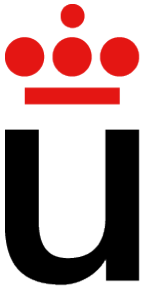

Universidad
Rey Juan Carlos

GRADO EN INGENIERÍA TELEMÁTICA

Curso Académico 2021/2022

Trabajo Fin de Grado

Real and Simulated FPGA Based Computer Vision for Robots

Autor :

Richard Alexander Nicklas Rzepka

Tutores :

José María Cañas Plaza y Felipe Machado Sánchez

Trabajo Fin de Grado

Real and Simulated FPGA Based Computer Vision for Robots

Autor : Richard Alexander Nicklas Rzepka

Tutor : Jose María Cañas Plaza y Felipe Machado Sánchez

La defensa del presente Proyecto Fin de Carrera se realizó el día de de 2022, siendo calificada por el siguiente tribunal:

Presidente:

Secretario:

Vocal:

y habiendo obtenido la siguiente calificación:

Calificación:

Fuenlabrada, a de de 2022

Even the mountains
erode in the rain.

II

Acknowledgements

I would like to thank my tutors Jose María Cañas Plaza and Felipe Machado Sánchez for their support and patience.

I would also like to thank my family for their support.

....

Abstract

This Final Degree Project contributes with the JdeRobot organization in the creation of an open source toolchain for the development of FPGA applications for robots. With this objective, several pieces of software have been developed and tested to be included in its public repository.

The work undertaken focused on two different scenarios, one physical and hardware-oriented, and the other, simulated and software-oriented. A real GoPiGo robot, physical cameras, the Gazebo robotics simulator and FPGA technologies like the Verilog hardware description language and the Verilator simulator have been extensively used.

The research and developments presented in this paper take place inside the JdeRobot's FPGA-Robotics project with the aim of expanding its library of reusable nodes or "blocks". As experimental validation, several POCs (Proof of Concept) and robotic applications for FPGAs, both physical and simulated, have been designed, developed and tested. For instance a robot following a colored object using its onboard camera.

Resumen

Este Trabajo de Fin de Grado contribuye con la organización JdeRobot en la creación de una cadena de herramientas de código abierto para el desarrollo de aplicaciones FPGA para robots. Con este objetivo, se han desarrollado y probado varias piezas de software para ser incluidas en su repositorio público.

El trabajo realizado se ha centrado en dos escenarios diferentes, uno físico y orientado al hardware, y otro simulado y orientado al software. Un robot GoPiGo real, cámaras físicas, el simulador de robótica Gazebo y tecnologías FPGA como el lenguaje de descripción de hardware Verilog y el simulador Verilator se han utilizado de manera extensiva.

La investigación y los desarrollos presentados en este trabajo tienen lugar dentro del proyecto FPGA-Robotics de JdeRobot con el objetivo de ampliar su biblioteca de nodos o ”bloques” re-utilizables. Como validación experimental, se han diseñado, desarrollado y probado varias POCs (Proof of Concept) y aplicaciones robóticas para FPGAs, tanto físicas como simuladas. Por ejemplo, un robot que sigue un objeto de color utilizando su cámara de a bordo.

Contents

List of Figures

Chapter 1

Introduction

This project is framed in a very rich environment dedicated to developing open source solutions in the fields of Robotics and Computer Vision. Specifically, it is developed in a branch that aims to use Field-Programmable Gate Arrays (FPGAs) as the main processing unit of robots instead of the more common general-purpose Central Processing Units (CPUs) or application-specific integrated circuits (ASICs).

Robotics is one of the fastest growing fields in science and technology so innovation is highly sought after, the use of FPGAs opens a wide range of possibilities for increasing performance and diminishing costs. Even though it is a growing sector, FPGAs are already being used in many high-end robotics applications, usually for computer vision and motor control where sub-microsecond precision programming can be a sizeable advantage and also because their re-programmable nature and flexibility eases the prototyping stage of development.

Still, FPGAs only constitute a small portion of the controllers present in robots today. Users of robotic products usually demand either highly optimized or highly flexible products, while FPGAs provide functionality somewhere in between. A strong community of developers is the backbone of every technology but the free and open source environment surrounding FPGAs is still, although growing, rather small, therefore students and enthusiasts will struggle finding collaborative settings to learn, try and contribute.

This project represents further steps in the aim of the JdeRobot[1] open source organization and its FPGA-robotics[2] project, and myself to work towards an ever growing free and open

[1] `https://github.com/JdeRobot`

[2] `https://github.com/JdeRobot/FPGA-robotics`

Figure 1.1: KUKA KR 210 R2700 Robot

source development environment for Robotics, Computer Vision and FPGAs making these technologies more accessible for everyone.

1.1 Robotics

Robotics is an interdisciplinary sector of science and engineering dedicated to the design, construction and use of mechanical robots. These machines are intended to reduce or eliminate the workload performed by humans in production, execution or service processes. Actuators, end effectors, robotic manipulators, controllers and sensors like inertial measurement units, microphones and cameras make up the majority of robots.

The robotics sector is hugely diverse and continues to expand, feeding into several key areas that include healthcare, machine vision, warehouse and production line automation, autonomous vehicles, the military complex and general industrial operations.

Production processes employ the vast majority of robots nowadays. The global industrial robots market size was valued at USD 15.60 billion in 2021 and is expected to triple by the

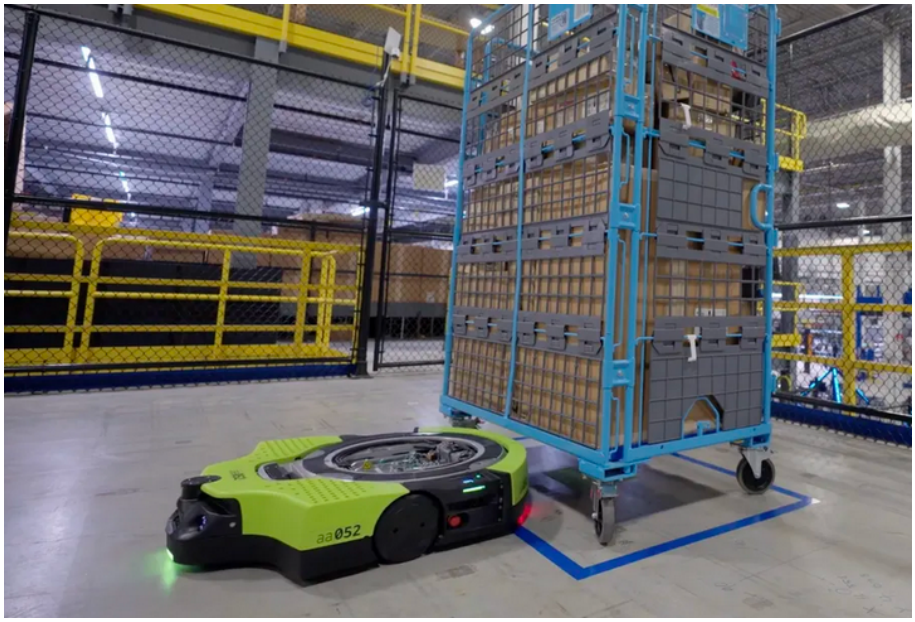

Figure 1.2: Amazon ”Proteus” Warehouse Robot

end of the decade[3]. Industrial robots, like the Kuka KR 210 R2700 production line automation robot pictured in Figure 1.1, are a kind of mechanical equipment that is fed with data and programmed to execute activities linked to industry production automatically. This type of robot helps to improve productivity, reducing costs and generating high quality goods in automation applications.

Among the robots dedicated to industrial processes, one of the fastest growing sectors is logistics. Companies like Amazon, Alibaba or JD.com, which are among the biggest e-commerce corporations in the world, are spending billions of USD in the research and development of automation and robotics applications to optimize their production processes.

Examples of this technological revolution are the latest version of Amazon's warehouse robot ”Proteus” pictured in Figure 1.2. Hailed as ”fully-autonomous” it can safely navigate around human employees, unlike some previous versions that had to be kept separated in a caged area. Another example is JD's fully automated $40.000\,\mathrm{m}^2$ Shanghai facility[4], this warehouse used to be operated by around 500 human employees but now only relies on a few to perform robot maintenance.

[3] https://www.fortunebusinessinsights.com/industry-reports/industrial-robots-market-100360

[4] https://www.youtube.com/watch?v=RFV8IkY52iY&t

Figure 1.3: iRobot's Roomba J7 Autonomous Vacuum Cleaner

Beyond industrial production processes, robotics has a very important future in the direct provision of services to humans. The aging of the population or the emergence of circumstances that isolate people, such as pandemics, highlight the growing need for autonomous solutions for everyday tasks.

Robots rely on their sensors and actuators to perform these functions. Cameras are amongst the most popular sensors found in most of advanced robots today as they provide extensive information about the robot's surroundings, but computer vision is not an easy task.

Computer Vision and Robotics

Computer or machine vision is a field of research that has a strong presence in robotics. It attempts to simulate the capabilities of the human eye and brain. this is typically done with the help of sensors that provide the robot with the necessary data to help with the perception of the environment and its surroundings. Cameras are commonly found in most robots as they may potentially provide much information. Nevertheless, extraction of this information from camera pixels is not simple. Computer vision involves image processing, a very resource-intensive and complex task. Usually, the costs of image recognition is driven by the powerful processors and complex software needed to handle these processes. The most advanced prototypes are already taking advantage of the hardware optimization that devices like ASICs and FPGAs provide to increase performance and reduce costs.

Figure 1.4: Self-driving Waymo Jaguar I-Pace on the streets of San Francisco.

One of the main problems studied extensively in computer vision during the last decades is autonomous navigation. Currently, there is extensive research on how to use Artificial intelligence and specifically, Artificial Neural Networks (ANN) to realize autonomous behavior in vehicles and robots. Even though full autonomous driving is still to be achieved we can already see examples of very advanced robotic navigation in popular consumer products.

Autonomous robotic vacuum cleaners have been available on the market for years but the latest versions, such as the iRobot Roomba J7 pictured in Figure 1.3, are incorporating novel computer vision and machine learning techniques to improve its capabilities. Such as visually detecting wires or sockets which could block the vacuum or recognizing and avoiding obstacles blocking its path. These robots are no longer just vacuum cleaners but ”smart” devices capable of adaptative behaviour and complex decision making.

Companies like Google’s Waymo, Baidu or Tesla have spearheaded the research and development in the field of autonomous driving in the last decades but thousands of other enterprises have joined the race to develop new technologies that have the potential to disrupt markets worth trillions of USD, like freight shipping or public transportation. Examples of this revolution can already be seen, vehicles carrying Waymo’s latest prototypes, like the self-driving Waymo Jaguar I-Pace pictured in Figure 1.4, have already begun offering driver-less rides in

San Francisco[5] while companies like TuSimple and Tesla are already testing self-driving trucks on public roads in the United States.

Other sectors also employ computer vision extensively, military applications are probably one of the largest areas for computer vision. Automatic terrain recognition can provide a rich set of information about combat scenes which can be used to support strategic decisions. Missile guidance and target selection also rely heavily in computer vision algorithms.

The fields of robotics and computer vision are set to grow dramatically in the next decades expanding their presence in every aspect of life. One of the main research and development branches for these technologies is hardware optimization. This project aims to research how open source FPGAs can be used for this purpose in both real and simulated robots.

1.2 Field-Programmable Gate Arrays

A Field-Programmable Gate Array or ”FPGA” is an integrated circuit whose internal structure is not predefined but designed to be configured after manufacturing by the end user. Robots have traditionally relied on Central Processing Units (CPUs) with generic processors or Application-Specific Integrated Circuits (ASICs) to perform all the resource-intensive processing needed, but the use of FPGAs for these purposes has been increasing in the last decades.

An FPGA is based on a matrix of configurable logic blocks that are connected by programmable interconnects. These integrated circuits are not made to be application-specific as opposed to ASICs, but do benefit from the hardware optimization aspect of their design. Unlike traditional micro-processors that require multiple cores to perform parallel computing, these type of integrated circuits make use of hardware description languages (HDLs) to design multiple concurrent processing structures, allowing for much higher performance. Another core aspect of FPGAs is their re-programmability, their flexible structure allows this piece of hardware to be repurposed when needed. This has become much a more important consideration when taking into account the reduction of semiconductor production the world has seen in recent years. The scarcity of primary resources, geopolitical tensions and the slowdown of global supply lines are seriously straining the capacity of companies to acquire additional hardware

[5] https://www.bloomberg.com/news/articles/2022-03-30/google-s-waymo-to-offer-public-fully-driverless-rides-in-san-francisco

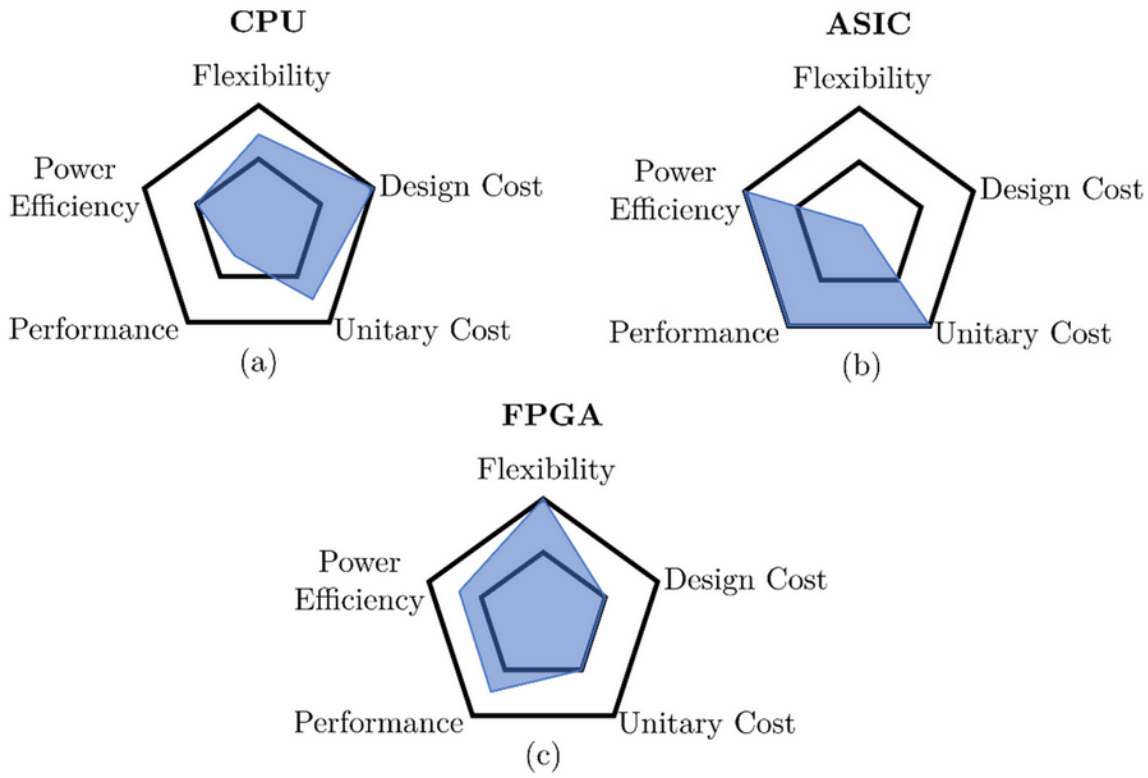

Figure 1.5: Summary Comparison between CPUs, ASICs and FPGAs

and this only highlights the importance of FPGA's re-programmability.

In Figure 1.5 a summary of the comparison between the three hardware architectures mentioned can be seen [1].

(a) CPUs are probably the most widespread technology, as every PC is equipped with one of them. Their availability, flexibility and ease-of-use make them the perfect early development candidates when deploying algorithms or developing code.

(b) ASICs are extremely geared towards high-performance and low unitary cost for high production environments. Products designed for specific tasks where economies of scale allow for a custom solution will use this kind of device expecting to offset the high fixed costs.

(c) Field Programmable Gate Arrays (FPGAs) offer a unique combination of flexibility and high performance, they play a big role in the design and implementation of product prototypes as they can be programmed to perform similarly to highly specialized platforms. FPGAs may also be more cost-effective for smaller designs or lower production volumes.

The field of robotics and computer vision can benefit particularly from these features. General-purpose micro-processors waste resources that may be critical for robots, with the use of FPGAs

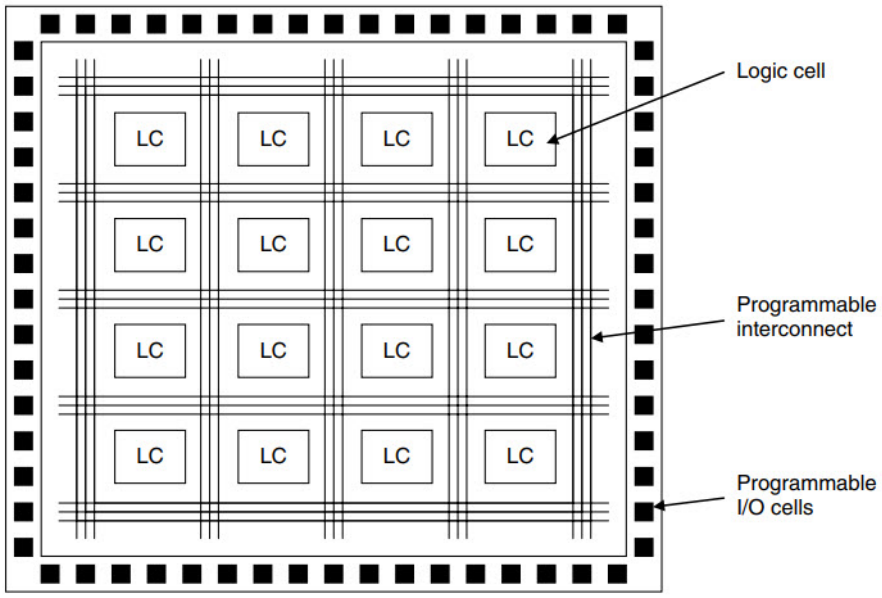

Figure 1.6: FPGA Internal Structure Diagram

autonomy and performance can be increased, and operation costs reduced.

1.2.1 FPGA Structure

The general FPGA architecture consists of three types of modules: Configurable logic blocks (CLB), programmable interconnects and programmable I/O cells. Arrays of configurable logic blocks, also known as Logic Cells (LCs) are interconnected to each other with input/output cells through vertical and horizontal connection channels, as shown in Figure 1.6.

In general, FPGAs can be said to have a fairly regular structure, although the logic blocks and routing architecture varies from one manufacturer to another. Logic Blocks contain the application logic and usually are made up of multiplexed Look-up tables (LUTs) that implement the combinational logic functions. An example of CLB architecture is shown in Figure 1.7, it portrays a simplified Diagram of Xilinx XC4000 CLB.

Not only are the logic blocks configurable, but the routing matrix as well as the input and output cells are programmable, giving the FPGAs great flexibility in adjusting to the specifications of each design.

