Tiago Miguel Saraiva Carrasqueira

Licenciado em Ciências da Engenharia Eletrotécnica e de Computadores

FPGA in image processing supported by IOPT-Flow

Dissertação para obtenção do Grau de Mestre em

Engenharia Eletrotécnica e de Computadores

Orientador: Doutor Filipe de Carvalho Moutinho, Professor Auxiliar, Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa

Co-orientador: Doutor Rogério Alexandre Botelho Campos Rebelo, Investigador, Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa

Júri:

Presidente: Doutor João Carlos da Palma Goes

Arguente: Doutor Luís Filipe dos Santos Gomes

Vogal: Doutor Filipe de Carvalho Moutinho

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Março de 2019

FPGA in image processing supported by IOPT-Flow

Copyright © Tiago Miguel Saraiva Carrasqueira, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa.

A Faculdade de Ciências e Tecnologia e a Universidade Nova de Lisboa têm o direito, perpétuo e sem limites geográficos, de arquivar e publicar esta dissertação através de exemplares impressos reproduzidos em papel ou de forma digital, ou por qualquer outro meio conhecido ou que venha a ser inventado, e de a divulgar através de repositórios científicos e de admitir a sua cópia e distribuição com objetivos educacionais ou de investigação, não comerciais, desde que seja dado crédito ao autor e editor.

Acknowledgments

To the “Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa” (FCT-UNL), for these 5 years of learning, experiences and an amazing course. I have learned so much and I could not be prouder of this course and all the people involved in it.

To my advisor and co-advisor, Professor Filipe de Carvalho Moutinho and Doctor Rogério Alexandre Botelho Campos Rebelo, for all the support, availability and hundreds of hours spent in the development of this project. I cannot thank you guys enough, I have learned so much from both of you. You have become an inspiration and a role model to me, and that is something I will take from this project to the rest of my life. I will always learn something from you, and to work with you was an honour. To João Pedro Matos Carvalho, for implementing the GLCM algorithm is software and feedback about execution times. It provided the opportunity to work alongside him me and to write a paper about our work. Thank you.

To my parents and brother, my best friends and my inspiration, I thank you for all the support, advises, love and understanding you’ve given me so far. Although the three of you are very far away right now, I feel like you are with me doing this project and without that presence I wouldn’t be able to finish it.

Thank you for all the understanding, all the trips to see me in the weekends when I could not go home to save time and use it in this project. To my mother, for every call, for every concern, for every cheer up speech, I cannot thank you enough. Thank you for being my smile, my strength, and to cheer me up every time. To my father, for being tough with me and always give me your advice to work harder, to become a better person and to give my all in this project. You have been my strength, and my determination. To my brother, for all the fun moments and laughs, you have been critical to help me to face any challenge with a smile and seeing the good side of everything.

To my grandmas and grandpa, for all the calls given me support and all the lunches you made so I would not waste time in making them and focus in this project. To all my friends, for all the understanding and fun moments over the years. I missed our events, our meetings, many times. And you guys kept supporting and meeting me. Thank you for all the strength you have given me.

To Margarida, for all the support, for all the love and understanding over this long journey. Your presence was critical to develop this project, in the good and the bad moments. Without you I would not make it.

Abstract

Image processing is widely used in the most diverse industries. One of the tools widely used to perform image processing is the OpenCV library. Although the implementation of image processing algorithms can be made in software, it is also possible to implement image processing algorithms in hardware. In some cases, the execution time can be smaller than the execution time achieved in software.

This work main goal is to evaluate the use of VHDL, DS-Pnets, and IOPT-Flow to develop image processing systems in hardware, in FPGA-based platforms. To enable it, a validation platform was developed. A set of image processing algorithms were specified, during this work, in VHDL and/or in DS-Pnets. These were validated using the IOPT-Flow validation tool and/or the Xilinx ISE Simulator. The automatic VHDL code generator from IOPT-Flow framework was used to translate DS-Pnet models into the implementation code. The FPGA-based implementations were compared with software implementations, supported by the OpenCV library. The created DS-Pnet models were added into a folder of the IOPT-Flow editor, to create an image processing library.

It was possible to conclude that the DS-Pnets and their associated tools, IOPT-Flow tools, support the development of image processing systems. These tools, which simplify the development of image processing systems, are available online at http://gres.uninova.pt/iopt-flow/.

