for Efficient Hardware Implementation. In: *2011 21st International Conference on Field Programmable Logic and Applications*. IEEE, 2011. Available from: doi:10.1109/fpl.2011.94.
30. KRAFT, Marek; SCHMIDT, Adam; KASIŃSKI, Andrzej. High-Speed Image Feature Detection Using FPGA Implementation of Fast Algorithm. In: 2008, vol. 1, pp. 174–179.
31. TAGZOUT, Samir; ACHOUR, Karim; DJEKOUNE, Oualid. Hough transform algorithm for FPGA implementation. *Signal Processing*. 2001, vol. 81, no. 6, pp. 1295–1301. Available from: doi:10.1016/s0165-1684(00)00248-6.
32. DJEKOUNE, O.; ACHOUR, K. Incremental Hough transform: an improved algorithm for digital device implementation. *Real-Time Imaging*. 2004, vol. 10, no. 6, pp. 351–363. Available from: doi:10.1016/j.rti.2004.07.001.

33. JEONG, Hyo-Kyun; JEONG, Yong-Jin. Design of Hough transform hardware accelerator for lane detection. In: *2013 IEEE International Conference of IEEE Region 10 (TENCON 2013)*. IEEE, 2013. Available from: doi:10.1109/tencon.2013.6719020.

34. HAJJOUJI, Ismaïl El; MARS, Salah; ASRIH, Zakariae; MOURABIT, Aimad El. A novel FPGA implementation of Hough Transform for straight lane detection. *Engineering Science and Technology, an International Journal*. 2020, vol. 23, no. 2, pp. 274–280. Available from: doi:10.1016/j.jestch.2019.05.008.

35. ELHOSSINI, A.; MOUSSA, M. Memory efficient FPGA implementation of hough transform for line and circle detection. In: *2012 25th IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*. IEEE, 2012. Available from: doi:10.1109/ccece.2012.6335003.

36. SEO, Sang-Woo; KIM, Myunggyu. Efficient architecture for circle detection using Hough transform. In: *2015 International Conference on Information and Communication Technology Convergence (ICTC)*. IEEE, 2015. Available from: doi:10.1109/ictc.2015.7354612.

37. ORLANDO, Chuquimia; ANDREA, Pinna; CHRISTOPHEL, Marsala; XAVIER, Dray; GRANADO, Bertrand. FPGA-Based Real Time Embedded Hough Transform Architecture for Circles Detection. In: *2018 Conference on Design and Architectures for Signal and Image Processing (DASIP)*. IEEE, 2018. Available from: doi:10.1109/dasip.2018.8597174.

38. ZHOU, Xin; ITO, Yasuaki; NAKANO, Koji. An Efficient Implementation of the One-Dimensional Hough Transform Algorithm for Circle Detection on the FPGA. In: *2014 Second International Symposium on Computing and Networking*. IEEE, 2014. Available from: doi:10.1109/candar.2014.32.

39. ALIM, F. Ferhat-taleb; MESSAOUDI, K.; SEDDIKI, S.; KERDJIDJ, O. Modified circular Hough transform using FPGA. In: *2012 24th International Conference on Microelectronics (ICM)*. IEEE, 2012. Available from: doi:10.1109/icm.2012.6471412.

40. SOLOD, Panadda; JINDAPETCH, Nattha; SENGCHUAI, Kiattisak; BOORANAWONG, Apidet; HOYINGCHAROEN, Pakpoom; CHUMPOL, Surachate; IKURA, Masami. Memory Optimization for Accelerating Hough Transform on FPGA using High Level Synthesis. In: *2019 IEEE International Circuits and Systems Symposium (ICSyS)*. IEEE, 2019. Available from: doi:10.1109/icsys47076.2019.8982398.

41. MICROCHIP. *M2S-HELLO-FPGA-KIT Overview*. Microsemi Product Portfolio. Also available from: https://www.microsemi.com/existing-parts/parts/150925. (accessed in Dec 2022).