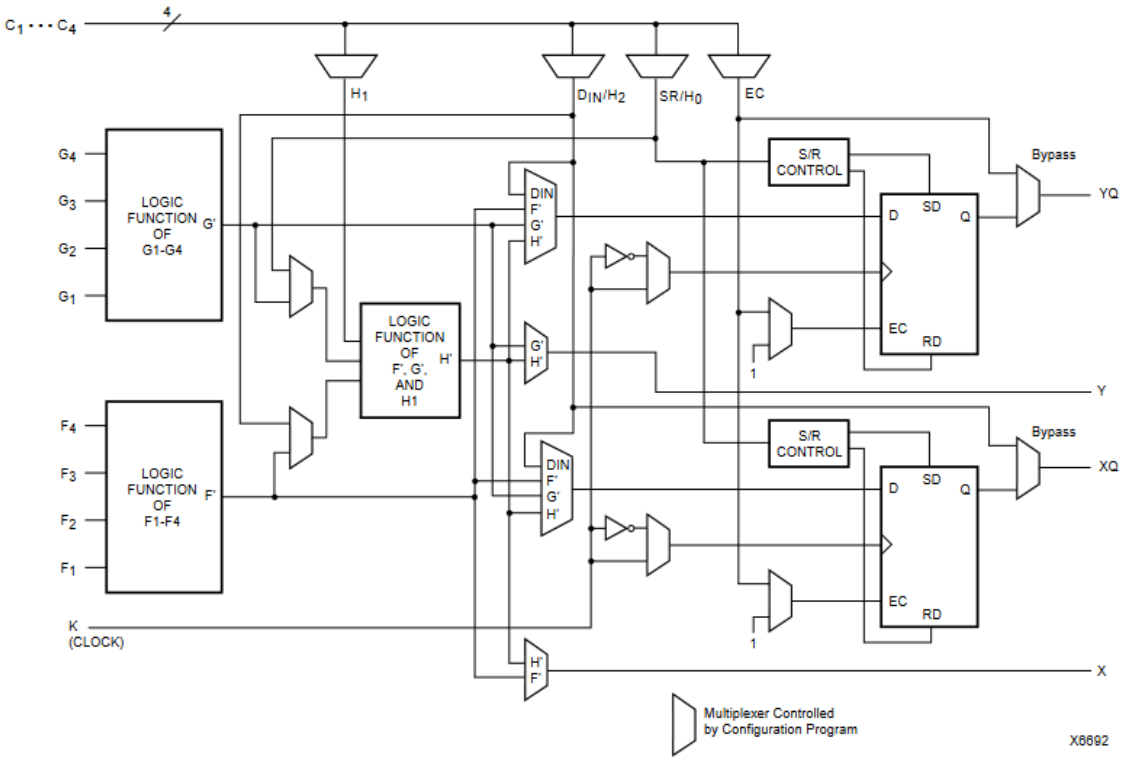

Figure 1.7: Block Diagram of Xilinx XC4000 CLB

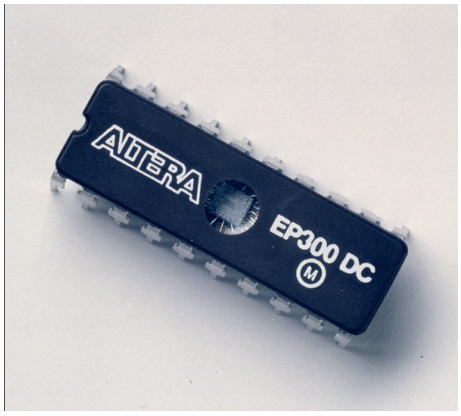

Figure 1.8: Altera EP300

1.2.2 History

Programmable read-only memory (PROM) and programmable logic devices (PLDs) are the precursor technologies that eventually led to Field-Programmable Gate Arrays. These devices could be programmed on the factory or ”field” but were manufactured in hard-wired batches.

The first reprogrammable logic device is the Altera EP300 1.8 manufactured in 1984, its configuration cells could be erased by shining an Ultraviolet light though a quartz present on the device [11].

The first commercially viable field-programmable gate array dates from 1985 and was invented by Xilinx co-founders Ross Freeman and Bernard Vonderschmitt. The XC2064 [6] offered 800 programmable gates, 64 configurable logic blocks and programmable interconnects between them. It meant the beginning of a whole new sector of integrated circuits.

In the following years Xilinx and Altera became the two main manufacturers. By the end of the decade FPGAs were already presenting up to 10000 logic gates like the ones in the Xilinx XC3000 series or Altera’s X.

During the 90s prototypes reached hundred of thousands of gates, like the one funded by the Naval Surface Warfare Center and developed by Steve Casselman 1992 which presented 600000 array gates [4] The whole decade became a period of rapid growth for FPGAs, both in sophistication and volume of production. Multiple companies joined the field eroding a significant portion of the market held by Xilinx and Altera until that point.

By the early 2000s the number of logic gates was reaching millions and FPGAs, which up until that point were being primarily used in telecommunications and networking, found their

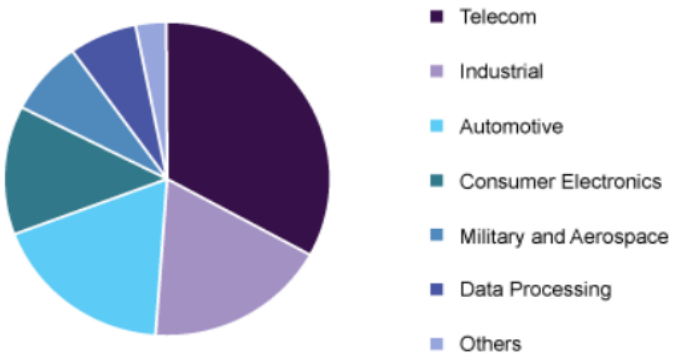

Figure 1.9: Global FPGA market share, by application, 2019 (%)

way into the automotive, consumer electronics and industrial markets. Updated data of FPGA's distribution by application can be seen at Figure 1.9.

Growth continued during the 2010s, hardware optimization has become more relevant as the limits of semiconductor advancement has slowed industry-wide as Moore's law predicted improvements in micro-processors reach its physical limits. The number of gate arrays was reaching dozens of millions as Xilinx and Altera remained the two biggest companies developing FPGAs.

By the end of the decade some of the world's largest semiconductor manufacturing companies were investing heavily in the technology, Altera was acquired by Intel Corporation in 2015 and AMD announced its acquisition of Xilinx in October 2020. Some companies like Google are still focusing on ASICs for AI acceleration and machine learning while others began focusing on innovations in FPGA-based acceleration for their projects and infrastructure. Companies running the world's largest data centers like Amazon, Baidu, and Microsoft are proving that FPGAs can compete in performance while providing the versatility of being able to be repurposed for other uses later on.

1.3 Open source FPGA toolchain

FPGA development usually takes place in integrated development environments provided by the FPGA manufacturers. In most cases these applications are proprietary and almost always not free, although versions with limited functionality are sometimes available free of charge.

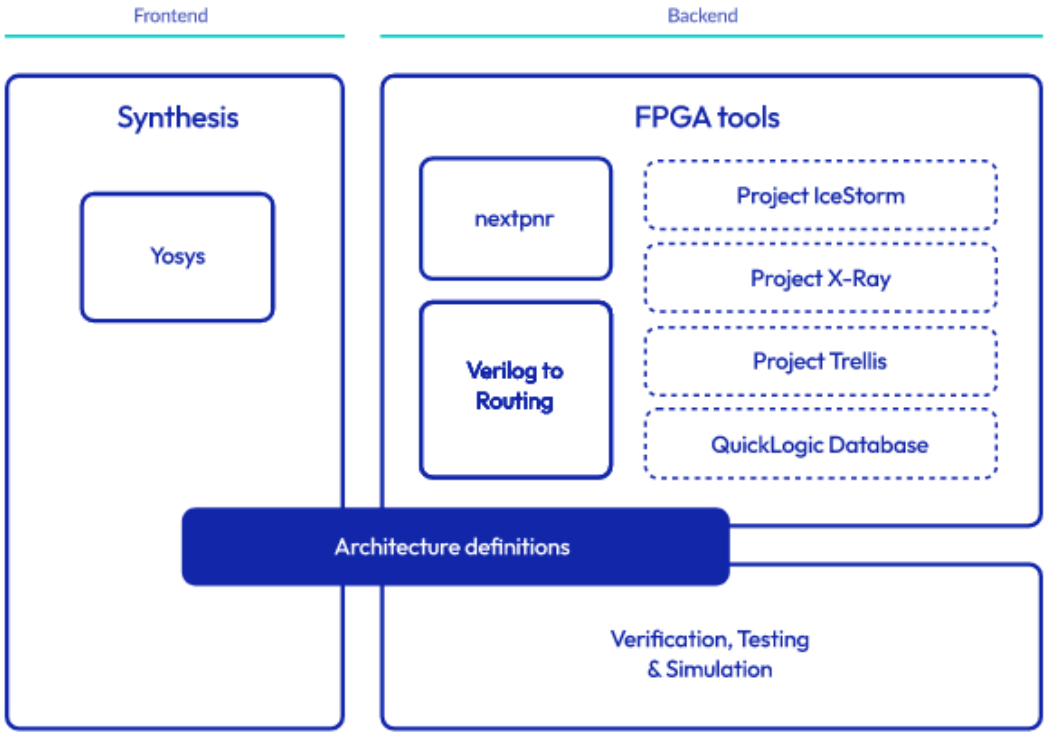

Figure 1.10: F4PGA project structure

Examples of this are Xilinx's "Vitis" and "Vivado ML" software platforms[6] or "Quartus" distributed by Intel (formerly Altera)[7].

Many open source projects have been sprouting around FPGA development due to its increasing popularity. These projects aim to create a complete set of open source development tools to manage and program FPGAs. The Project IceStorm [8], led by Clifford Wolf, is a relevant example, it aims to document the bitstream format of the Lattice iCE40 FPGAs and provide simple tools for analyzing and creating bitstream files. Expanding on the IceStorm project is the Icestudio tool, Figure 1.11, which has been used extensively for this paper. It provides a combination of Verilog and a visual programming environment for the development of FPGA applications.

Another example is the F4PGA[8] (formerly Symbiflow) organization, Figure 1.12, which of-

[6] https://www.xilinx.com/products/design-tools.html

[7] https://www.intel.com/content/www/us/en/products/details/fpga/development-tools.html

[8] https://f4pga.org/

fers a fully open source toolchain for the development of FPGAs of multiple vendors. F4PGA aims to push FPGAs towards more widespread adoption by optimising and automating FPGA development workflows with a set of pluggable open source tools. Its goal is to become a complete FOSS (Free and open source software) toolchain that is Multi-platform, Vendor-neutral and Interchangeable tool suite. Currently, it targets the Xilinx 7-Series, Lattice iCE40, Lattice ECP5 FPGAs, QuickLogic EOS S3 and is gradually being expanded to provide a comprehensive end-to-end FPGA synthesis flow [3]. Lately, projects under the F4PGA organization umbrella have become the main platforms and communities for open source FPGA development. In Figure 1.10 the F4PGA project structure can be seen.

All these projects seek to push FPGAs develpment towards more widespread adoption by providing open source frameworks, but as a relatively recent endeavor, there is a lack of literature and tools showing its potential in the robotics field.

JdeRobot[9], Figure 1.13, is a non-profit organization that aims to provide tools, libraries and reusable nodes for Robotics, Artificial Intelligence and Computer Vision. This organization is investigating via projects like ”Neural FPGA”[10] and ”FPGA-Robotics”[11] how these fields can benefit from incorporating FPGAs in their designs using only open source tools.

The research and developments presented in this paper take place inside the FPGA-Robotics project with the aim of expanding its library of reusable nodes or ”blocks” and, designing and testing both physical and simulated POCs (Proof of Concept) and applications for FPGAs.

[9] https://jderobot.github.io/

[10] https://github.com/JdeRobot/neuralFPGA

[11] https://github.com/JdeRobot/FPGA-robotics

Figure 1.11: Icestudio logo. Open IDE for FPGA development

Figure 1.12: F4PGA logo

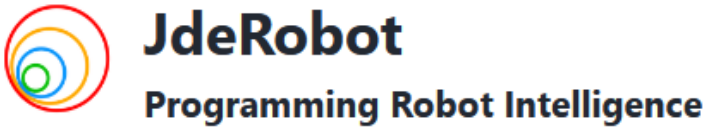

Figure 1.13: JdeRobot organization logo

Chapter 2

Objectives

This chapter outlines the main objectives to be achieved in this project, the motivation behind them and the methodology followed to accomplish them.

2.1 Motivation

Multiple things motivated me towards making this project, first and foremost I am passionate about science and technology as a whole, this project allowed me to use the knowledge acquired during my studies to climb the steps leading to state of the art software development. Not only learning but also contributing to projects and organizations working at the cutting edge of the free and open source movements in the fields of Robotics, FPGAs, and Computer Vision.

I was also motivated by the fact that I had limited knowledge of many of the topics researched during this assignment, during my studies I did not get to dive very deep in some really interesting subjects that proved to be essential in reaching certain milestones. Hardware simulation, electronic circuits, Hardware Description Languages (HDLs), high frequency timed-based programming, Robotics or Computer Vision are just some of the incredibly interesting fields with growing real-life application and environments I got to work with during this process.

2.2 General Goals

The main objective of this project is developing FPGA applications for robots, focusing on the areas of artificial vision, image processing and locomotion. This main objective can be divided

in two major sections.

The first one takes place in the real, physical world. Developing applications to be synthesized and deployed on real FPGAs that can be connected to real robots. The work done in this section can be grouped in a series of ”blocks” created with the intent of providing additional functionality that can be incorporated with other programs already developed by collaborators to the FPGA-Robotics project repository. Two applications are presented in this paper to demonstrate the use of the developed blocks and many others in working examples: 4.3.1 Object detection POC and 4.3.2 GoPiGo 3 robot following a color ball POC. This task does not only imply the development of software but the assembly of the robot with different peripherals like cameras, motors and wheels, monitors and their respective connections.

The second objective is achieving the same results in a completely simulated environment, including both simulated robots and simulated FPGAs. Robotics and Hardware oriented development is somewhat slow, expensive and cumbersome, so being able to simulate the complete chain of technologies involved in these areas is an extremely helpful tool for the development of new applications. Easier debugging, infinitely flexible testing environments, deterministic results and very low cost are some of the advantages that bringing these applications to a simulated world can provide. In a similar fashion to the previous objective this one also focused on developing ”blocks” that help provide functionality to large, complex applications that include the simulation of the processing unit (FPGA), the simulation of the robot and its environment, and the connections between them. A final application is presented to demonstrate the use of the different blocks of technology: 5.3.1 Simulated Tracking Robot POC.

For each of these objectives the following sub goals were undertaken:

1. Researching and understanding the different systems that make an FPGA run a robot

2. Developing code; C++ applications and Verilog modules for data processing in robots and interfacing the real or simulated FPGA with the real or simulated sensors and actuators that provide this information.

3. Assembling both a real and a simulated camera wielding differential drive robot and using it to test the different modules and functionalities.

4. Analysing the results and including the applications into the organization’s repositories.

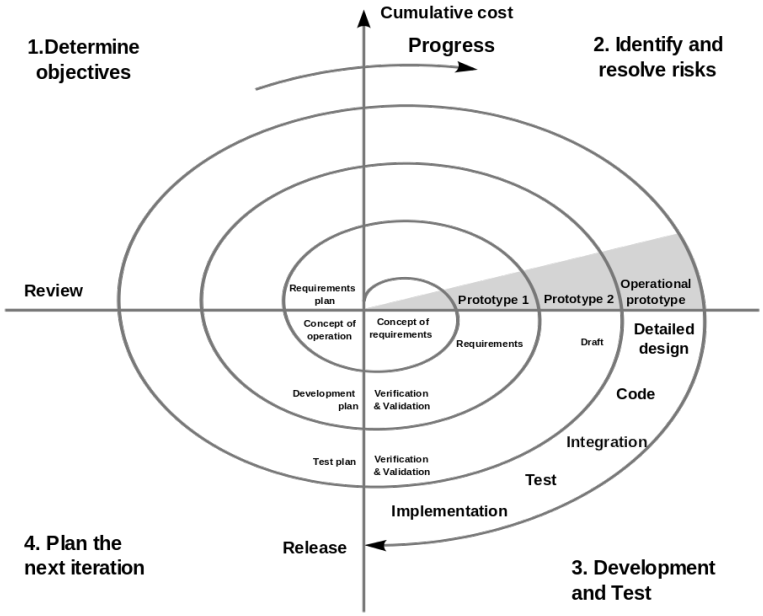

Figure 2.1: Spiral Model (Boehm, 1988).

Each of these steps required researching a wide range of different topics and technologies for the Physical and Simulated parts. The Verilog code for data processing was intentionally shared between the two domains. This requirement aims to prove that the same Verilog application may run on both the physical FPGA toolchain and the simulated FPGA toolchain without modification.

2.3 Methodology

The Spiral model, shown in Figure 2.1, first described by Barry Boehm in his 1986 paper, ”*A Spiral Model of Software Development and Enhancement”* [7] was used throughout this project as the main development process model, it is a risk-driven incremental development system based on four phases: Planning, Design, Construct and Evaluation. Every incremental cycle of development engaged in each of these steps as a way to reach certain milestones.

This model was complemented by weekly meetings with the tutors where we reviewed the results of the previous cycle and established well defined objectives to be undertaken by the next one. This model allowed me to tackle very complex issues in a manageable way.

Researching was a major part of the work done for this project as many of the topics investigated represent the state-of-the-art of their respective fields, particularly for the free and

open source FPGA community. There was a general lack of information and online community support, consequently, reaching dead-ends and relying on work-arounds were pretty common situations encountered in many of the paths taken towards achieving results.

All the software and code developed was archived in a personal Github repository created for this project[1] were most of the design, prototyping and testing was carried out. Working applications and examples were integrated into the JdeRobot FPGA-robotics Github repositories[2] through ”issues” and ”pull-requests” that were reviewed by other contributors to the organization.

During this whole process an online blog[3] was kept that can be accessed in the project's Github repository, where one can take a deeper look at the research and development process.

[1] https://github.com/RoboticsLabURJC/2017-tfg-richard-nicklas

[2] https://github.com/JdeRobot/FPGA-robotics

[3] https://roboticslaburjc.github.io/2017-tfg-richard-nicklas/logbook/

Chapter 3

Tools and Framework

This chapter portrays the most important tools that allowed the development of this project. The achieved results represent a new layer in a big technological pile that uses multiple programming languages, libraries/APIs, frameworks and general and specific use software and hardware.

3.1 Development Environment

3.1.1 Programming Languages

Verilog

Verilog[1] is a Hardware Description Language (HDL). HDLs are used to model the electronic systems and digital logic circuits that exist on FPGAs, they represent a high-level abstraction of the logic designs that are synthesized onto FPGA boards. Verilog modules handle most of the data processing and computer vision algorithms throughout this project and are used in both physical and simulated applications.

Verilog differs from conventional programming languages in that the execution of statements is not strictly linear. Verilog designs consist of a hierarchy of modules. Each module is defined with sets of input, output and bidirectional ports. Internally a module contains a list of wires, registers and the combinational logic in charge of performing tasks.

Concurrent and sequential statements define the behavior of the module, describing the relationships between ports, wires and registers. Sequential statements are placed inside begin/end

[1] `https://www.verilog.com/`

blocks and executed in sequential order, but all concurrent statements and all begin/end blocks are executed in parallel in the design. Modules may contain one or more instances of another module to define a sub-behavior.

Verilog IEEE 1364-2005 was selected to be used in this project as it is one of the most popular HDL available and it is fully compatible with the open source toolchain used for developing the programs discussed in this paper.

C++

C++[2] 11 is known as one of the best alternatives for coding time-critical applications and has the advantage of being a very popular library/API language. In this project C++ 11 is used mainly for the FPGA simulation applications. In plugins interconnecting the different simulation platforms and to provide additional logic to the system.

Python

Python[3] version 3.6.5 played an important role as a scripting and testing tool. Some examples of use were image format manipulation, testing hardware like the GoPiGo 3 robot or compilation and build scripts for the simulation environment.

3.1.2 OS and Development tools

Linux distributions like Ubuntu 20.04, CentOS 8, and Raspberry Pi OS were used as the main development environments for simulated applications and testing while the IDE for FPGA design and coding "Icestudio" ran on Windows.

Git/Github[4] was used for version control of this project's repository and also to work and contribute to JdeRobot's [5] organization repositories.

Sublime text 3[6], Visual Studio Code[7] and plain Vi/Vim[8] served at times as IDEs for C/C++

[2] https://isocpp.org/
[3] https://www.python.org/
[4] https://github.com/
[5] https://jderobot.github.io/
[6] https://www.sublimetext.com/
[7] https://code.visualstudio.com/
[8] https://www.vim.org/

and Python development. Most of the Verilog/HDL development was done on Icestudio, Sublime Text 3 and EDAPlayground.