Keywords: Field Programmable Gate Array (FPGA), VHDL, Input Output Place Transition Flow (IOPT-Flow), Image Processing, DS-Pnets, OpenCV

Resumo

Processamento de imagem é vastamente utilizado nas mais diversas indústrias. Uma das ferramentas mais utilizadas para realizar processamento de imagem é a biblioteca OpenCV. Também é possível implementar algoritmos de processamento de imagem em hardware, e, em algumas situações, obter melhor performance.

O objetivo principal deste projeto foi avaliar o uso de VHDL, DS-Pnets e IOPT-Flow para o desenvolvimento de sistemas de processamento de imagem em hardware em plataformas baseadas em FPGA. Um conjunto de algoritmos de processamento de imagem foram especificados em VHDL e DS-Pnets e validados usando a ferramenta de simulação do Xilinx ISE e/ou a ferramenta de validação das IOPT-Flow. O gerador automático de código VHDL das ferramentas IOPT-Flow foi usado para traduzir modelos DS-Pnet para código VHDL, que é depois integrado no sistema de processamento de imagem. Para validar o sistema desenvolvido, as implementações em FPGA foram comparadas com implementações em software, realizadas com o auxílio da biblioteca OpenCV. Os modelos DS-Pnet criados foram utilizados para criar uma biblioteca de processamento de imagem nas IOPT-Flow.

Da utilização do sistema de processamento de imagem desenvolvido é possível concluir que as DS-Pnets e as suas ferramentas associadas, as IOPT-Flow, suportam o desenvolvimento de sistemas de processamento de imagem em hardware, nomeadamente em plataformas baseadas em FPGA. Estas ferramentas, que simplificam o desenvolvimento de sistemas de processamento de imagem, estão disponíveis na internet em http://gres.uninova.pt/iopt-flow/.

Palavras chave: Field Programmable Gate Array (FPGA), VHDL, Input Output Place Transition Flow (IOPT-Flow), Processamento de imagem, DS-Pnets, OpenCV

Table of contents

1 Introduction 1

1.1 Background and Encouragement 1

1.1.1 Background and understanding the challenge 1

1.1.2 Encouragement and reasons why use VHDL and IOPT-Flow 2

1.2 Objectives and Contributions 3

1.3 Document structure 3

2 State of Art 5

2.1 Image processing 5

2.2 Technologies used for image processing 7

2.2.1 MATLAB 7

2.2.2 OpenCV 11

2.2.3 C++ 15

2.2.4 Some applications used for image processing 16

2.2.5 Conclusions 16

2.3 FPGA 16

2.3.1 Origins and introduction to the FPGA 17

2.3.2 Why the FPGA can be used for image processing 19

2.3.3 Use of the FPGA in real life projects 20

2.3.4 High Level Synthesis 22

2.3.5 Other uses for the FPGA 26

2.3.6 Conclusions about the use of the FPGA for image processing 27

2.4 DS-Pnets and IOPT-Flow 27

2.4.1 DS-Pnets 27

2.4.2 IOPT-Flow 29

2.4.3 Final remarks 31

3 Image processing validation framework 33

3.1 Introduction 33
3.2 Validation platform 34
3.2.1 FPGA-based implementation board 34
3.2.2 FPGA development environment 35
3.2.3 Serial Terminal 36
3.2.4 Operating system 37
3.3 Validation flow 38
3.4 Implemented modules 40
3.4.1 Memory Interface Generated module 41
3.4.2 Memory Interface Controller Module 44
3.4.3 UART modules 46
3.4.4 Main controller 51
3.4.5 Main File 53
3.4.6 UCF File 53
3.5 Pre-processing and evaluation application 53
4 GLCM algorithm module 57
4.1 Introduction 57
4.2 System structure 58
4.3 Convert to Gray module 59
4.4 System behaviour 60
5 DS-Pnets and IOPT Flow in image processing 63
5.1 Introduction 63
5.2 Green channel and Red channel 63
5.3 Negative 66
5.4 Bright and contrast 67
5.5 Simple grayscale 68
5.6 Weighted grayscale 68
5.7 Binarization 69