42. TERASIC. *DE10-Nano Kit Overview*. Terasic SoC Platform Products. Also available from: https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=&No=1046. (accessed in Dec 2022).

43. INTEL. *Cyclone V Device Overview*. An overview of the Cyclone V devices family. Also available from: https://www.intel.com/content/www/us/en/docs/programmable/683694/current/cyclone-v-device-overview.html. (accessed in Dec 2022).

44. MICROSEMI. *PB0115 - SmartFusion2 SoC FPGA Overview*. An overview of the SmartFusion2 devices family. Also available from: https://www.microsemi.com/document-portal/doc_view/132721-pb0115-smartfusion2-soc-fpga-product-brief. (accessed in Dec 2022).

45. INTEL. *Intel Quartus Prime Software*. FPGA Design Software. Also available from: https://www.intel.com/content/www/us/en/products/details/fpga/development-tools/quartus-prime.html. (accessed in Dec 2022).

46. MICROSEMI. *Libero SoC Design Suite*. FPGA Design Software. Also available from: https://www.microchip.com/en-us/products/fpgas-and-plds/fpga-and-soc-design-tools/fpga/libero-software-later-versions. (accessed in Dec 2022).

47. STMICROELECTRONICS. *STM32H745ZI Datasheet*. Dual 32-bit Arm® Cortex®-M7 up to 480MHz and -M4 MCUs, up to 2MB Flash, 1MB RAM, 46 com. and analog interfaces, SMPS. Also available from: https://www.st.com/resource/en/datasheet/stm32h745zi.pdf. (accessed in Dec 2022).

48. STMICROELECTRONICS. *AN5020 - Application Note*. Digital camera interface (DCMI) for STM32 MCUs. Also available from: https://www.st.com/resource/en/application_note/an5020-digital-camera-interface-dcmi-on-stm32-mcus-stmicroelectronics.pdf. (accessed in Dec 2022).

49. INTEL. *AN796 - Application Note*. Cyclone V and Arria V SoC Device Design Guidelines. Also available from: https://cdrdv2-public.intel.com/666598/an-cv-av-soc-ddg-683360-666598.pdf. (accessed in Dec 2022).

50. LARSON, Scott. *I2C Master VHDL Implementation*. 2015. Also available from: https://forum.digikey.com/t/i2c-master-vhdl/12797. (accessed in Dec 2022).

51. NXP. *UM10204 - User Manual*. UM10204 - I2C-bus specification and user manual. Also available from: https://cache.nxp.com/secured/assets/documents/en/user-guide/UM10204.pdf. (accessed in Dec 2022).

52. ANALOG DEVICES. *ADV7513 Datasheet*. 165MHz High Performance HDMI Transmitter. Also available from: https://www.analog.com/media/en/technical-documentation/data-sheets/adv7513.pdf. (accessed in Dec 2022).

53. HO, Thanh Phong. *Fonts generated by GLCD Font Creator - GitHub Repository*. 2021. Also available from: https://github.com/phonght32/fonts. (accessed in Dec 2022).

54. INTEL. *Cyclone V HPS TRM*. Cyclone V Hard Processor System Technical Reference Manual. Also available from: https://www.intel.com/content/www/us/en/docs/programmable/683126/21-2/introduction-to-the-hard-processor-system-98309.html. (accessed in Dec 2022).

55. DENK, Wolfgang. *Boot loader for Embedded boards based on PowerPC, ARM, MIPS etc. - GitHub Repository*. 2000. Also available from: https://github.com/u-boot/u-boot. (accessed in Dec 2022).

56. TORVALDS, Linus. *Altera Open Source Linux kernel repository for SoC - GitHub Repository*. 2005. Also available from: https://github.com/altera-opensource/linux-socfpga. (accessed in Dec 2022).

57. LELLI, Juri; SCORDINO, Claudio; ABENI, Luca; FAGGIOLI, Dario. Deadline scheduling in the Linux kernel. *Software: Practice and Experience*. 2015, vol. 46, no. 6, pp. 821–839. Available from: doi:10.1002/spe.2335.