3.2 Applied Software and Frameworks

3.2.1 IceStudio

Icestudio is a open source graphic IDE for open FPGAs released by the FPGAWars organization[9]. It includes driver configuration, complete tool-chain installation, graphical design, build, verification and loading into FPGA hardware boards. It also provides some basic building blocks and functionalities to help develop FPGA applications.

As a graphic IDE it greatly simplifies some of the most complex aspects of FPGA programming and has the advantage of being fully compatible with the open source FPGA boards used for this project, the Icezum Alhambra I and II.

The Icestudio IDE is software still in development, the first versions used, 0.3.3 and 0.4.0, presented some major bugs and the lack of a big user base or community behind to provide support proved to be at times a challenging environment to work with. Latter versions tried like the 0.5.0 addressed some of these issues and allowed the project to progress. A screenshot of the IDE version 0.4.0-dev can be seen in Figure 3.1

In this project Icestudio served as the main integration development environment for the ”physical” FPGA applications. It covered the instantiation and integration of different Verilog modules. Also the compiling, building, synthesizing and uploading of the programs to the real FPGA.

3.2.2 Verilator

Verilator[10] v4.216 is a free and open source software tool which converts synthesizable Verilog code to cycle-accurate behavioral models in C++, thus allowing the simulated Verilog modules to run on regular processors and providing a way to easily interface with its inputs and outputs.

With Verilator we can achieve a completely simulated execution environment, the advan-

[9] https://github.com/FPGAwars/icestudio

[10] https://www.veripool.org/verilator/

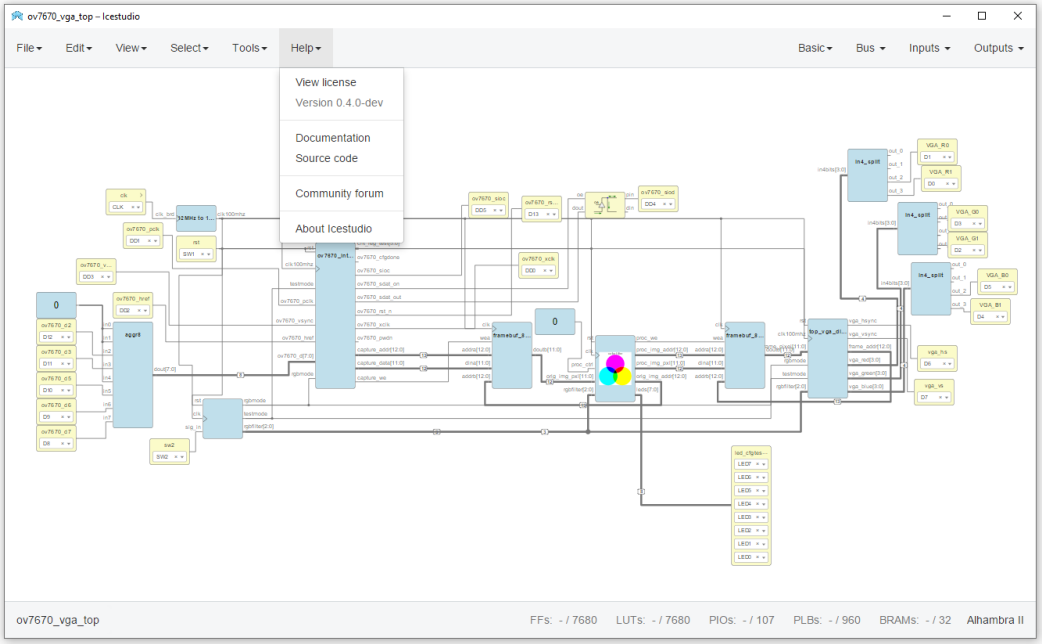

Figure 3.1: Icestudio IDE v0.4.0-dev

tages this provides for the development of FPGA applications cannot be overstated. Reduced costs as no hardware is needed, easier development as C++ can be used to interface with the modules and possibly most important, accurate, fast and replicable testing and debugging.

3.2.3 ROS and Gazebo

- **ROS**

  The Robot Operating System (ROS)[11] is an open source set of software libraries and tools that help build robot applications. It works like a meta-operating system and provides functionality like hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes and package management. ROS is rather complex and has a somewhat steep learning curve but adds a lot of value to many robotics applications.

  For this project ROS version ”Noetic Ninjemys” is used to define and implement simulated robots that can test the functionalities developed in Verilog modules and as the messaging application connecting the different processes.

[11] `https://www.ros.org`

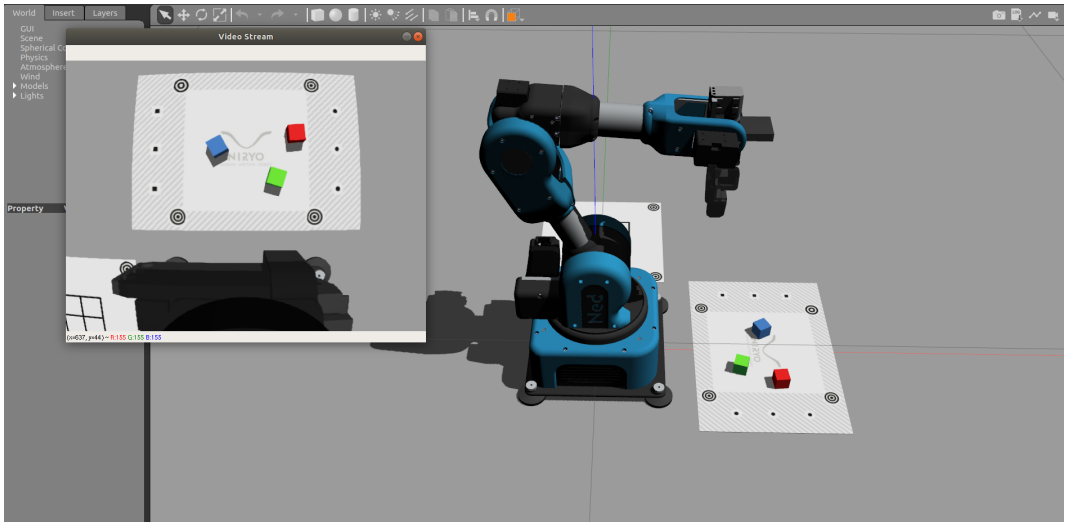

Figure 3.2: Simulation of a robotic arm on the Gazebo software

- **Gazebo**

  Gazebo[12] is an open source 3D robotics simulator, an example can be seen in Figure 3.2. It provides realistic rendering of environments including high-quality lighting, shadows, and textures. It can model sensors that ”see” the simulated environment, such as laser range finders, Kinect style sensors and most important to this project, cameras.

  In this project Gazebo version 11.11.0 is used to simulate robots and the environment from which a simulated FPGA can gather information for image processing and object detection.

With ROS and Gazebo a complete framework capable of accurately simulating robots and their environment is achieved. This framework is capable of both sending and receiving data from different processes running other parts of the system like a OpenCV webcam plugin or Verilog code running on a simulated FPGA.

3.2.4 OpenCV

OpenCV[13] (Open Source Computer Vision Library) is an open source library of programming functions mainly aimed at real-time computer vision. OpenCV primary interface is written in C++ which eased development of this project. OpenCV is used to handle and transform the

[12] https://gazebosim.org/
[13] https://opencv.org/

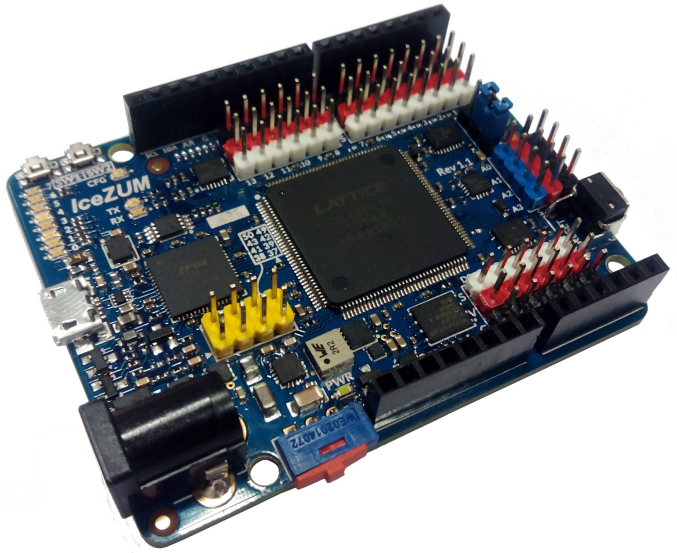

Figure 3.3: Icezum Alhambra v1.1.

image related data that is transferred between the different parts of the software. It also provides complex image processing functions that can be used to compare and test similar functionality developed for FPGAs.

For the simulation part of this project version 4.2.0 of OpenCV is widely used as an image and video handling tool. Examples of this are the integration of webcams as C++ plugins or the resizing of the recorded videos to fit the Verilog modules interface.

3.3 Hardware

3.3.1 FPGA Boards

The Icezum Alhambra I, and Icezum Alhambra II[14] (figures 3.3 and 3.4) electronic boards are open source both in software and hardware, and are compatible with open source toolchains, fulfilling one of the main goals in this project of using free and open source resources. Both come with an iCE40HX4K embedded FPGA from Lattice, 20+ Input/output 3.3/5V pins, multiple general-purpose LEDs and buttons, USB power supply and more.

These boards were designed by the FPGAWars organization following the open source movement spirit for educational purposes and therefore, are not particularly powerful nor have

[14] `https://alhambrabits.com/alhambra/`

Figure 3.4: Icezum Alhambra v2.

the best technical specifications available on the market. Still, they are fully compatible with the IceStudio IDE and are suitable for the proof of concept designs developed in this project.

3.3.2 GoPiGo 3 Robot

GoPiGo 3[15] is a differential drive robot manufactured by Dexter Industries that offers a wide range of sensors and actuators to assemble on it. The GopiGo 3 runs on an operating voltage of 7-12V and comes with two 66.5mm wheels attached to two direct current (DC) motors, odometry is provided by two magnetic encoders capable of 720 pulses per wheel rotation. It is ROS compatible and comes with a very complete set of libraries to run software in several languages on its onboard Raspberry Pi 3.

The robot comes with a proprietary board that connects via two I2C ports, a serial port (SPI) and two analog digital ports to external devices like the mentioned Rasperry Pi. For this project project the Raspberry Pi was only used for testing purposes as the Robot's "brain" was switched for the Icezum Alhambra II to serve as the image data processor, to control the different sensors, actuators and the GoPiGo 3 proprietary board with its FPGA.

[15] `https://www.dexterindustries.com/gopigo3/`

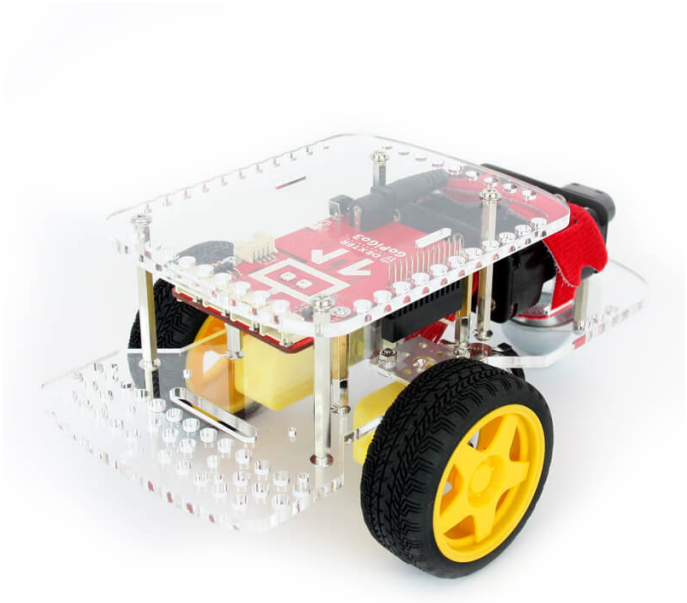

Figure 3.5: GoPiGo3 Robot

3.3.3 OV7670 Camera

The OV7670 [9] is a low cost CMOS image sensor that provides the functionality of a single-chip VGA camera and image processor. It provides full-frame 8-bit images in multiple formats, controlled through the Serial Camera Control Bus (SCCB) interface. The camera operates at up to 30 frames per second in VGA and allows for user control over image quality, formatting and output data transfer. Image processing functions like exposure, gamma, white balance, color saturation, hue control and more, are also programmable through the SCCB interface.

The OV7670 Camera was extensively used as the main image data producer, its small footprint and specification requisites were ideal for the Icezum Alhambra II board and it proved powerful enough to handle the proof of concepts and tests undertaken during this project.

3.3.4 Aditional HW, Electronic Circuits and Signal/Logic Analyzer

All the different hardware components were manually interconnected with 10/15 centimeter jumper wires. A protoboard was used to extend the circuits allowing the use of a signal analyzer. The "AZ-Delivery Logic Analyzer" is an 8 channel logic analyzer purposed for analog data recording, this device connects to a PC over an USB port and uses the "Saleae Logic Soft-

Figure 3.6: OV7670 Camera

ware" to record signals traveling between the different hardware components. Interpreting and analyzing these signals proved essential for the configuration of the hardware components and the debugging of the software application.

Some additional hardware was used, like a VGA adaptor to display the FPGA processed images on a monitor or some custom PCBs that helped with hardware interconections.

Chapter 4

Physical FPGA

This chapter's main objective is researching and developing Computer Vision algorithms and other functionalities as "blocks" to be included in FPGA applications. Also designing and testing some working examples that make use of these and several other blocks in *physical FPGA boards and real robots* that incorporate them.

4.1 FPGA-Robotics

The developments presented in this chapter take place in the JdeRobot organization FPGA-Robotics project. As explained in the introduction, FPGA-robotics is a development environment that intends to create open source FPGA tools, libraries and reusable nodes for Robotics.

When this project started there were already several blocks and tools developed, as well as some applications or Proof of Concepts (POCs) by other collaborators in the FPGA-Robotics repository. General support functionality was provided by Icestudio Verilog modules or blocks that can be easily incorporated in IceStudio projects; examples of this are addition, multiplication, square roots and other mathematics related functionality blocks, or bit manipulation modules that covered assignation and counters. Other general functionality blocks provided more complex behaviours like SCCB/I2C interfacing and communication, or PWM (Pulse-Width Modulation) Control. Some blocks provide functionality to integrate specific peripherals like displays through VGA or connecting to a OV7670 camera[1].

[1] `https://github.com/JdeRobot/FPGA-robotics/tree/master/blocks/icestudio/icestudioCollection/FPGA-Robotics-v1-stable/blocks`

A self-balancing robot that solves the inverted pendulum control problem has been successfully programmed using an open FPGA toolchain by Juan Ordoñez et al [5]. The code can be found in FPGA-Robotics repository[2]. The application includes a perception module for an inertial sensor, a proportional–derivative controller, and a driver module for two DC motors. The system includes an Arduino micro-controller for the sensor driver and an Icezum Alhambra board (with an iCE40 FPGA from Lattice) as the primary computing unit. All of the FPGA modules were developed using Icestudio.

Researching and analyzing this previous work served as the basis for the developments described in this Final Degree Project. The present work constitutes a step forward from the previous work developed on FPGA-Robotics, demonstrating that using open source tools for reconfigurable computing is a viable alternative for the development of robotics and machine vision applications.

Icestudio Icestudio is the main IDE used for the developments presented in this chapter. It covers the whole cycle of development, from module design and coding to the upload of the software on an FPGA. Icestudio provides a visual interface to easily integrate different Verilog modules or blocks like the ones described before into a single application to be uploaded to a real FPGA.

These modules are compiled, synthesized and analyzed in search of syntax errors, impossible to generate constructions (logically functional hardware, but with no possibility of being synthesized on the target FPGA) or timing problems (time related issues regarding the execution speed of the logic blocks).

During the FPGA synthesis process, an RTL (Register Transfer Level) or HDL design is converted into a gate level representation or a logic component. This means that FPGA code provided in high level language such as Verilog is converted into logic gates. There are 3 main steps undertaken during FPGA synthesis

- **Instantiation and Substitution**

  A library consisting of primitive modules such as Boolean functions including AND, OR,

[2] `https://github.com/JdeRobot/FPGA-robotics/tree/master/phys_fpga/old/Self-Balancing_Robot`

NOR, etc., and other modules that have been defined by the user are created. These primitive modules are further expanded by adding language operators and constructors.

- **Inference**

  The synthesis tool detects and recognizes particular patterns in the Verilog logic design and treats them in particular user defined manner.

- **Logic Optimization**

  In the final step the Boolean operations are grouped together in various combinations using logic optimization techniques.

Afterwards, the synthesized elements are placed and connected to the FPGA blocks checking availability and feasibility. The process ends with the generation of the bit stream to load into the FPGA. This process has to be performed every time changes are made to the design or code.

4.2 Physical Blocks

Several modules covering very different functionality were researched and developed with the intent of achieving complex image processing applications for robots in an open source FPGA. All of them will be further described in detail in this section.

- **Basic Image Processing Blocks**

  Research and development of simple Verilog modules to test FPGA functionalities on an online IDE and test bench application called ”EDAPlayground”. Achieving automation of image file reading, writing and encoding. Developing basic image processing(color filtering) blocks.

- **Camera and Display Blocks**

  Transitioning to a real FPGA. Integrating VGA Display modules and connecting a VGA output and display to the FPGA. Afterwards, including real input data from a camera into the system, now capable of recording, processing and displaying live video with an open source FPGA board.

- **SPI Communication Block. GoPiGo 3 robot integration**

  Transition to the Icezum Alhambra II and developing Serial Peripheral Interface (SPI) communication modules for the ”GoPiGo 3” robot.

- **Improved Color filter and Object detection Block**

  Developing a color filtering and histogram-based object detection block.

4.2.1 Basic Image Processing Blocks

Several Modules were developed to test basic functionalities of FPGAs and achieve simple image processing. After considering multiple options, an online IDE and test-bench called EDAPlayground[3] was selected to develop, simulate, synthesize and test the first Verilog modules. EDAplayground provides a versatile development environment that was enough to fit the early projects needs.

The following blocks were created and are archived in the FPGA-Robotics Github repository[4]

- **Logic Gates**

  Verilog code and test-bench to test basic HDL behaviours like inputs/outputs and some simple logic operations like AND, NOR, OR, etc.

- **RAM and RGB RAM**

  Two modules were developed to provide data storage and memory access on Verilog. One for simple data (ram.v) and a custom data storage to handle 24bit-RGB data (rgb_ram.v)

- **Hex to RGB**

  A new module capable of reading hexadecimal data from a .BMP image file and store it in the RAM/rgb_ram.v module.

- **Color Filter**

[3] https://www.edaplayground.com/

[4] https://github.com/JdeRobot/FPGA-robotics/tree/master/phys_fpga/old/Basics

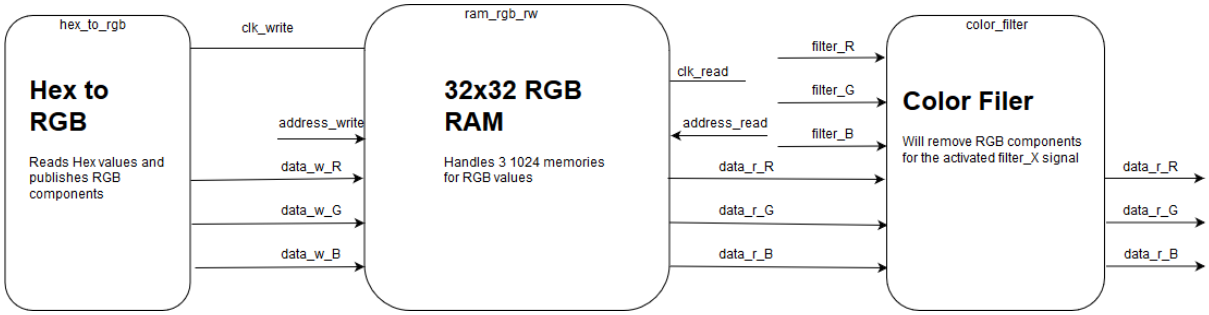

Figure 4.1: HexImageFilter module diagram

A final module that provides the functionality of color filtering was included to the repository. It reads the RGB data from the memory storage block and applies a simple RGB mask to delete a specific color component.