5.8 Four weighted grayscales 69
5.9 Four Weighted grayscale with register 70
5.10 Horizontal projection 71
5.11 Closing remarks 73
6 Validation 75
6.1 Introduction 75
6.2 Green channel and Red channel 76
6.3 Negative 78
6.4 Bright and contrast 78
6.5 Simple grayscale 79
6.6 Weighted grayscale 80
6.7 Binarization 80
6.8 Four weighted grayscales 81
6.9 Four weighted grayscales with register 81
6.10 Horizontal projection 82
6.11 GLCM algorithm execution 83
6.12 Results discussion 84
6.12.1 Execution times 84
6.12.2 Outputs 85
7 Conclusions and future work 87
References 89

Acronyms

ASIC Application Specific Integrated Circuit

BRAM Block RAM

CSV Comma-Separates Values

CMOS Complementary Metal Oxide Semiconductor

CPLD Complex Programmable Logic Device

CLB Configurable Logic Block

DS-Pnet Dataflow, Signals and Petri net

DDR2 Double Data Rate type 2

FPGA Field Programmable Gate Array

GPU Graphical Processing Unit

GLCM Gray-Level Co-Occurrence Matrix

HDL Hardware Description Language

HLS High Level Synthesis

IOPT Input Output Place Transition

LUT Look Up Table

PNML Petri Net Markup Language

RAM Random Access Memory

RGB Red, Green, Blue

UART Universal Asynchronous Receiver Transmitter

USB Universal Serial Bus

VHDL VHSIC Hardware Description Language

VGA Video Graphics Array

List of Tables

Table 2.1: Main industries where image processing it is applied and some examples. 5

Table 2.2: Main functions of MATLAB to process images [14]. 8

Table 2.3: HLS detailed process in Xilinx Vivado HLS...................................... 25

Table 3.1: Configuration Parameters for UART (Realterm). 37

Table 3.2: DDR2 interface configuration parameters. 41

Table 3.3 Parallel processes in Imig module and their description....................... 44

Table 3.4: Main operations controlled by ControlImig. 52

Table 4.1: Implemented GLCM algorithm equations. 57

Table 6.1: GLCM execution time. 83

Table 6.2: Results obtained from executing the algorithms in the FPGA vs OpenCV. 83

List of Figures

Figure 2.1: Graphical interface in MATLAB [16]. 9

Figure 2.2: Graphical interface for histogram equalization in MATLAB [17]. 10

Figure 2.3: Image of oil processed in MATLAB [20]. 11

Figure 2.4: Sobel and Laplacian filters applied on an image in OpenCV [21]. 12

Figure 2.5: Sample flowchart for the recognition of shapes in OpenCV (adapted from [22]). 13

Figure 2.6: Application of the algorithm for the recognition of vehicles in OpenCV [23]. 14

Figure 2.7: Comparison of logic composed CPLDs and FPGA (adapted from [28]). 17

Figure 2.8: Modelling capability between Verilog and VHDL (adapted from [29]). 18

Figure 2.9: Processing time in milliseconds for MATLAB, C/C++ and the FPGA for images of different sizes (adapted from [32]). 20

Figure 2.10: Comparison of the performance of the FPGA using VHDL, SystemC and OpenCV [34]. 20

Figure 2.11: Robot grabbing a blue object after the recognition and processing of the objects image by the FPGA [35]. 21

Figure 2.12: Algorithm for car detection made in OpenCV and converted to VHDL, being executed in the FPGA [12]. 22

Figure 2.13: Use of RTL to implement programming model in different platforms (adapted from [38]). 23

Figure 2.14: Use of Xilinx Vivado HLS to implement programming model in different platforms (adapted from [38]). 24

Figure 2.15: HLS process in Xilinx Vivado HLS (adapted from [38]). 24

Figure 2.16: Histogram flowchart for its implementation if C++ (adapted from [39]). 25

Figure 2.17: Output of the Histogram and Histogram Equalization implemented using Xilinx Vivado HLS tool [39]. 26