58. KAWAZOME, Ichiro. *User Space Mappable DMA Buffer - GitHub Repository*. 2019. Also available from: https://github.com/ikwzm/udmabuf. (accessed in Dec 2022).

Appendix A

Complete Camera Board Design

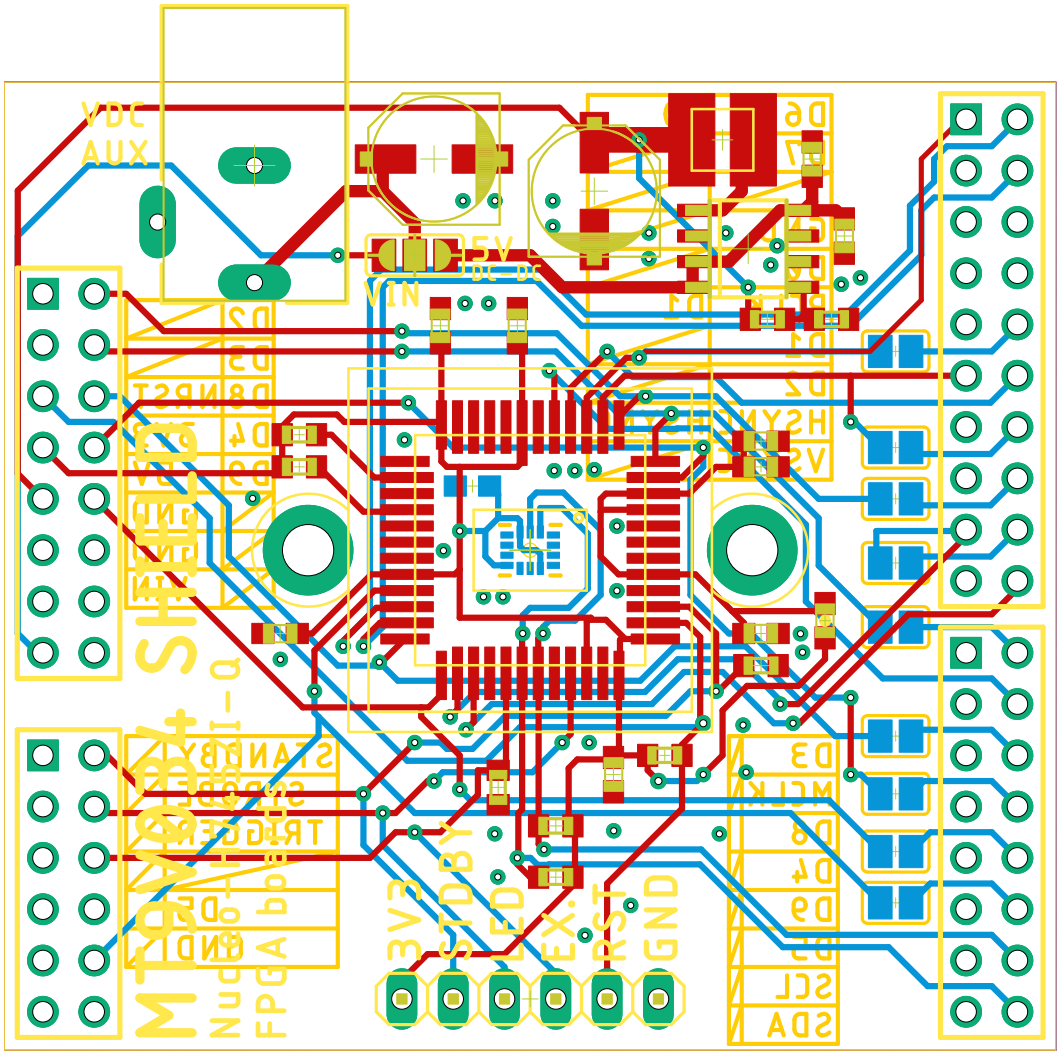


Figure A.1: Complete camera board layout.