An application called ”HexImageFilter” incorporates the mentioned blocks into a Verilog Module capable of simple image processing, the code can be accessed here[5]. A diagram of the three modules, their interfaces and how they interconnect is shown in Figure 4.1.

4.2.2 Camera and Display Blocks

The main goal in this section is the transitioning to a real FPGA board and incorporating real input and output device blocks to start physical testing of the image processing Verilog modules.

The project was designed to be uploaded to an Icezum Alhambra I. This board is quite restricted in hardware specifications so the image processing had to be limited in resolution, size and frame rate. This also meant that both the input and output video streams needed trimming in order to be successfully stored, processed and displayed.

4.2.2.1 VGA Output

A Verilog module was developed and included into the project that allowed images to be displayed through VGA[6]. This node handles the synchronization signals, hsync and vsync, to

[5] https://github.com/JdeRobot/FPGA-robotics/tree/master/phys_fpga/old/ComputerVision/HexToVGA/examples

[6] https://github.com/JdeRobot/FPGA-robotics/tree/master/phys_fpga/old/ComputerVision/VGA

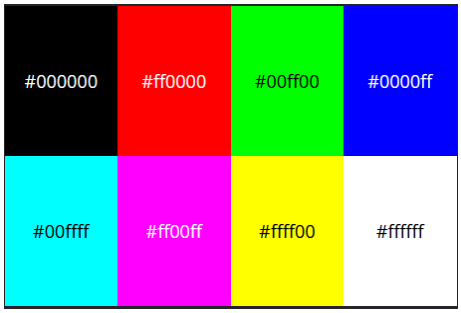

Figure 4.2: 3-bit RGB Color palette

produce a video array visible on a PC monitor. This module is very useful for debugging of vision-based applications.

The first prototype application that made use of this module read a simple 170x300 binary map to test the correct behaviour of the VGA block. The hardware restrictions of the Alhambra I board meant that the Verilog modules could only process 3 bits of data per pixel for images of this size, thus producing output in what is known as a 3-bit RGB, Figure 4.2.

The input image in this prototype was monochrome (1-bit grayscale), so even though the FPGA modules could process and display 8 colors per pixel, the filtered image shows only two of them.

The next iteration included a 32x32 hex image as input with 24 bits of RGB data per pixel, this color palette is known as ”truecolor” and is used in most modern display systems. As only 3-bit color can be displayed with the Verilog modules, simple RGB filters needed to be implemented to display a reduced version of the image. As can be seen coded in Figure 4.3, color components exceeding a 50 % threshold on its RGB 8 bit value were translated a 1 or otherwise as 0, effectively subsampling from 16,777,216 colors to 8.

The results of applying 3 of the 8 different RGB masks possible to an example 32x32 image can be seen in Figure 4.4. Filtering the R, RB and G components respectively.

All the data processing in these modules was done on the spot. Pixels were read, processed and displayed one after the other without any notion of the previous pixels or frames. This architecture is enough for simple image processing but the aim of this project is to use more complex computer vision algorithms like convolutions and complex filters. These operations require access to different parts of the images data at the same time, making storage a necessity.

The last iterations of this example incorporated the rgb_ram.v Verilog module from the

clk
clr
x_img[9:0]
y_img[9:0]
R[5:0]
G[5:0]
B[5:0]
pixel[2:0]
addr_read[11:0]

```
1  //Color filter 0-255
2  parameter Rmin = 8'b10000000;
3  parameter Gmin = 8'b10000000;
4  parameter Bmin = 8'b10000000;
5
6  parameter width = 32;
7  parameter height = 32;
8
9  //Px Coords
10 wire [9:0] x_img;
11 wire [9:0] y_img;
12 wire [5:0] R;
13 wire [5:0] G;
14 wire [5:0] B;
15 reg [2:0] pixel;
16
17 assign addr_read = x_img*y_img;
18
19 always @(posedge clk) begin
20
21     pixel[0] = 1;
22     pixel[1] = 1;
23     pixel[2] = 1;
24
25     if ((x_img >= 0) && (x_img < width ) && (y_img >= 0) && (y_img < height))
26     begin
27         if(R < Rmin)
28             pixel[0] = 0;
29         if(G < Gmin)
30             pixel[1] = 0;
31         if(B < Bmin)
32             pixel[2] = 0;
33    end
34     else
35     begin
36         //rest of the screen
37         pixel = 3'b111;
38
39     end
40
41 end
```

Figure 4.3: Color Filter Verilog Code

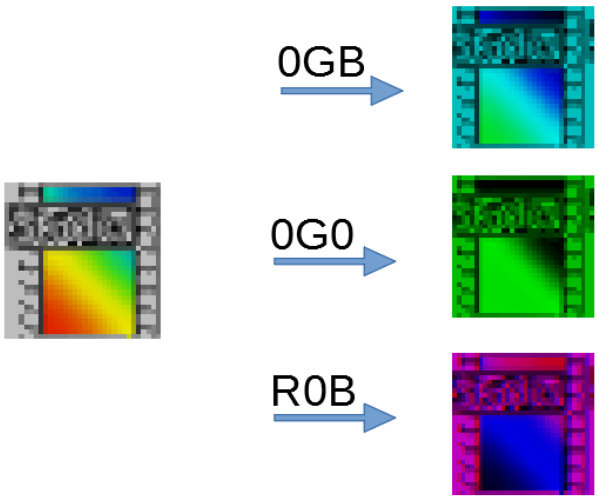

Figure 4.4: Three examples of RGB Filtering.

previous section to provide storage capabilities to the system. The final module was capable of storing a complete frame of data per cycle consisting of 32x32 24-bit RGB pixels. The declaration of the memory register is shown in the following code snippet.

```
parameter sizeOfLengthReal = 3072; // image data : bytes: 3073 32 * 32
    * 3 (RGB components)

// Memory
reg [7 : 0] img_storage [0 : sizeOfLengthReal-1];// memory to store
    24-bit data image
```

4.2.2.2 Camera Input

This section's main objective is to start using real input data from a camera instead of hexadecimal or binary images. After incorporating the OV7670 block a system capable of recording, processing and displaying live video with an open source FPGA is finally created.

The OV7670 camera is used for this purpose. As described in previous chapters, the Alhambra I has serious hardware limitations so the default behavior of the OV7670 overwhelmingly outperforms its capabilities. For this reason a modified version of the camera module developed by JdeRobot's team was included into the system[7]. The data stored in the FPGA is 18 bit per pixel (instead of 24bit per pixel) and only a 60x80 pixel subset of the recorded image is used for image processing and display.

The module added to the system handles the configuration and initialization of the camera and feeds the video stream to the Image Processing modules. A signal analyzer has become a necessity to be able to debug these behaviours.

A Logic Analyzer and the "Saleae Logic 1.2.18" software were used for sampling and decoding of the different input and out channels that connect to the OV7670 camera. This communication was using the SCCB compatible I2C (Inter-Integrated Circuit) protocol. An example of the recorded signal and decoding can be seen in Figure 4.5.

At first glance it seemed that the I2C protocol was working as intended but sampling differ-

[7] https://github.com/JdeRobot/FPGA-robotics/tree/master/phys_fpga/alhambra_ii/icestudio

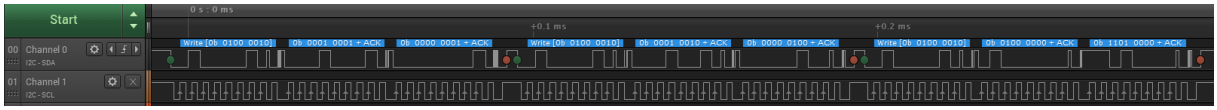

Figure 4.5: I2C sampling

ent channels like VSYNC (Vertical synchronization (Output)) and HREF (Horizontal synchronization (Output)) noise could be seen polluting the signals.

Several hardware upgrades were needed to finally achieve a working camera input, the Icezum Alhambra I was replaced with the Icezum Alhambra II with better specs. The length of the wires connecting the different components of the system, that up until that point were 20cm long, were replaced by 10cm ones to reduce the degradation of the signals, redundant connections were removed and a 6-bit VGA adapter was incorporated to manage the reduced image size our sobel/color filters were producing. The end result was a much cleaner solution capable of recording, processing and displaying video stream.

4.2.3 SPI Communication Block. GoPiGo 3 robot integration

The Camera and FPGA board were incorporated into a GoPiGo 3 robot with the aim of using the processed image data to feed linear and angular velocity to the robot's motors.

After setting up the robot and testing its native applications the next steps were to unravel the different layers that exist between the Python code used to control its motors and actuators. The idea is to identify the point where the FPGA board will fully replace the Raspberry Pi that was doing all the processing up to that point.

The Raspberry Pi uses an SPI interface to connect to the GoPiGo board. The Serial Peripheral Interface (SPI) is a synchronous serial communication interface specification used for short-distance communication used mainly in embeded systems. It follows a master-slave architecture and uses 4 signals; Serial Clock (SCLK), Master Out Slave In (MOSI), Master In Slave Out (MISO) and Slave Select (SSEL).

By analyzing the SPI communication performed by the GoPiGo c++ libraries instructions a new SPI Verilog module could be developed whose job was to replace the initialization and instruction mechanisms that up until that point were being handled by the Raspberry Pi and from now on would be done by the FPGA. This reverse-engineering task was necessary because there

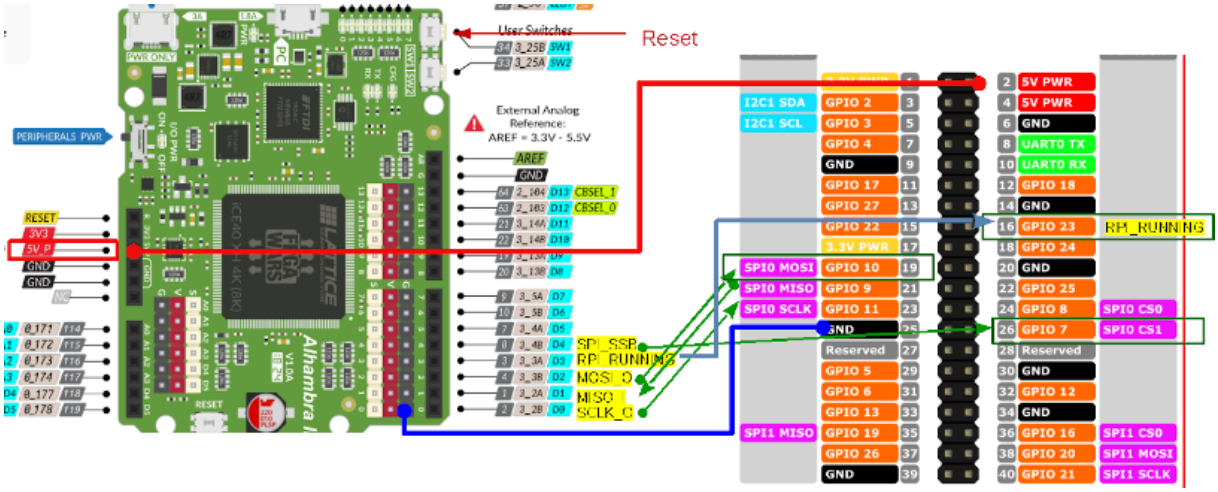

Figure 4.6: Alhambra II - GoPiGo3 SPI connections

was no available documentation or code for the GoPiGo 3 proprietary board. In Figure 4.6 an schema of the connections between the Alhambra II board and the GoPiGo 3 board is shown.

A Verilog module from the open source "FPGA Libre" organization[8] called SPI_Master was selected and expanded to facilitate communication between the Alhambra II and the GoPiGo's board. A block diagram of this module can be seen at Figure 4.7.

The GPG3 folder[9] in my personal repository for this project contains all the developed code and modules that were produced with the intent of creating the GPG3 SPI communication POC.

Even though my involvement with this particular block was more focused on the SPI communication itself and developing and testing basic applications like using this module to light GoPiGo's LEDs, this work was later used by my tutor Felipe Machado to produce the final blocks that handle SPI communication. They can be found in FPGA-Robotics repository[10]. These blocks were later expanded to handle further functionality of the GoPiGo like Motor control.

[8] http://fpgalibre.sf.net

[9] https://github.com/RoboticsLabURJC/2017-tfg-richard-nicklas/tree/main/GPG3

[10] https://github.com/JdeRobot/FPGA-robotics/tree/master/blocks/verilog/utils/spi/alhambra_ii

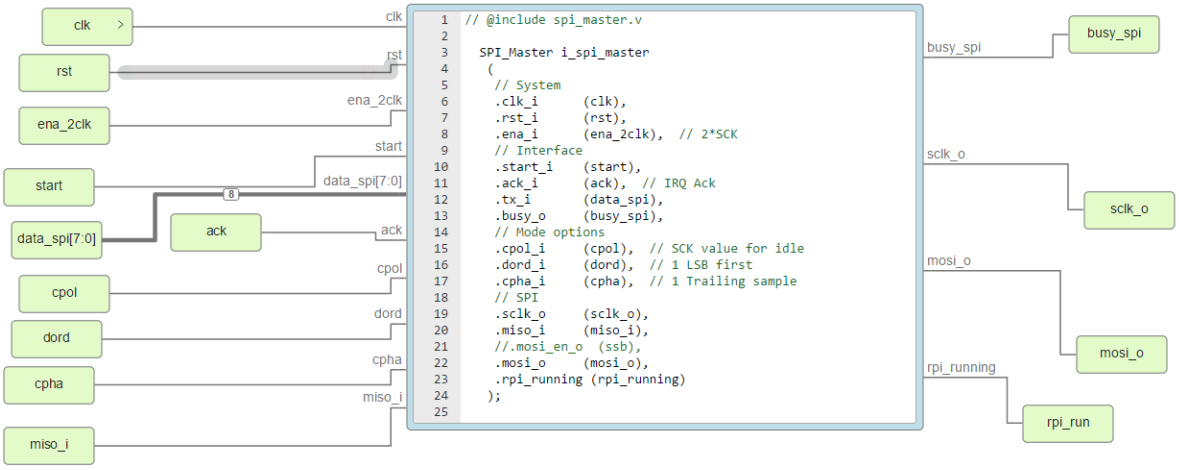

Figure 4.7: Interface of the spi_master block shown in Icestudio

4.2.4 Improved Color Filter and Object Detection Block

This section relies on the research done to produce a block that intends to tackle the classical problem in computer vision and image processing of object recognition and detection. Or in other words, determining whether or not the image data contains some specific object, feature, or activity.

The best algorithms for such tasks are currently developed on convolutional neural networks. Algorithms based on Fast R-CNN (Fast Regional-Based Convolutional Neural Networks) or single-shot detectors like YOLO (You Only Look Once) are already close in performance to humans. Other solutions rely on Histogram-based techniques for object detection. Mechanisms like SIFT (Scale-Invariant Feature Transform) or feature descriptors like HOG (Histogram of Oriented Gradients) rely on pixel histograms to locate local features in images for the purpose of object detection.

Due to the constrains that FPGA development entails the engineered block in this section is designed as a proof of concept for histogram-based object recognition yet not as complex as the algorithms mentioned in the previous paragraph.

4.2.4.1 Improved Color filter

The image processor Verilog module color_proc.v expands on the implementation discussed in previous sections to provide more complex RGB based filtering. This version receives the original pixel and a 3 bit signal to select the colors to filter as input and establishes the value or the processed pixel to be displayed by the VGA module as output.

The 3 bit mask signal enables filtering for the 8 color values shown in Figure 4.2

The following Verilog code snippet exemplifies how this process is done inside the module.

```
always @ (orig_pxl, rgbfilter) // should include RGB mode
begin
  case (rgbfilter)
    3'b000: // No filter, output same as input
      proc_pxl <= orig_pxl;
    3'b100: begin // Red filter
      if (orig_pxl[c_msb_red])
        proc_pxl <= orig_pxl;
      else
        proc_pxl <= BLACK_PXL;
    end
    3'b010: begin // Green filter
      if (orig_pxl[c_msb_green])
        proc_pxl <= orig_pxl;
      else
        proc_pxl <= BLACK_PXL;
    end
    3'b001: begin // Blue filter
      if (orig_pxl[c_msb_blue])
        proc_pxl <= orig_pxl;
      else
        proc_pxl <= BLACK_PXL;
    end
    3'b110: begin // Red and Green filter
      if (orig_pxl[c_msb_red] & orig_pxl[c_msb_green])
        proc_pxl <= orig_pxl;
      else
```

```
        proc_pxl <= BLACK_PXL;
      end
      3'b101: begin // Red and Blue filter
        if (orig_pxl[c_msb_red] & orig_pxl[c_msb_blue])
          proc_pxl <= orig_pxl;
        else
          proc_pxl <= BLACK_PXL;
      end
      3'b011: begin // Green and Blue filter
        if (orig_pxl[c_msb_green] & orig_pxl[c_msb_blue])
          proc_pxl <= orig_pxl;
        else
          proc_pxl <= BLACK_PXL;
      end
      3'b111: begin // Red, Green and Blue filter
        if (orig_pxl[c_msb_red] & orig_pxl[c_msb_green] &
            orig_pxl[c_msb_blue])
          proc_pxl <= orig_pxl;
        else
          proc_pxl <= BLACK_PXL;
      end
    endcase
  end
```

4.2.4.2 Object Detection

The algorithm designed in this section tries to follow the theory behind Color Histograms [11] for object detection. In image processing and photography, color histograms are a representation of the distribution of colors in an image, counting the number of pixels present for each value of color range on each image row or image column.

This implementation counts the pixel distribution over the width of the frame. The most prominent object's centroid can be extracted from the histogram by locating the highest concentration of filtered pixels and the distribution of the surrounding values. The total number of pixels filtered can help identify the object's size relative to the camera, and somehow, the

[11] https://en.wikipedia.org/wiki/Color_histogram

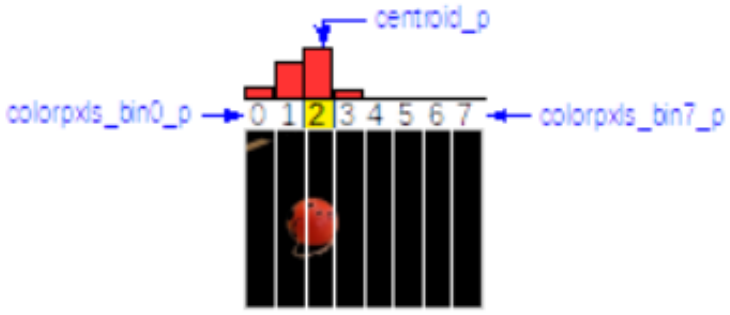

Figure 4.8: Histogram of filtered Red Pixels

distance to that object.

This particular implementation is intended to be integrated into a wheeled robot with an onboard camera, like the GoPiGo 3 discussed in previous sections, to guide it to the detected objects. For this reason the location of the object on the vertical plane is deemed less relevant and only the distribution on the horizontal plane is analyzed. The image processing module was expanded to record histograms of filtered pixels for each image frames processed. A graphical representation of this structure can be seen at Figure 4.8.

A minimum pixel threshold can be applied to each column in the histogram to filter noise. This threshold should be fine tuned for the specific camera used and the environment conditions to help in the selection of the desired objects.