Figure 2.18: DS-Pnet elements. 28

Figure 2.19: IOPT-Flow editor. 29

Figure 2.20: IOPT-Flow simulation tool.. 30

Figure 2.21: Timing diagram. 31

Figure 2.22: Automatic code generators. 31

Figure 3.1: The block diagram of an image processing FPGA-based system. 33

Figure 3.2: Simplified block diagram of the validation platform. 34

Figure 3.3: Avnet Spartan-3A DSP 1800A Video Kit. 35

Figure 3.4: Xilinx ISE 14.7. 36

Figure 3.5: Realterm. 36

Figure 3.6: Realterm configuration tab. 37

Figure 3.7: Validation platform block diagram.. 38

Figure 3. 8: Validation flow. 39

Figure 3.9: Validation platform modules block diagram. 40

Figure 3.10: Memory interface generation for DDR2. 41

Figure 3.11: DDR2 initialization [46]. 42

Figure 3.12: DDR2 write operation [46]... 42

Figure 3.13: DDR2 read operation [46] 43

Figure 3.14 : Imig state chart. 45

Figure 3.15: UART receiver state chart. 48

Figure 3.16: UART receiver aux 128 bits state chart... 49

Figure 3.17: UART transmitter state chart.. 49

Figure 3.18: UART transmitter aux 128 bits state chart. 50

Figure 3.19: C# program state chart... 54

Figure 3.20: Visual Studio 2017. 56

Figure 4.1: GLCM Algorithm block diagram. 58

Figure 4.2: Grayscale module. 59

Figure 4.3: GLCM module activity diagram. 61

Figure 5.1: Red channel component. 63

Figure 5.2: Green channel component in the IOPT-Flow editor. 64

Figure 5.3: Green channel simulation. 64
Figure 5.4: Green channel timing diagram. 65
Figure 5.5: Green channel generated VHDL code. 66
Figure 5.6: Negative module. 67
Figure 5.7: Bright and contrast module. 68
Figure 5.8: Simple Grayscale module. 68
Figure 5.9: Weighted Grayscale module. 69
Figure 5.10: Binarization module. 69
Figure 5.11: Four weighted grayscales module. 70
Figure 5.12: Four weighted grayscale with register module. 71
Figure 5.13: Horizontal projection component. 72
Figure 6.1: Image used for testing with dimensions 640x480. 76
Figure 6.2: Image used for testing with dimensions 640x48. 76
Figure 6.3: Red channel algorithm simulation. 77
Figure 6.4: Red channel algorithm implementation in EmguCV. 77
Figure 6.5: Green channel algorithm simulation. 77
Figure 6.6: Green channel algorithm implementation in EmguCV. 77
Figure 6.7: Negative algorithm simulation. 78
Figure 6.8: Negative algorithm implementation in EmguCV. 78
Figure 6.9: Bright and contrast algorithm simulation. 79
Figure 6.10: Bright and contrast algorithm implementation in EmguCV. 79
Figure 6.11: Grayscale simpler version algorithm simulation. 79
Figure 6.12: Grayscale simpler version algorithm implementation in EmguCV. 79
Figure 6.13: Grayscale optimized algorithm simulation. 80
Figure 6.14: Grayscale optimized algorithm implementation in EmguCV. 80
Figure 6.15: Binarization algorithm simulation. 80
Figure 6.16: Resulting image from the Binarization algorithm. 81
Figure 6.17: Grayscale optimized algorithm for 4 pixels simulation. 81
Figure 6.18: Grayscale with a register algorithm simulation. 82
Figure 6.19: Horizontal projection algorithm simulation. 82

Figure 6.20: Horizontal projection algorithm simulation. 83

Figure 7.1: FPGA vs OpenCV execution times. 84

1 Introduction

This chapter introduces the main subjects of the document. These subjects are focused on the use of FPGA (Field-Programmable Gate Array) for image processing, technologies that are currently used to process images (like OpenCV) and on the use of IOPT (Input-Output Place-Transition) Flow to facilitate FPGAs reconfiguration. The challenges and motivations to develop this document are also presented with detail and critical sense. Finally, it is presented the document structure and how it is organized.

1.1 Background and Encouragement

This sub section is set to describe the background that led to the development of this document, and the main reasons why it was so encouraging to start it.

1.1.1 Background and understanding the challenge

The image processing subject has evolved in the past years, and it has become an indispensable tool to a lot of industries, such as medical, manufacturing, photography and more. The need to recognize objects in pictures, find some patterns in medical images such as x-ray images or even just editing a single photo in your laptop has increase so much in the past years that powerful tools are needed to optimize these processes.

But what is, in fact, an image? An image is a digital representation of visual or abstract information. The visual information can be acquired using sensors like a photodiode that uses a green pass filter to increase the sensor output for a specific colour. This sensor can be used to generate 2D images using x- and y-directions between the area of interest and the sensor. For an image acquisition in 3D, a camera can be used [1].