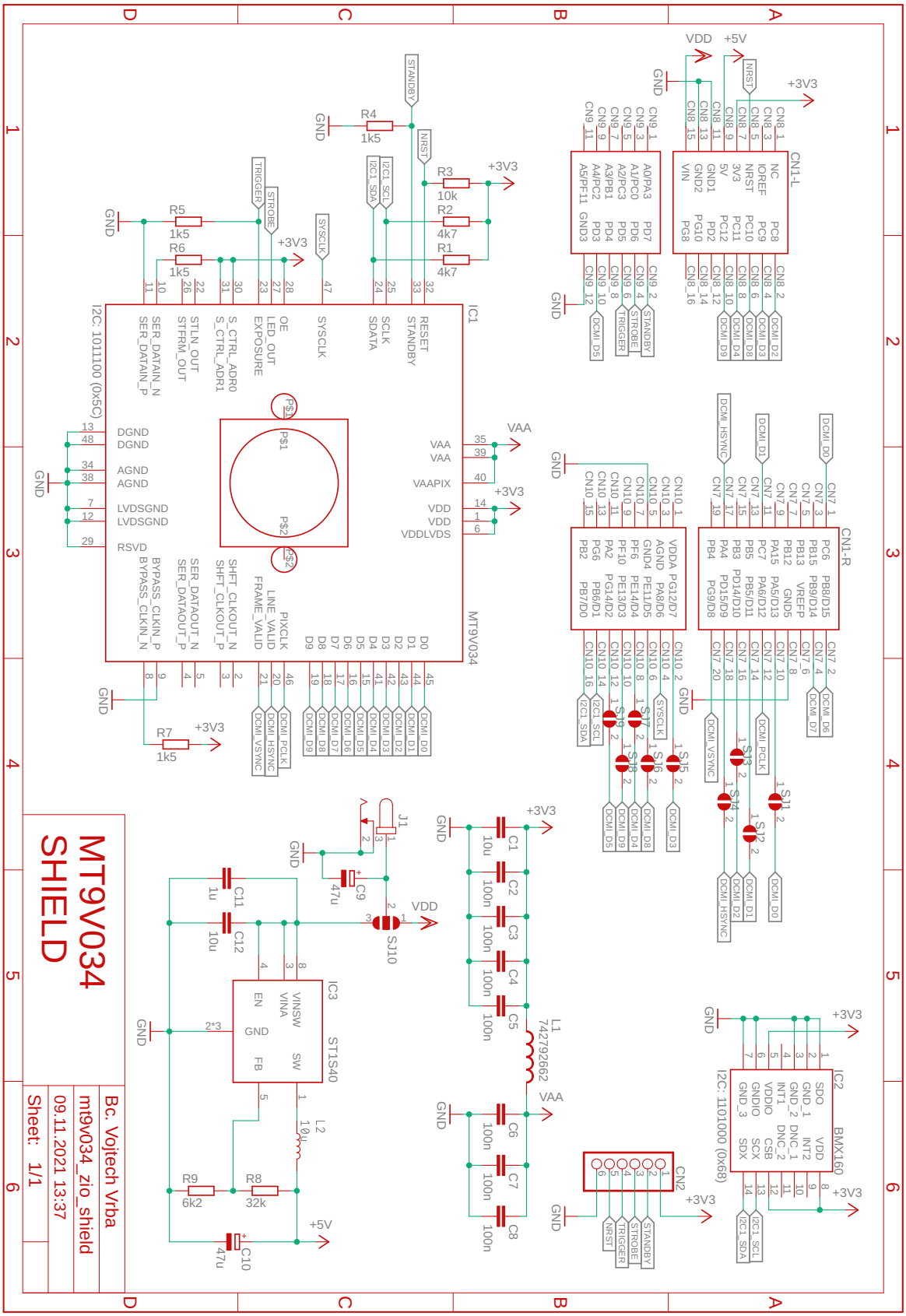


Figure A.2: Complete camera board schematic.