The following Verilog code snippet shows how the filtered red pixels are stored in the ”histogram” register.

```
 reg [5:0] histogram [79:0];

//histogram stores the filtered pixels in each column, resets every frame.
always @ (posedge clk, posedge rst)
begin
  if (rst) begin
    for(i=0;i<=79;i=i+1) begin
      histogram[i] <= 0;
    end
  end
  else if (end_pxl_cnt) begin
    for(i=0;i<=79;i=i+1) begin
```

```
    histogram[i] <= 0;
   end
  end
  else begin
   if (orig_pxl[c_msb_red]) begin
    histogram[px_pos] <= histogram[px_pos] + 1;
   end
  end
end
```

This Verilog snippet shows how the histogram's bin with the highest value of pixels is stored for each frame.

```
 always @ (posedge clk, posedge rst)
begin
  if (rst) begin
    prev_high <=0;
    col <=0;
  end
  else if(tmpw) begin
     prev_high <= histogram[px_pos];
     col <= px_pos;
   end
end
```

The output for the object detection is an 8 bit register. Even though the histogram has information from the 80 pixel width of the image, this implementation outputs only 8 possible values that provide a single dimensional position for the detected object on the horizontal plane.

The following code snippet indicates the possible values for this module's output

```
// 1000 0000 left-most
// 0100 0000 more to the left
// 0010 0000 left
// 0001 0000 slightly to the left
// 0000 1000 slightly to the right
// 0000 0100 right
```

```
// 0000 0010 more to the right
// 0000 0001 right-most
```

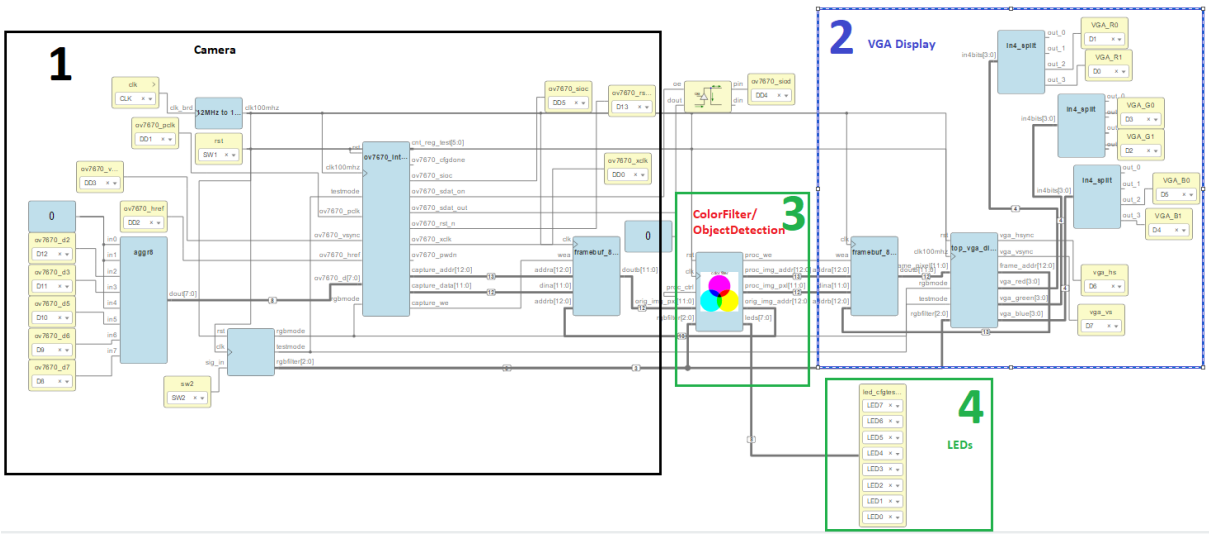

Figure 4.9: Object Detection POC Module Diagram

4.3 Applications

With the intention of demonstrating the use of the presented blocks on working applications two POCs will be described in this section.

4.3.1 Object Detection POC

Figure 4.9 shows the final Verilog module diagram on Icestudio for the Object Detection POC. The black box labeled ”1” contains the Camera modules (section 4.2.2.2) that manage the OV7670 camera and feed data to the image processor module. The blue box labeled ”2” includes the VGA display modules (section 4.2.2.1) that receive processed pixels to be presented on a monitor. The green box labeled ”3” contains the main image processor, the color filter and object detection module explained in section 4.2.4. Finally, also in green with the label ”4”, the LED output in charge of displaying the detected objects on the horizontal plane.

In Figure 4.10 a schema of the connections between the Alhambra II board, the OV7670 Camera and a VGA connection is shown.

The presented solution manages to achieve accurate results in object detection as is demonstrated though the next experimental example. A video recording can be found in YouTube[12]

Footage of a finger moving over a black background is recorded. Using the OV7670 Camera

[12] https://www.youtube.com/watch?v=ioeNptRcPDY

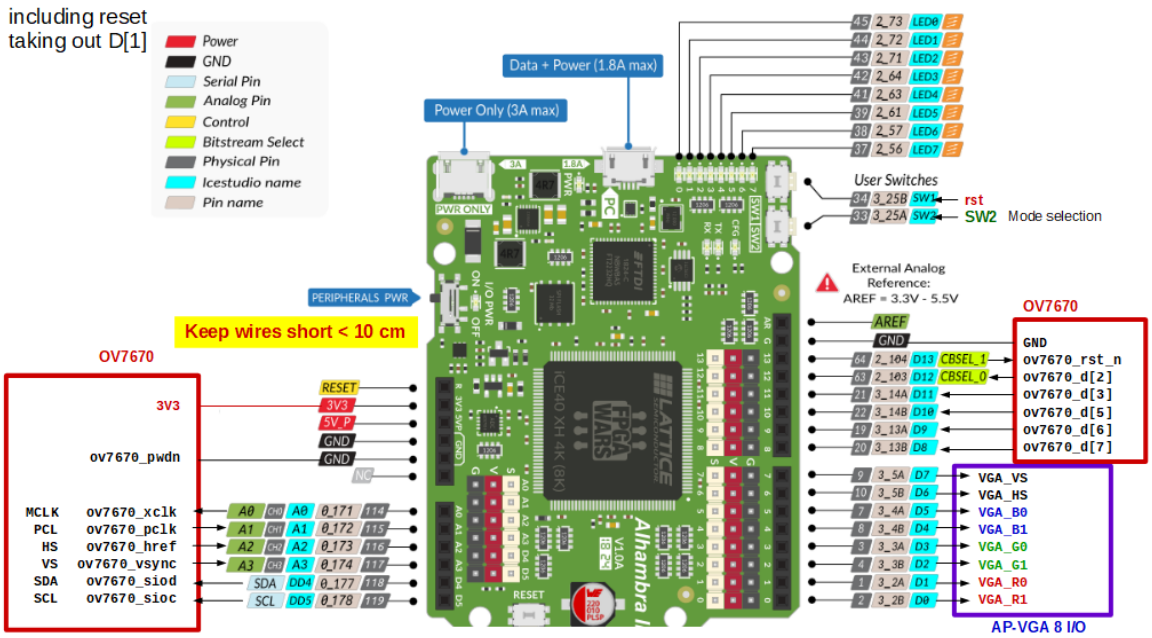

Figure 4.10: VGA - Alhambra II - OV7670 connections

connected to the Icezum Alhambra II and the Camera Verilog modules the application receives the recorded image data. Afterwards, the images are handled to the image processor module to filter red pixels and detect the position of the filtered object on the horizontal plane relative to the camera's frame.

With the help of the VGA display Verilog modules the processed images can be seen on a computer monitor through VGA. Figure 4.11 shows two images of the filtered video output being displayed though VGA on a PC monitor. Note that only the red pixels pass the color filter.

Figure 4.11: Filtered Video Output

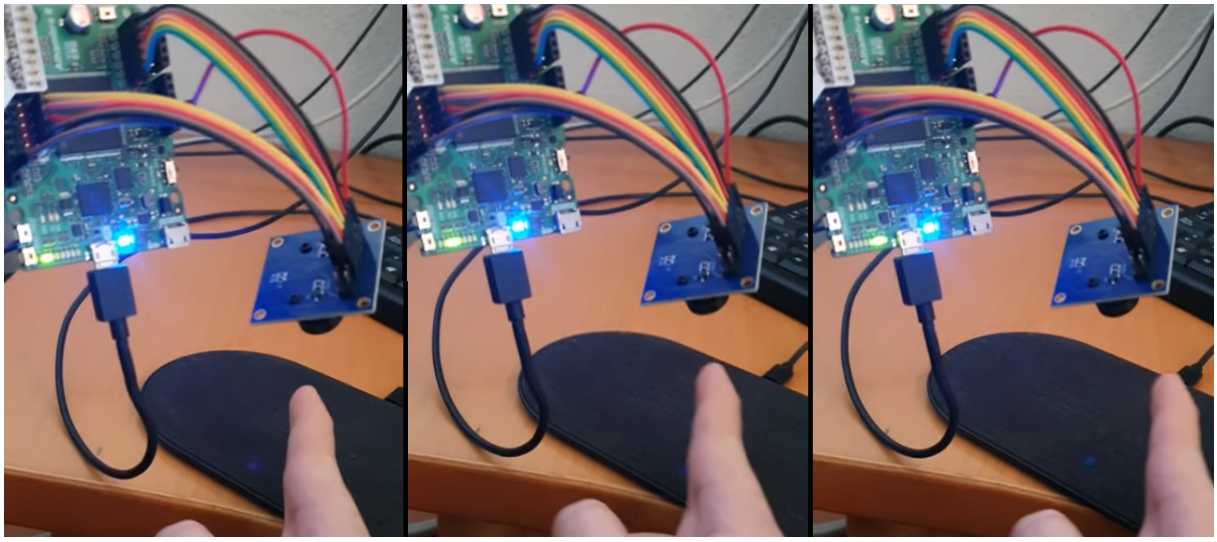

Figure 4.12: Object Detection on Green LEDs

The 8 bit object detection output is also presented on the FPGA board 8 general-purpose green LEDs.

Figure 4.12 portrays the recording of the scene and the output of the object detection is wired to the green LED Display of the Icezum Alhambra II, note how the illuminated LEDs change depending on the position of the finger relative to the camera's frame.

This physical working application manages to achieve color filtering and object detection on an open source FPGA board and successfully integrates different peripherals like the Camera and monitor to the system. This implementation established the first steps for the development of more sophisticated filters and detection algorithms for FPGAs.

4.3.2 GoPiGo 3 robot following a color ball POC

This POC displays a working prototype for object tracking with a differential drive robot. A simplified block diagram of the physical system can be seen in Figure 4.13.

Some of the developments and analysis done for the integration of the Alhambra II to the GoPiGo3 robot (Section 4.2.3) in addition to subsequent iterations of other Verilog blocks discussed in this document were integrated by colleagues from the JdeRobot organization to achieve a more complex and complete object detection POC for the GopiGo 3 robot. The beta-testing and refinement of parts of this application and its blocks was also done inside this Final Degree Project. The files and documentation can be found on the FPGA-robotics's "phys_fpga"

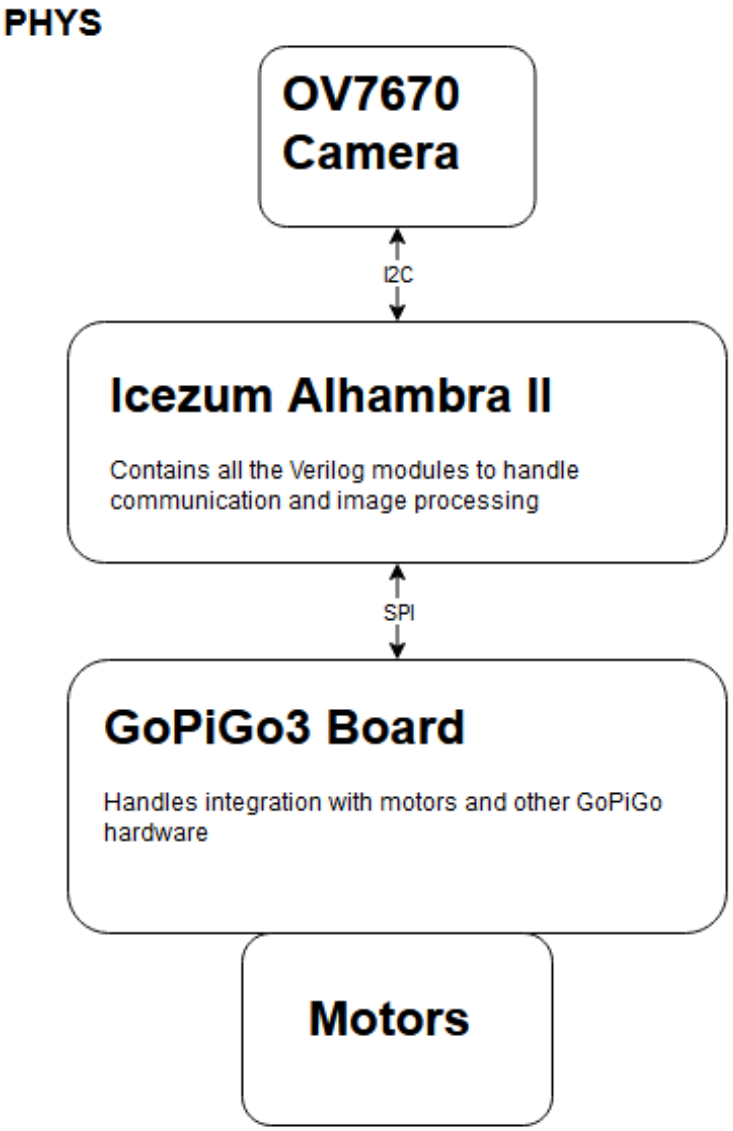

Figure 4.13: Simplified Physical model diagram

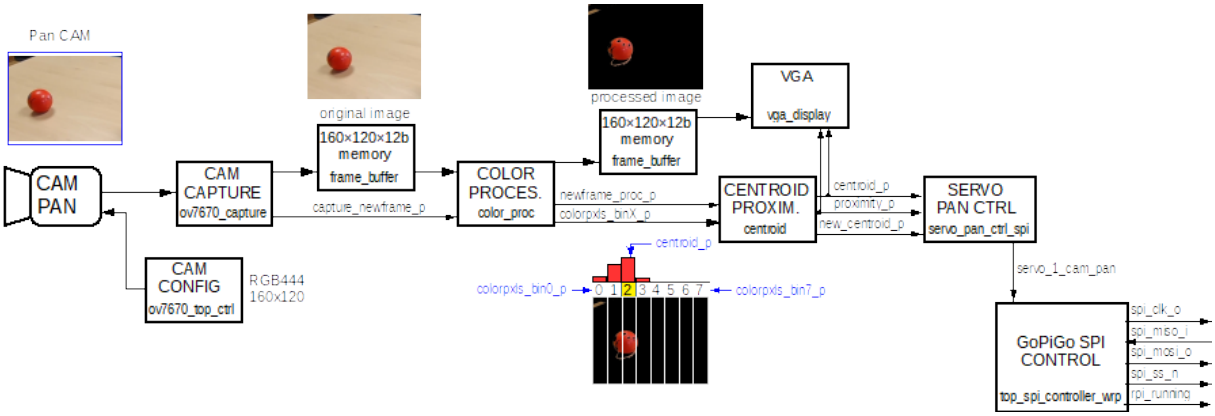

Figure 4.14: Blocks for camera guided robot

github repository[13]

Figure 4.14 displays a block diagram of the different components that make the Verilog application. Besides the functionalities already covered in previous sections of this document, this POC includes several new modules and improvements, both in software and hardware.

The most important blocks included in this POC are described below:

- **GoPiGo Blocks**:
  - **gopigo_spi_ctrller**: Receives Motor PWM and leds commands for the GoPiGo3, and sends them via SPI. Before sending them, it checks whether there has been any change since the last sending.
  - **motors_ctrl_v4**: Control for motors in closed loop with centroid and proximity. The speed of the motors is generated for the GoPiGo SPI Controller block.
- **Image Blocks**:
  - **ov7670 Camera**: Blocks related to the OV7670 camera.
    - **ov7670_iface**: OV7670 interface.
    - **framebuff_80x60_12b**: Buffer block for the image.
    - **ov7670_ctrl**: Module in charge of communicating with the SCCB module.
    - **ov7670_capture**: Block that manages image capture.

[13] https://github.com/JdeRobot/FPGA-robotics/tree/master/phys_fpga

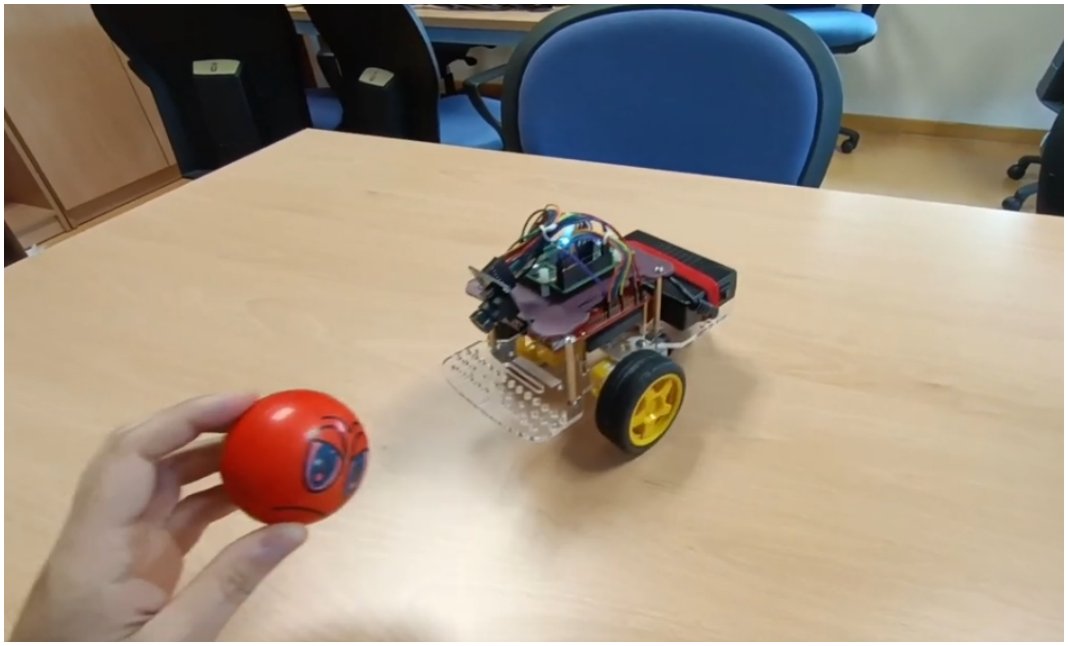

Figure 4.15: A GPG3 Robot using an FPGA to track a red ball

- **Processing**: Blocks related to image processing.
  - · **color_processing**: Basic version of image processing for the color filter.
  - · **color_processing_v3**: Advanced version of image processing for the color filter. The centroid and proximity are obtained at the output.
  - · **mode_sel**: Block to select the filter mode for the basic color processing block (color_processing).
- **Visualization**: Blocks related to visualization.
  - · **vga_display**: Block to display the image on a VGA screen.

In Figure 4.14 a snap from a YouTube video[14] demonstration of this POC done by collaborators from the JdeRobot organization can be seen. The FPGA Modules are configured to filter and detect red objects and a red ball is presented before the robot's camera. The robot then moves to keep track of the detected object.

This POC demonstrates how reconfigurable programming can be used to create complex image processing algorithms and sensor/actuator control for robotic applications. This way the application takes advantage of the improved performance and flexibility that FPGAs provide.

[14] `https://www.youtube.com/watch?v=rbdQ36ZJ7Lo`

Chapter 5

Simulated FPGA

In the previous chapter the different steps involved in the development and testing of physical FPGA blocks and applications were described. This somewhat cumbersome process is time-consuming and resource-expensive. In this chapter an argument for the simulation of some of the procedures involved in this process is presented as a more efficient alternative.

5.1 SIM FPGA-Robotics

A simulated FPGA toolchain has been created in the FPGA-Robotics project[1] to provide a faster and more versatile alternative framework for the development of FPGA based robotic applications.