After acquisition, images are stored in memory, and become available to be processed. Ann example of an image processing application is the PhotoShop, that allows the image to be transformed using a set of options available by the app [2].

A custom image processing application can also be developed using for example programming languages. Programming languages can be divided in 2 main sets, high-level programming languages and low-level programming languages.

The native language of a computer is binary, and everything from instructions to sets of data must be given in this form, being this language called machine language. Later, the

hexadecimal format was introduced, allowing each digit to represent 16 different values. But this process of developing applications is not practical. To abstract these operations, assembly languages were created, which have instructions that execute to a set of machine language operations. This type of language is defined as low-level; the programmer needs to be familiar with the hardware architecture in order to write code efficiently, which does not make the language abstract enough to be used in a large scale programming [3].

Later, the high-level programming languages, like MATLAB, C (which can be described as general-purpose language) were introduced providing a higher abstraction level and making application development easier. For the code to be executable by different machines, it must be translated to machine specific assembler languages by programs called compilers. These are then converted to machine code by assemblers [3].

Image processing applications can be developed using high level programming languages. There is a different set of languages, such as the Hardware Description Languages (HDLs), that can also be used for image processing. The purpose of these languages is to describe the behaviour of digital systems, while the programming languages provide a set of instructions for the CPU to perform a specific task. In the past, designers had to manually place gates to define schematics that would be the final circuit, but later the appearance of logic synthesis allowed circuits to be described at a register level. Examples are VHDL (VHSIC Hardware Description Language) and Verilog. The FPGA (Field-programmable Gate Array), which is discussed in chapter 2, uses HDLs [4].

With a vast set of languages to develop an image processing system, some of them can be used more efficiently depending on the image processing task. Processing large images can decrease a system performance, as well as heavy processing. The challenge encountered is to keep the system with a good performance when processing large images, and what is the most suitable language for this task. Some widely used languages for image processing are MATLAB and OpenCV. This document proposes an alternative, the use of VHDL to support FPGA-based image processing systems.

1.1.2 Encouragement and reasons why use VHDL and IOPT-Flow

. In a software language or library like MATLAB, OpenCV or C++, the instructions saved, in memory, are executed sequentially; whereas the VHDL language allows the execution of these instructions in parallel on the hardware, which saves in execution time. As a simple example, if there is an instruction implemented in software with the purpose of adding two numbers, to perform the multiplication operation the processor must execute this instruction several times to get the desired result. If the same instruction is implemented in hardware, it can be implemented using a multiplier that performs the operation in a single execution step (during the clock period). This is the main motivation to use VHDL, because it supports implementations that can accelerate the processing task.

To allow a faster development of VHDL code, the IOPT-Flow tool is used. The IOPT-Flow is a tool framework that supports systems: specification through an intuitive graphical modelling formalism that combines Petri nets and dataflows; validation through simulation tools; and implementation through automatic code generators [5].

1.2 Objectives and Contributions

The main objective in the realization of this work is to evaluate the use of FPGAs, VHDL, DS-Pnets and IOPT Flow to perform image processing. To achieve this objective, a research about the subject was made, focusing on image processing using several programming languages and the FPGA. The research gives a contextualization on how image processing is done, the best approaches to model an image processing system and the best technologies to do so. The research takes more focus in FPGA, OpenCV and IOPT-Flow tools, since these are the technologies used to achieve the main objectives. In order to implement and test a set of image processing systems, a validation framework was developed. This validation framework supported the implementation of a set of image processing algorithms and filters, some developed, in VHDL, others generated by the IOPT-Flow tools. The collection of the developed IOPT-Flow components is the beginning of an IOPT-Flow library for image processing. From this work it also resulted one published paper on REC'2019 – XV Jornadas sobre Sistemas Reconfiguráveis, February 14-15, 2019, in Minho University, Guimarães, Portugal, with the title FPGA for image processing [6].

1.3 Document structure

The document structure presents intuitively the document's content. The proposed and used document structure is the following:

Chapter 2 is the state of art of image processing. This chapter contains all the information needed about image processing to contextualization of the subject discussed in this document, and the main technologies used for it. Most of this information is referenced from other authors that wrote documents and articles discussing the main topics of image processing. The chapter also describes some alternatives to perform image processing, namely the FPGA and the IOPT-Flow tools.