5.1.1 Motivation

Developing robotic computer vision algorithms for FPGAs is a rather complex endeavor, it not only demands a diverse set of knowledge and tools but also can require pretty large budgets. Having to acquire all the necessary hardware and building an actual robot to test applications is expensive and time-consuming. Furthermore, hardware development carries inherent difficulties that one may want to avoid. Many of the setbacks encountered during the research and development of the previous chapter were examples of this; Hardware malfunctions, noise polluting the signals travelling between the different hardware components causing bugs and

[1] `https://github.com/JdeRobot/FPGA-robotics/tree/master/sim_fpga`

miss-configurations, and possibly most important, the debug and test loop for developing the different blocks and applications was extremely lengthy.

FPGA Simulation The synthesis and upload processes of physical FPGAs can take a lot of time. For the largest FPGAs and most complex modules, debugging the hardest systematic errors like synthesis mismatchs, can take days through the method of prototyping on the actual board. Furthermore, the latest synthesis tools employ advanced optimizations that can lead to errors. Even for relatively simple applications and FPGA boards like the ones presented in this document this process can take several minutes. Compare this to the almost instant time it would take to compile regular software versions of the same algorithms.

FPGA simulation allows for the optimization of the lengthy test and debug loop present in hardware development, and eliminates the need to purchase hardware beyond the ubiquitous PCs where the simulations can take place.

Another important difference to be noted between hardware and software development are debugging tools. When developing hardware applications programmers may often find themselves stuck when things don't work as the usual tools used for debugging are missing or require additional software and hardware.

Test Environment Simulation Test benches are the most common way to directly test the logic inside the RTL designs. A test bench allows the system to bypass the need to test with additional hardware, this is done by simulating the different input signals and handling the output signals that the FPGA modules use. Afterwards, the recorded chronograms of the control signals must be checked to verify the functional correctness of the hardware model.

This is a necessary step when creating hardware designs and quite convenient during the first steps of development, but as prototypes grow more complicated and involve more complex inputs and outputs the design of the test benches becomes a very complicated task. This is usually the case for robotic applications, specially the ones that rely on computer vision and image processing.

In this chapter a series of simulation based software applications and plugins/blocks are designed and developed to provide a versatile alternative to hardware testing for FPGA robotics applications. With the help of open source software like OpenCV, ROS and Gazebo a more

powerful, controlled and configurable framework for testing has been created.

Compared to Hardware testing this environment provides simulations that can be performed with exactly the same deterministic results every time. This contrasts with real world applications where, for example, visual inputs are never exactly the same and noise can interfere with the signals traveling between the different hardware components. This framework can easily produce very complex inputs and outputs that can be seamlessly generated and fed to the, in this case, simulated FPGA.

5.1.2 Simulating FPGAs with Verilator

FPGA manufacturing companies offer proprietary simulation software at a price but there is a growing set of open source alternatives. The developments presented on this chapter make use of the Verilator tool to replace the physical FPGA with cycle-accurate behavioral models of the Verilog modules in C++ or SystemC. With the use of Verilator we can address many of the aforementioned issues but it also comes with some drawbacks.

Verilator is a cycle-based simulator, which means it does not evaluate time within a single clock cycle, and does not simulate exact circuit timing. Instead, the circuit state is typically evaluated once per clock-cycle. This means it cannot be used for timing simulations, asynchronous (clockless) logic, timed signal delays, or in general any signal changes that involve the concept of time. On the other hand, because everything between clock edges is ignored, Verilator's simulations run extremely fast and work great for simulating the functionality of synchronous digital logic circuits.

Verilated models expose the inner hardware logic on software, this means that a more traditional approach to debugging can be used. Tools like gdb (GNU Debugger) or simple `fprint` tracing can help tremendously when developing Verilog RTL files. Verilated modules can also produce VCD output files containing the series of time-ordered value changes for the signals in a given simulation model. With the help of waveform viewer software like GTKWave[2] these signals can be analyzed to help with debugging.

Structure of the Verilated Model For a given class "prefix" Verilator outputs a `prefix.h` header file which defines a class named `prefix` which represents the generated model to be

[2] `http://gtkwave.sourceforge.net/`

instantiated. It will additionally create a `prefix.cpp` file, together with additional header and implementation files for internals. This model class defines the interface of the Verilated model. The output of the process will also contain a `prefix.mk` file that may be used with "Make" to build a static library with all required objects in it.

The generated model class file manages all internal state required by the model, and exposes the following interface to allow interaction:

· Top level IO ports are exposed as references to the appropriate internal equivalents.

· Public top level module instances are exposed as pointers to allow access to inner items.

· The root of the design hierarchy is exposed via the `rootp` member pointer to allow access to model internals.

In C++ output mode, like it is used for this project, a C++ wrapper and main loop for the simulation must be written by the end user, this file instantiates the model class and links with the Verilated model.

5.1.3 Design

Verilator simulates the FPGA Verilog code and several software extensions may be developed to connect that Verilog code to real or simulated robot sensors and actuators. These proposed software drivers make the sensory readings available to the Verilog modules and send the outputs to the appropriate actuator to extend the usefulness of the simulated FPGA-Robotics toolchain.

Figure 5.1 displays a block diagram of the design concept this simulated toolchain is trying to achieve.

Connection to sensors and actuators: drivers and blocks As was the case with Physical FPGA-Robotics, the SIM FPGA-Robotics[3] repository already presented some developments that implement a functional POC of Verilated modules and a GUI.

This proof of concept on how to build hardware simulators using Verilator was developed by David Lobato (JdeRobot member) and can be accessed here[4]. The application features "InputDriver" and "OutputMonitor" classes that, as their name suggests, handle the mentioned inputs

[3] `https://github.com/JdeRobot/FPGA-robotics/tree/master/sim_fpga`

[4] `https://github.com/JdeRobot/FPGA-robotics/tree/master/sim_fpga/examples/example1`

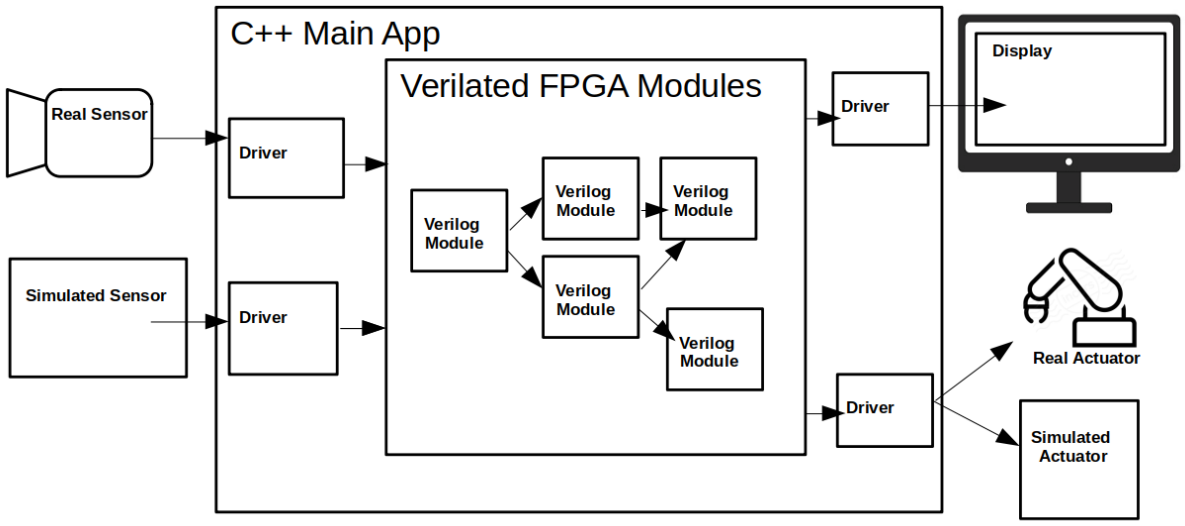

Figure 5.1: Design Concept Diagram

and outputs to and from the Verilated Model.

It includes a Verilog RTL design with complex inputs and outputs that implements a pixel processor and color filter. A diagram of the Verilog modules can be seen in Figure 5.2 and a brief description of the most important input and outputs follows:

· Input and output images are stored in 80x60 12-bit memories

· A 3-bit color filtering mask input signal

· An 8-bit output ”Centroid” signal intended for LED displays that indicates the detected object’s position on the horizontal plane

The POC also comes with a Graphic User Interface (GUI), pictured in Figure 5.3, where different input signals can be selected and the output signals are displayed. This GUI also allows to dynamically select the number of FPGA simulation cycles to be run per iteration of the main software loop. The latest versions of this POC were feeding the Verilated model with static images to test the functionality of the image processor and color filter.

Using this previous POC example as a stepping stone to simulated FPGA development along this project a series of plugins or blocks has been developed with the intend of expanding the functionality in a similar fashion to the one described in the Physical FPGA chapter.

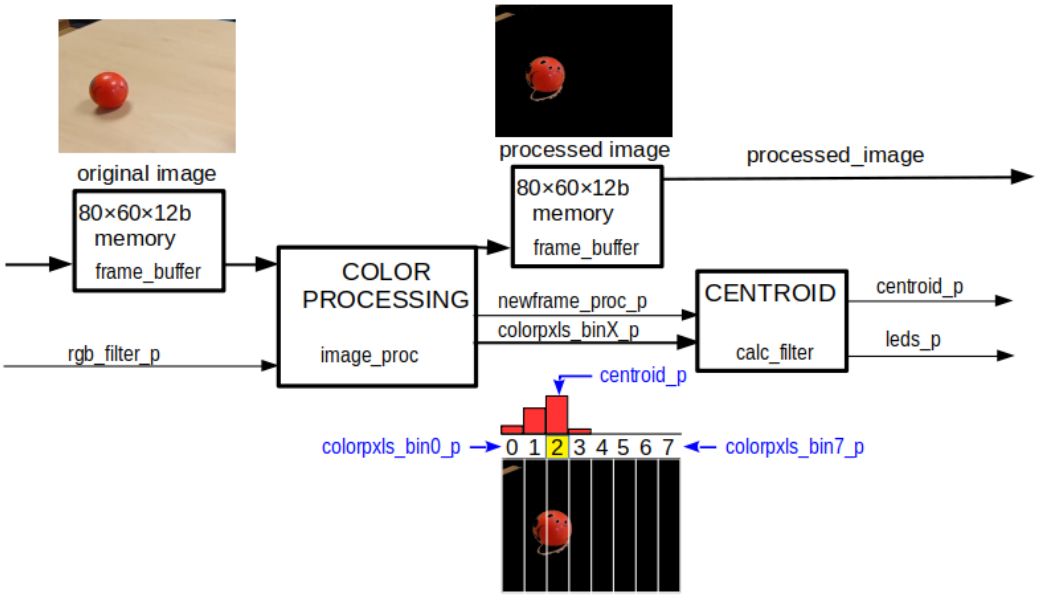

Figure 5.2: Simplified Verilog Module Diagram

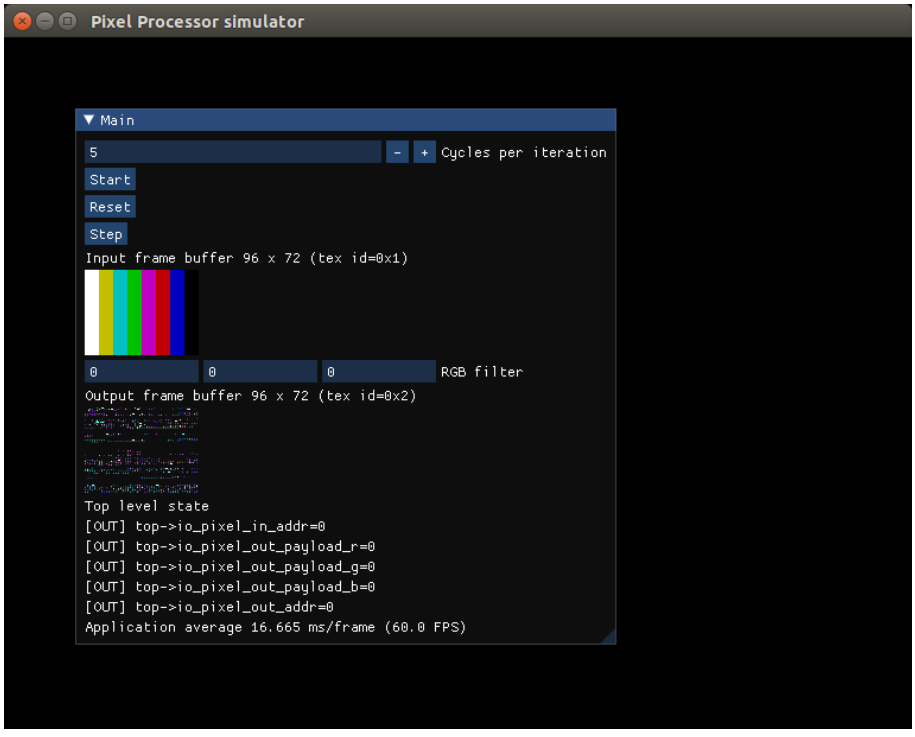

Figure 5.3: Pixel Processor simulator GUI

5.2 Simulated Blocks

This section presents the three developed software blocks for robotics and computer vision applications.

5.2.1 Webcam Block

This first block intends to replace the static images that Verilated Model was being fed with real video footage taken from a webcam.

The target is to test the same RTL designs used by the POC presented in the previous section (Image processing and Color filtering Verilog modules). This new block incorporates live Webcam feed to the system in place of the static images that were being used up until that point. As discussed in the following sections, creating and incorporating simulated inputs and environments is a complex task so being able to use this Webcam Block as a versatile testing platform for the RTL designs is extremely useful. Code can be found in the FPGA-Robotics's SIM_FPGA repository[5]

OpenCV is the de facto standard library for computer vision mainly aimed at real-time operation. It has been used for implementing all the logic necessary to incorporate a webcam. Integration was facilitated by the fact that OpenCV's primary interface is written in C++.

Figure 5.4 shows a simplified diagram of the different parts that the C++ application contains, including the Webcam Block

The following code snippets portray some OpenCV functions to receive the image data from the default webcam installed in the system and transforming that data so the Verilated modules can process them.

```
//instantiate and open Webcam
  cv::Mat input_feed;
  cv::Mat resized_input_feed;
  cv::VideoCapture cap(0);

  if (!cap.isOpened()) {
    std::cout << "cannot open camera";
```

[5] https://github.com/JdeRobot/FPGA-robotics/tree/master/sim_fpga/examples/poc/example4

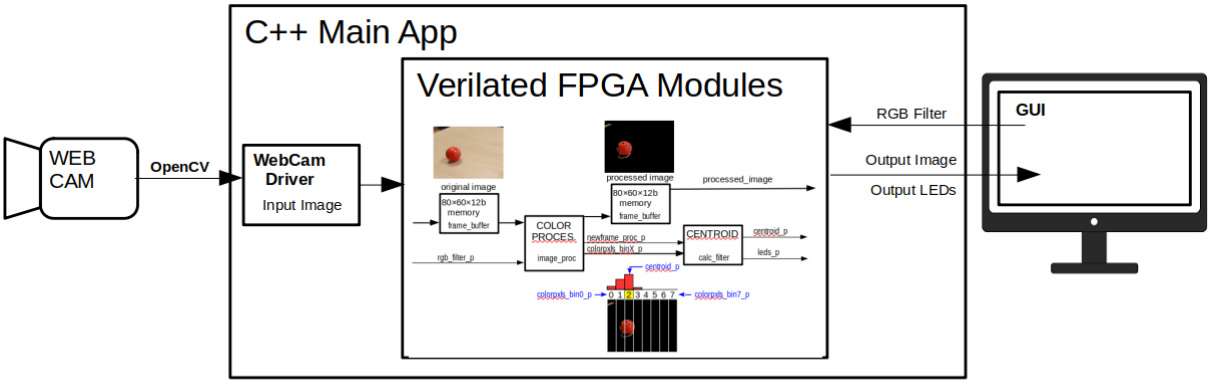

Figure 5.4: Webcam block diagram

```
}
```

```
//Inside the main loop
//Feed and resize image data
cap >> input_feed;
cv::resize(input_feed,resized_input_feed,cv::Size(cols,rows),cv::INTER_LINEAR);
```

Figure 5.5 shows the demo application displaying the processed live webcam images. Figure 5.6 shows the output with a RED Filter mask is applied by the Verilated modules. Notice that in Figure 5.6 pink and even gray pixels are passing the filter. The thresholds for the color filters inside the Verilog Module are too low for accurately tracking an object like the red one in my hand.

This is a prime example of the utility of this application; the inner system can be easily debugged dynamically and the correct values for the filters can be found for different lighting conditions. Finding out the correct values of these parameters with a real FPGA and would take many tries, each one taking a very long time due to the slow synthesis and uploading processes. Another alternative, creating a test bench capable of feeding the system with similar inputs would take a tremendous amount of time.

A video demonstration of this example can be found here[6].

[6] https://www.youtube.com/watch?v=Q-7BDBg4F00

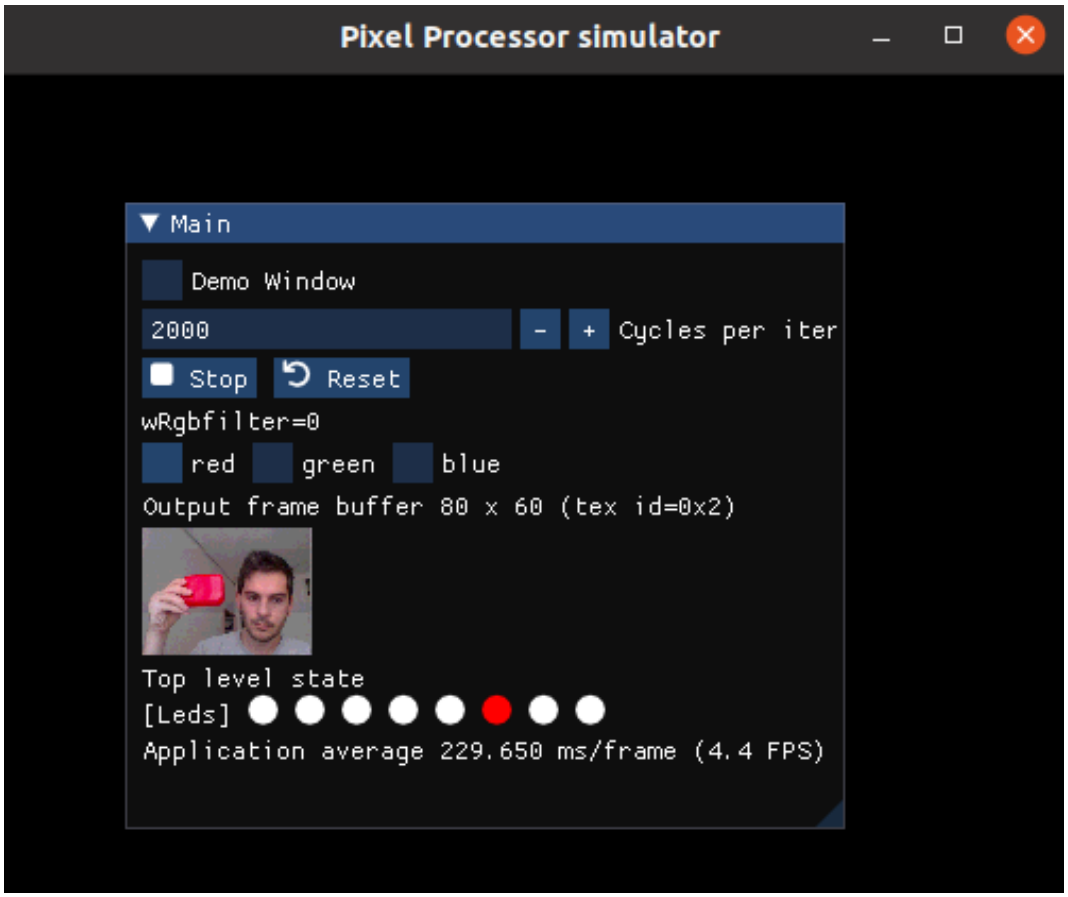

Figure 5.5: GUI Displaying Webcam feed and LED output

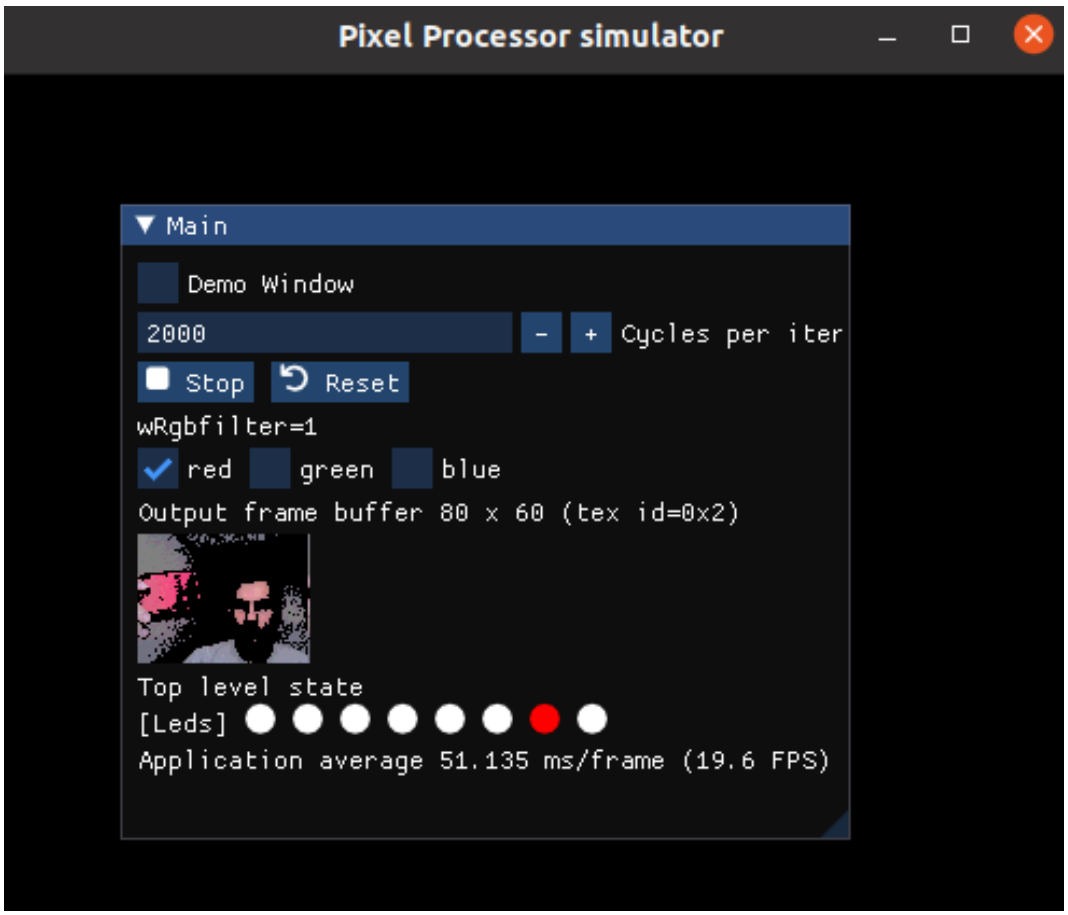

Figure 5.6: Filtered Webcam Feed

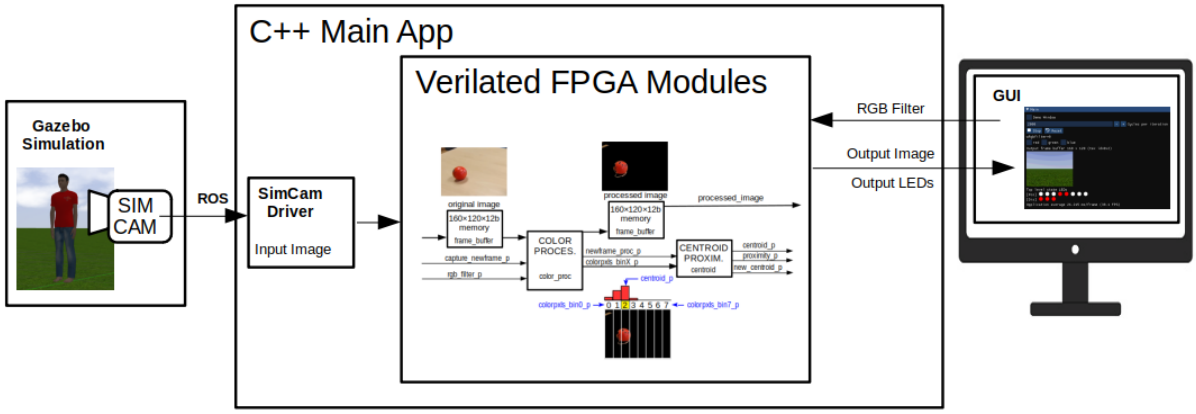

Figure 5.7: SimCamera block diagram

5.2.2 SimCamera Block

The SimCamera block, located in this github repository[7], achieves a completely simulated input for the Verilated model. This is accomplished with the help of ROS and Gazebo. Figure 5.7 represents a simplified diagram of the designed system.

Gazebo is a open source 3D robotics simulator described in section 3.2.3. For this Block it was used to create a simulated world where a simulated camera records the footage to be used by the Verilated Modules. This application is launched independently from the SimCamera but it is needed to publish the information the block is expecting.

The SimCamera block uses ROS, a middleware for robotic applications described in section 3.2.3, to handle all the communication between the C++ application that instantiates the Verilated model and the simulated world where the image data comes from. This is done through ”ROS Topics”, which are communication buses over which different nodes or applications can exchange messages.

Gazebo supports SDF files to describe the simulation to be loaded. An SDF file defines the world, the robot's characteristics and what plugins to load. A `camera.world` SDF file[8] was developed to simulate the robot (consisting of just a camera), the world and also load a camera plugin that publishes the recorded images to the ROS Topic ”/image_raw”.

[7] `https://github.com/JdeRobot/FPGA-robotics/tree/master/sim_fpga/examples/poc/example5`

[8] `https://github.com/JdeRobot/FPGA-robotics/tree/master/sim_fpga/examples/poc/example5/worlds`

On the other side, the SimCamera block is subscribing to that same Topic and instantiating a callback function to handle new image data received from the simulated world. This data is then fed to the Verilated model for image processing. The following code snippets show part of the code developed for this block, specifically how the subscription to the ROS topic is made and the mentioned callback function.

```
//ROS Integration
ros::init(argc, argv, "image_listener");
ros::NodeHandle nh;

image_transport::ImageTransport it(nh);
image_transport::Subscriber sub = it.subscribe("image_raw", 1,
    imageCallback);
```

```
void imageCallback(const sensor_msgs::ImageConstPtr& msg)
{
  try
  {
    input_feed = cv_bridge::toCvCopy(msg, "bgr8")->image;
    cv::waitKey(30);
  }
  catch (cv_bridge::Exception& e)
  {
    ROS_ERROR("Could not convert from '%s' to 'bgr8'.",
        msg->encoding.c_str());
  }
}
```

Figure 5.8 shows a screenshot from a video demonstration[9]. On the left side of the image, a Gazebo simulation can be seen where several "stop signs" are placed in front of the simulated camera. On the right side, the GUI is displaying the output images processed by the Verilated model.

[9] https://www.youtube.com/watch?v=m2YIrA4VZRk

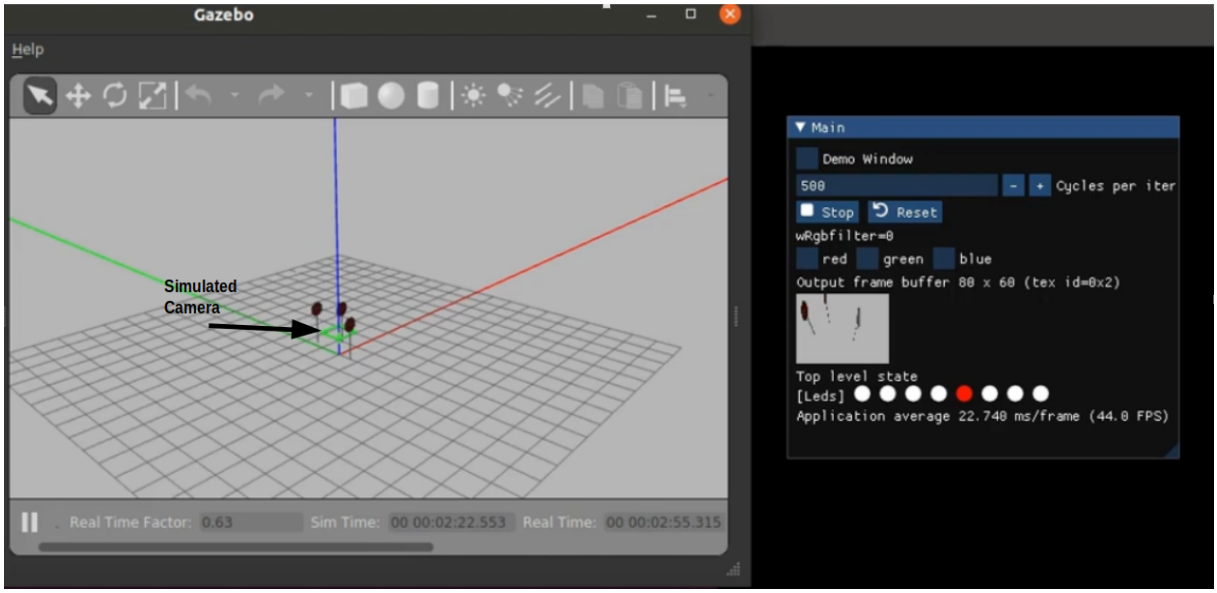

Figure 5.8: Simulation of Video Recording and FPGA image processing

5.2.3 SimMotors Block

Figure 5.9 shows a simplified block diagram of the developments achieved in this section. The SimMotors Block completes the simulation by including a differential drive robot into the simulated world and implementing the logic needed to integrate the necessary inputs and outputs between the Verilated Model, the main application and the simulated world.

Up until this point the RTL designs being used by the simulated FPGA only provided image processing and color filtering. This new block[10] integrates the updated processing modules used for the physical GoPiGo POC described in section 4.3.2 [11] from the Physical FPGA. These Verilog modules include new functionality and modified inputs and outputs.

Simulated World developments A differential drive robot was designed in Gazebo to replace the physical GoPiGo3 in the simulation. The SDF file found here[12] includes the new robot model pictured in Figure 5.10, an updated world and new plugins to communicate with the

[10] https://github.com/JdeRobot/FPGA-robotics/tree/master/sim_fpga/examples/poc/example6

[11] FPGA-robotics/Projects/ComputerVision/ulx3s/apio/ov7670x3_vga160x120_spipan/

[12] https://github.com/RoboticsLabURJC/2017-tfg-richard-nicklas/tree/main/ROS/diffdrive_cam_bot

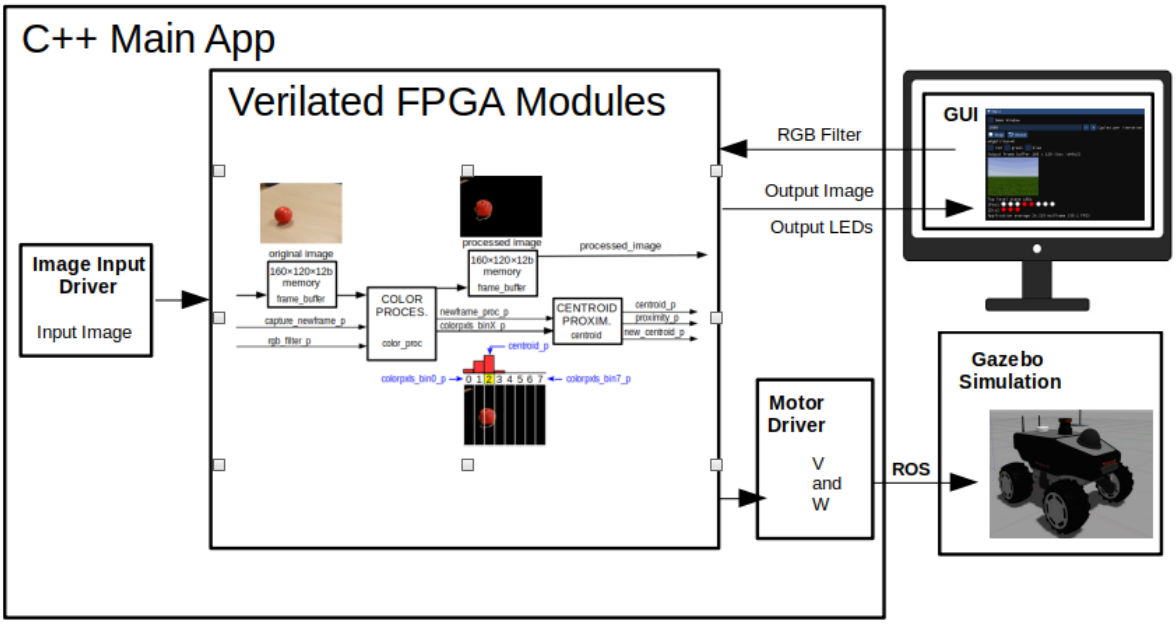

Figure 5.9: SimMotor block diagram

ROS Topics that will be receiving commands for the robot's locomotion.

Main Application Developments Developments in the Main application and GUI include instantiating the updated interface from the Verilated model. Changes in the RTL design are summarized below and can be seen in Figure 5.11

· Doubling the image size, from 80x60 pixels to 160x120.

· **Capture_newframe input**: A 1-bit signal to be activated every complete new image frame to notify the Verilated model.

· **Proximity output**: A new 3-bit output identifying the distance to the detected object centroid is calculated by counting the pixels that have passed the selected filter. A value of "0" means no object detected and values from "1" to "7" represent increasing proximity to the object so "1" means "very far away" and "7", "very close".

With the new RTL design integrated into the system, linear and angular speeds (V and W respectively) can be calculated to allow object tracking. In this case the V and W values are estimated with the help of the new 3-bit Proximity signal and the already implemented 8-bit Centroid signal that the Verilated Module is outputting from its image processing and object detection mechanisms. If no object is detected the Robot will spin in search of new targets.

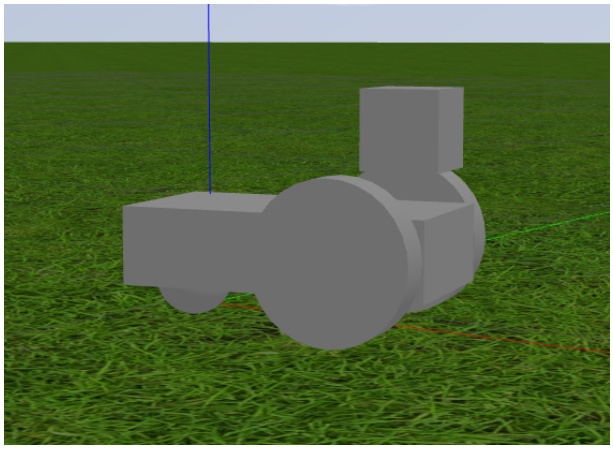

Figure 5.10: Camera equipped Differential Drive Robot in Gazebo

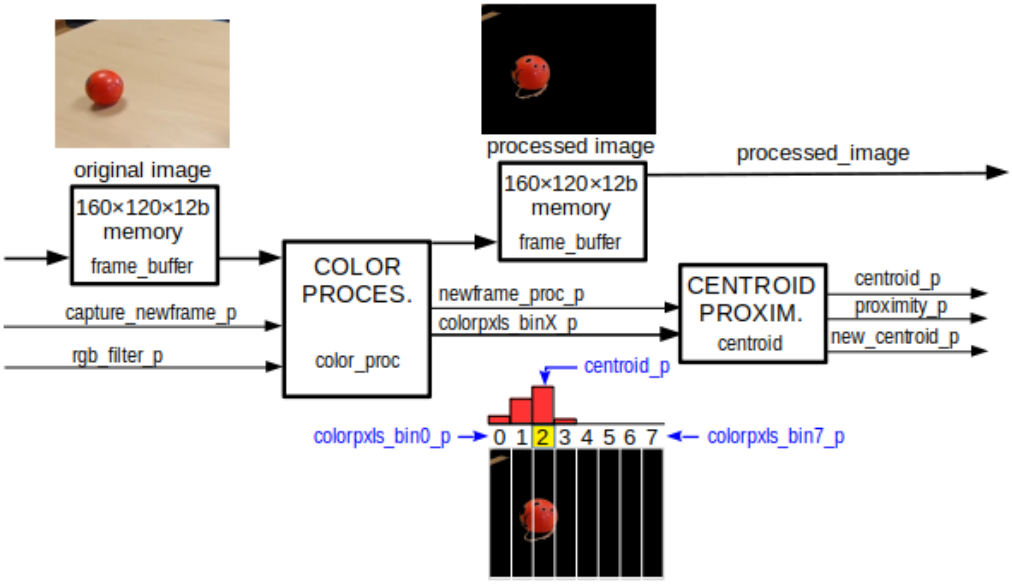

Figure 5.11: Verilog Module Diagram

New communication mechanisms were developed in the C++ SimMotor Block to transmit movement commands to the simulated robot. This is achieved with the help of the ROS Topic `/cmd_vel`. This topic handles `geometry_msgs/twist`[13] messages consisting of two 3 dimensional vectors representing linear and angular velocities (V and W respectively).

The GUI was also updated to display the value of the new Proximity output. The final version can be seen in Figure 5.12, which displays the robot's camera view of the simulated world. As no filter is selected every pixel is detected as a single object. The three "[Dis]" LEDs display the proximity signal, in this case the object is covering the whole frame so it has a value of "7" or "very close". The right "POS" LEDs display the centroid signal with the position of the detected object in the horizontal plane relative to the cameras frame, in this case the object is considered centered.