Chapter 3 presents the proposed validation framework to validate the use of a FPGA-based system for image processing. It also presents the modules developed to achieve that goal, except the image processing modules and modules developed using the IOPT-Flow tools. Those modules are described in Chapter 4 and 5, respectively.

Chapter 4 describes the GLCM (Gray-Level Co-Occurrence Matrix) algorithm. In the chapter is described how the algorithm works, what it aims to achieve and the hardware implementation of it.

Chapter 5 enumerates and describes the image processing modules developed in the IOPT-Flow tools. It also describes the main features of the IOPT-Flow tools that aid the development of modules.

Chapter 6 presents the results obtained by the validation platform. The results obtained consist in tests made to the validation platform in form of simulation, and later results obtained from the use of the validation platform. The final set of results are the single validation of each image processing module developed in the IOPT-Flow tools. This chapter also presents the results discussion. The results integrity is validated in order to achieve the main objective of developing an FPGA-based platform for image processing.

Chapter 7 presents the conclusions. Based on the results discussion, it is discussed if the validation platform can be used as an image processing system and if the IOPT-Flow tools can also be used for developing image processing modules.

The document structure ends with a list of references that support the research presented in this document.

2 State of Art

This chapter describes the historical background and the theoretical concepts behind the subjects discussed in this master's thesis. All the main concepts are enumerated and explained with detail.

In section (2.1) it is discussed why image processing is important, the primordial of processing image and the general requirements needed by modern systems to perform image processing efficiently.

In section (2.2) it is enumerated a compilation of software's and technologies used to perform image processing and the advantages of using one or another. After that, alternatives are discussed.

In section (2.3) it is introduced the FPGA as an alternative to the previously described technologies. It is made an overview of the uses of it, and in detail for processing image and why it was chosen to perform image processing. It is also described what High-Level Synthesis (HLS) is, how it can simplify FPGA-based systems development, and some application in image processing.

In section (2.4) it is described the DS-Pnet modelling formalism and the IOPT-Flow tool framework, and the possibilities that they bring to the engineering community.

2.1 Image processing

An image it is a unique way to represent and store information. Sometimes, we want some specific information out of this set. This is called image processing. Image processing is critical in industries like medical, robotic, military, among others, and so its development became mandatory. There are some industries where image processing has also an important role, like remote sensing, low security systems (in high security systems it's critical), in the production of textiles, and many more [7].

Where can image processing be applied?

Some examples of image processing can be found in the simplest applications. Here are some examples of them ordered by industries in Table 2.1.

Table 2.1: Main industries where image processing it is applied and some examples.

Industry	Example of application
Medical	Medical images such as x-ray images, brain scans, among others.
Manufacturing	Images of interest such as images of the environment surrounding a robot, traffic images, and more.
Security	Processing images/video of video surveillance cameras, cropping them to get a region of the image that is of our interest.
Autonomous vehicles	Cars from the Tesla Motors brand use cameras to get a view of the traffic and process it to give the driver some useful information.
General Systems	Any system using a camera need image processing, or even some tool to edit it.

In the medical industry, large images, such as x-ray image and brain scans, are often processed. These images need a lot of processing, sometimes to apply some predefined filters, shading, brightness, among others, to show if some bone is broken, or if a patient has any brain disorder. So, image processing is crucial for this industry. Sometimes this kind of images are usually big in size and in number, and it is needed to be processed fast [8] [9].

The manufacturing industry is probably which most benefits from image processing. The goal to use this kind of processing it's mainly to recognize patterns, objects, environments, leading to the robot decision on what is it going to do next, where it is going or to create a virtual map of where is it. This processing is a critical task, because the robot needs to decide almost immediately to any change in the "world" it sees [10].

For security, image processing is also essential, because security cameras are constantly getting pictures of their surroundings. As an example, iris recognition is one of the security methods used and involves a very high and reliable processing and can save a lot in power in comparison with a surveillance camera being operating all the time [11].

For autonomous vehicles it is very similar to the robotics industry, they are kind of intertwined. Images of the traffic captured by a camera installed in a car needs image processing so the system can show the traffic information clearer to the driver [12].