[13] `http://docs.ros.org/en/noetic/api/geometry_msgs/html/msg/Twist.html`

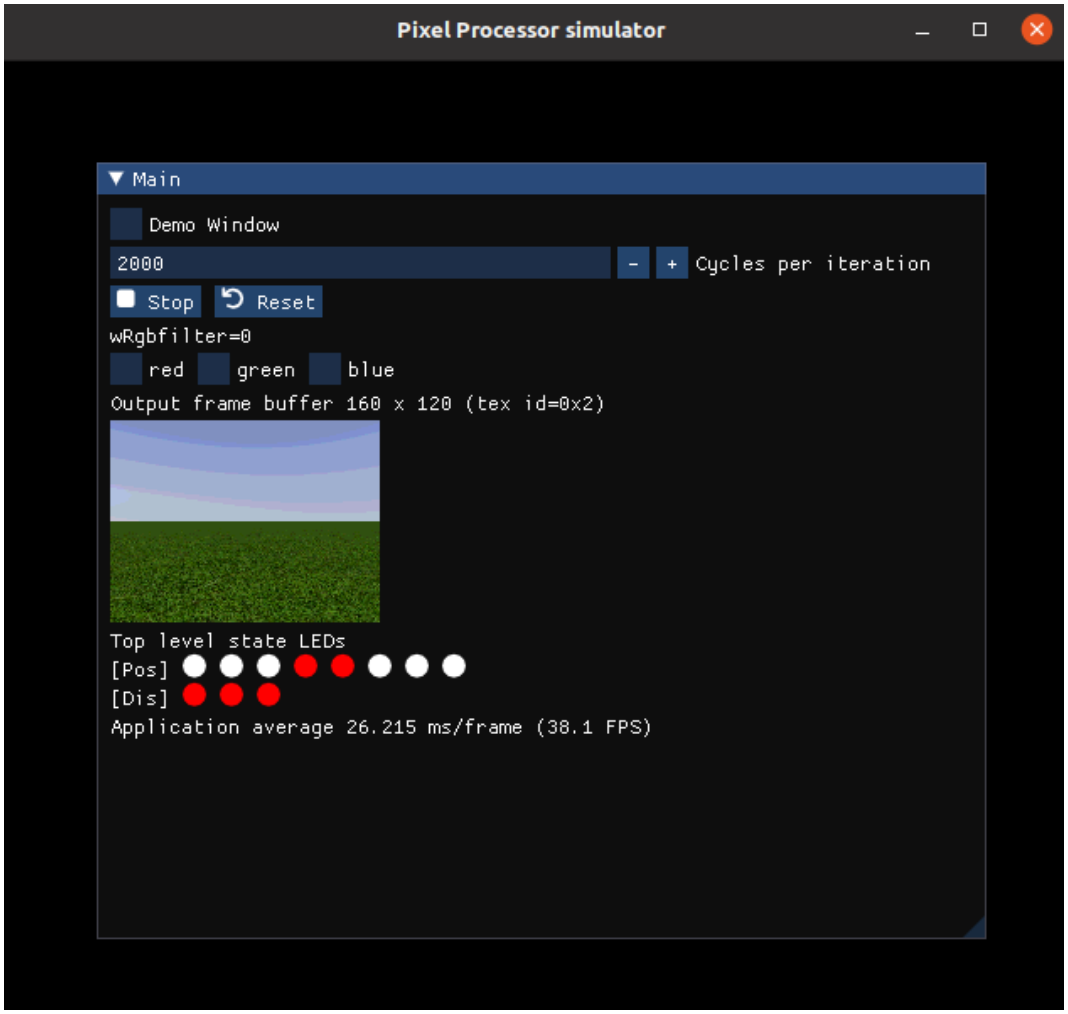

Figure 5.12: Final GUI

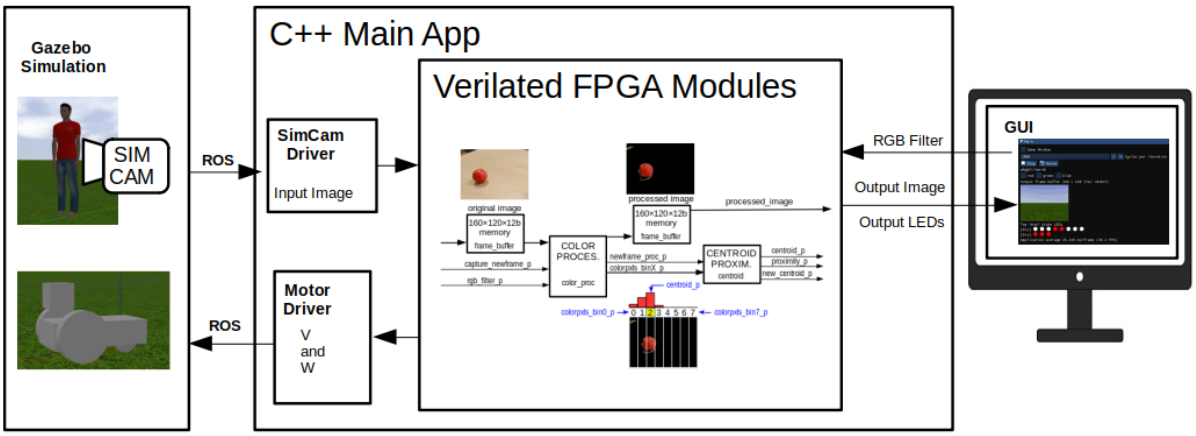

Figure 5.13: POC Block diagram

5.3 Application

A POC was developed to display a working integrated example of the different blocks presented in this chapter. Code for this POC can be found here[14].

5.3.1 Simulated Tracking Robot POC

Figure 5.13 shows a simplified diagram with the main blocks included in this robotics application POC. The main application, written in C++, expands the discussed software that served as a testing platform for the previous sections in this chapter. It includes an instance of the Verilated Modules, a GUI to display the inputs and outputs, the SimCamera Block, the SimMotors Block, and all the necessary logic to communicate with the Gazebo simulated world.

The simulated world has been expanded to include a moving human model, pictured in Figure 5.14. This person is wearing a distinctive red shirt so that the image processing logic running on the Verilated model inside the main application can easily detect the person from the simulated camera. Thanks to the Proximity and Centroid output signals the SimMotor Block can produce linear and angular velocity to direct the robot towards the detected object. The values are selected to track the object and reach a proximity value of "1". This information is then published via the `/cmd_vel` ROS Topic back to the Gazebo simulated world where the

[14] `https://github.com/RoboticsLabURJC/2017-tfg-richard-nicklas/tree/main/verilator/poc_verilator_simulation/sim/example8`

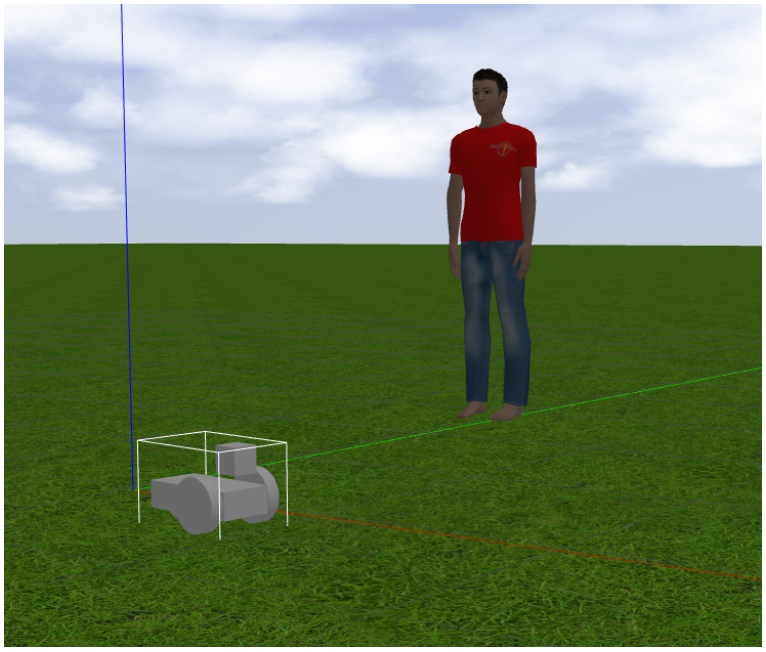

Figure 5.14: Human model in the Gazebo simulation

robot begins to move.

Figure 5.15 and 5.16 are screenshots from an execution of this POC. In the images the two discussed applications can be seen working. On the right side is the Gazebo world simulation with the simulated robot and human. On the left side is the GUI for the Main app, notice that the red filter is selected. In the processed output display the man's red shirt can be seen as detected, its position and distance are displayed with the GUI LEDs. Notice that in the second image the robot moved towards the detected object.

A video example of this POC has been uploaded to YouTube[15]. In this video we can see the robot successfully tracking and following the person that is moving in the simulated world.

This POC manages to reproduce the results of the physical FPGA POC detailed in section 4.3.2. It demonstrates how the complete development and execution environments needed by FPGA based hardware applications can be simulated. It is not difficult to see the versatility and benefits these kind of software developments can bring to this space; Easier debugging, less resources needed, faster developing times and flexible testing environments.

[15] `https://www.youtube.com/watch?v=VUQ8be0lpvI`

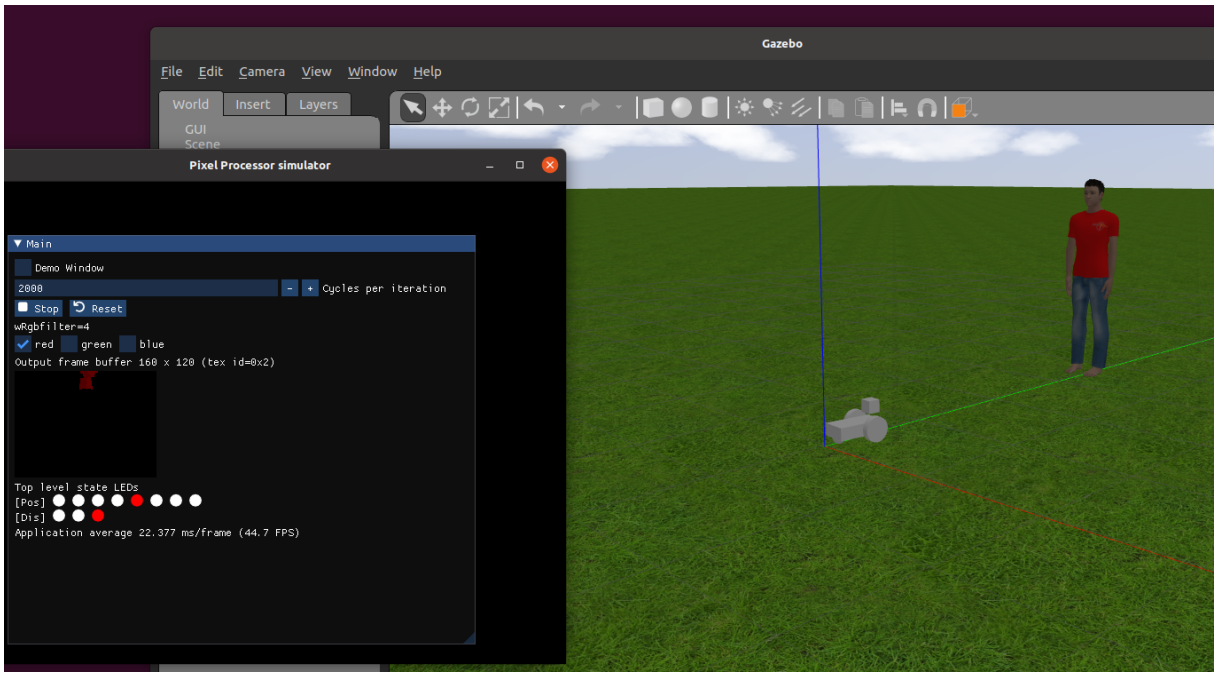

Figure 5.15: Simulation of a Robot tracking a person

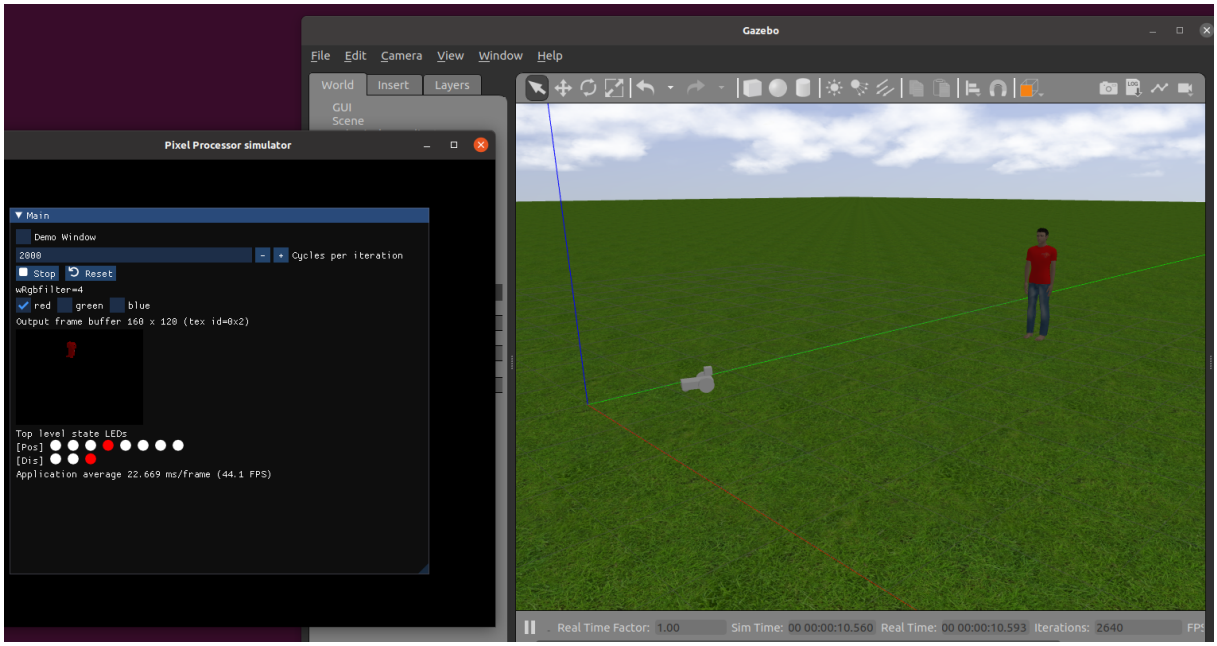

Figure 5.16: Simulation of a Robot tracking a person

Chapter 6

Conclusion

This chapter concludes the presented project. Throughout this paper the development of several FPGA based modules and POCs for robots has been portrayed. These applications, tests and documentation have contributed and will continue to help with the work that is being done in the international open source Reconfigurable Computing community.

The main goal of producing a series of software blocks for physical and simulated FPGA based development tool chains has been achieved. This has been done through extensive research and by acquiring a deep understanding of the covered topics. The utility and correct behaviour of these pieces of software has been proved though a series of POC applications detailed in sections 4.3.1, 4.3.2 and 5.3.1 of this document.

The field of robotics is particularly concerned with the optimization of its hardware and software since autonomy is one of the features that restrict functionality the most. Due to the conclusions reached through the developments presented in this document, it can be said that transitioning part of the development process from physical hardware to simulated environments can provide great advantages like reduced costs and, faster and more versatile designing, prototyping and testing processes. These benefits are already evident from the POCs discussed. Being able to debug FPGA applications that use live image data without having to create realistic test benches significantly accelerates development. Not needing to synthesize and upload the modules to the actual board greatly helps with the application testing process, facilitating the verification that everything works correctly.

Tackling the proposed objectives from two very different perspectives, one physical and hardware-oriented and the other, simulated and software-oriented, has allowed me to expe-

rience and learn from an incredibly diverse pool of knowledge. These developments covered very different topics and fields of research including but not limited to Computer Vision, Hardware Description Languages, Programmable logic devices like FPGAs, code repositories, robotics, diverse simulation software, electronics and embedded-systems, signal sampling, micro-controller communication protocols, middleware software and object detection algorithms.

Besides all the acquired and applied technical skills I also improved many soft skills like team-working, keeping a positive attitude and even improving my work ethic.

This project relied on the support of open source tools, projects and organizations but also contributed to them in the same way. This was one of the objectives that motivated me the most as I personally believe the free and open source movement is very important for the progress of science and technology, and I'm satisfied to be a part of it.

6.1 Future Developments

The work presented in this document opens new research paths and establishes the foundations for the development of more complex FPGA Modules for Robots. The different blocks developed and POCs tested contribute to a living open source ecosystem where hardware and software are being constantly improved. During the research of this project many areas for improvement and research became evident, the most relevant are presented below.

Hardware-in-the-loop In this work two different approaches to FPGA based development have been studied, one based on physical hardware and one completely based on simulated systems. Hardware-in-the-loop (HIL) simulation proposes a technique that is halfway between them. It is used in the development and testing of complex real-time embedded systems by incorporating real controllers into the simulation development loop. In HIL simulation all the inputs and outputs to the embedded-system need to be simulated but the actual processing is done on real hardware.

HIL simulation has been recently applied to the automatic generation of complex controllers for robots. Algorithms such as Back-to-Reality [12] (BTR) and Estimation Exploration [2] have

been proposed in this context as ways for the robot to infer a self-model and feed sensor and actuator data via simulation to embedded-systems.

The developments presented in Chapter 5 could be considered a form of HIL, but with simulated Hardware (via Verilator). HIL simulation with real Hardware has its own benefits and disadvantages compared to the ones presented in this project and I believe it is worth researching how other aspects of HIL might be a more fitting for robotic applications.

Algorithm optimization for Simulated blocks The algorithms used in the Simulated FPGA application POC to control the robot's linear and angular momentum are enough to demonstrate the functionalities involved in the POC but still very basic and rough. A PID (proportional–integral–derivative) control loop mechanism could be used to command more reactive and smoother movements to the simulated robot.

The framework developed in this project could facilitate the adjustment of these PID parameters in the ROS-Gazebo simulation instead of having to do physical testing.

Statistical analysis Even though the benefits that FPGA solutions and simulated environments provide to Robotics applications are already evident from this work and many others, I believe it would be very interesting to perform deep statistical performance comparisons between the same robotic applications deployed on FPGAs and on ordinary micro-processors. Some research has already been done in this field for simulated software; the "*Proposal of ROS-compliant FPGA Component for Low-Power Robotic Systems*" study [10]. On that paper ROS compliant FPGA component technology for an easy integration of FPGAs into robots is proposed. Its research found that the developed ROS-Compliant FPGA component performs 1.7 times faster compared to the ordinary ROS software component.

Another interesting area of research that has arised as a result of this project's developments is related to the aforementioned benefits that FPGA simulation development brings over traditional hardware oriented development. Very interesting statistical analysis can emerge from studying the improvements in development times and cost reduction that simulated FPGAs provide to the robotics and computer vision fields.

Bibliography

[1] E. Alcaín, P. R. Fernández, R. Nieto, A. S. Montemayor, J. Vilas, A. Galiana-Bordera, P. M. Martinez-Girones, C. Prieto-de-la Lastra, B. Rodriguez-Vila, M. Bonet, C. Rodriguez-Sanchez, I. Yahyaoui, N. Malpica, S. Borromeo, F. Machado, and A. Torrado-Carvajal. Hardware architectures for real-time medical imaging. *Electronics*, 10(24), 2021.

[2] J. Bongard and H. Lipson. Once more unto the breach: Automated tuning of robot simulation using an inverse evolutionary algori. *Artificial Life - ALIFE*, 01 2004.

[3] J. M. Cañas, J. Fernández-Conde, J. Vega, and J. Ordóñez. Reconfigurable computing for reactive robotics using open-source fpgas. *Electronics*, 11(1), 2022.

[4] T. Magin. Fpga history, Aug. 2004. `https://web.archive.org/web/20070412183416/http://filebox.vt.edu/users/tmagin/history.htm`.

[5] J. Ordóñez Cerezo, E. Castillo Morales, and J. M. Cañas Plaza. Control system in open-source fpga for a self-balancing robot. *Electronics*, 8(2), 2019.

[6] W. Roelandts. Xilinx xcell issue 32, June 1999. `https://www.xilinx.com/publications/archives/xcell/Xcell32.pdf`.

[7] B. W.Boehm. A spiral model of software development and enhancement, May 1988. `http://www-scf.usc.edu/~csci201/lectures/Lecture11/boehm1988.pdf`.

[8] C. Wolf. Project icestorm - lattice ice40 fpgas bitstream documentaion (reverse engineered), Apr. 2015. `https://github.com/YosysHQ/icestorm`.

[9] www.ArduCAM.com. Cmos ov7670 camera module datasheet, May 2015. https://www.openhacks.com/uploadsproductos/ov7670_cmos_camera_module_revc_ds.pdf.

[10] K. Yamashina, T. Ohkawa, K. Ootsu, and T. Yokota. Proposal of ros-compliant FPGA component for low-power robotic systems. *CoRR*, abs/1508.07123, 2015.

[11] R. H. C. M. Yiu-Fai Chan, Robert Frankovich and D. Wong. Computer history museum, altera ep300 design and development oral history panel, Aug. 2009. https://archive.computerhistory.org/resources/access/text/2012/10/102702147-05-01-acc.pdf.

[12] J. C. Zagal, J. R. del Solar, and P. Vallejos. Back to reality: Crossing the reality gap in evolutionary robotics. *IFAC Proceedings Volumes*, 37(8):834–839, 2004. IFAC/EURON Symposium on Intelligent Autonomous Vehicles, Lisbon, Portugal, 5-7 July 2004.