Finally, all sort of systems that uses a camera usually needs image processing. Usually filters, brightness, shading, mixing images, all these options are among the most popular solutions. Some examples of general systems where image processing can be applied are the

recognition of basketball players in a court during game, to get some statistics or the players movements in real time [13].

Resuming, can be concluded that there is a need to process images in various industries and image processing is widely used in various systems. In the next section it is going to be discussed how the image processing can be done using different technologies, starting with a small introduction to the technology and them it uses in image processing.

2.2 Technologies used for image processing

In this section it's described some of the most popular and used technologies to perform image processing. In the end, a comparison is made between them so it's understandable which technologies to choose to perform image processing and in what situations might bring some benefits the usage a specific technology.

2.2.1 MATLAB

This sub-section makes an overview on the highs and lows about the MATLAB software usage for image processing, how can be used to perform it and some examples of its use for real-life systems.

2.2.1.1 Brief introduction to MATLAB

MATLAB, developed in the early 1970s by Cleve Moler, it is one of the most popular computing environments that allows the development of algorithms based in mathematical functions, matrix operations and graphical interfaces. It is a high-level programming language with the possibility of simulating a mathematical model. So, we can store an image in a matrix and apply image processing to it [14].

MATLAB's working environment consists in a MATLAB Desktop, which allows the user to know the directory of the files developed and a history of the commands typed in so far. Next it has the MATLAB Editor, where the .m files are developed, debugged and saved. The .m is the extension referred to files developed and compiled in MATLAB [15].

MATLAB provides the possibility to define arrays and matrices of the data type described in the picture above, and some built-in standard arrays like an identity matrix, a matrix full of zero's, full of random numbers, and the maximum size of elements in an array depend of the version of the MATLAB currently used. All the matrix operations known from Algebra can be performed in MATLAB, because it allows the use of matrix arithmetic

operators. Having an image stored in a matrix, we can then apply all the matrix transformations and with that transform the original image.

The compiler used by MATLAB (MATCOM), which converts MATLAB native code into C++ code (high-level programming language discussed later in this section), for a faster computation [15].

2.2.1.2 MATLAB methods for image processing

MATLAB has the image processing toolbox (IPT), which is a collection of functions that extends the features of the MATLAB environment to the operation of processing images. Some of the functions that are useful for simple image processing are the following displayed in Table 2.2.

Table 2.2: Main functions of MATLAB to process images [14].

IPT function	**Example of application**
imfinfo('pout.tif')	Function that displays image information like the size and name of it, the format, width, height colour type, and many more.
imread	Allows the user to read image files of any type (JPEG, PNG, etc), in any virtual format and location and stores it on a variable.
rgb2gray	It does the conversion of an image in rgb to its grayscale equivalent.
image	Displays an image using the current colour map.
impixel	Return the rgb value of the image's selected pixel.
imwrite	Allows the image to be saved in a file with the extension of the user's choice, specifying the quality parameter.

These functions described above are some of the examples of what MATLAB IPT library allows the user to do. It allows the reading of an image, store it in a variable and then change it like the user wants, converting it to grayscale or more. Some image processing operations like the interpolation, which produces visually better results after enlarging or reducing an image is also available in MATLAB IPT library. Other examples are the Nearest Neighbour Interpolation and the Bilinear Interpolation [14].

MATLAB also has a useful tool called Simulink, that allows modelling and simulation of the behaviour of dynamic systems. Its primary interface is composed by blocks of diagrams and libraries to deal with them. It can be used for individuals with a lower background in programming and facilitate between individuals with different technological backgrounds. This tool also produces C code, that can be converted to other languages if needed [16].

To conclude this introduction, MATLAB has a lot more of functionalities and functions that deal with image processing, like histograms, image segmentation, visual patterns recognition and many other that are not referenced here. This section it is all about showing the main functionalities of MATLAB. So far, we can see that MATLAB has potential to be used to process images. In the next sub-section, an analysis is made about some projects where MATLAB is used with the main functionality of image processing and conclude about is efficiency in developing this task.

2.2.1.3 MATLAB as an image processing tool in real life projects

As concluded from the previous introduction to MATLAB, it has potential to be an image processing tool and it can be used in several areas of knowledge. The first example is the use of MATLAB's graphical interface to develop an interactive simulation platform with the purpose of digital image processing. There is a graphical interface developed with some options for the user to choose in Figure 2.1 [17].

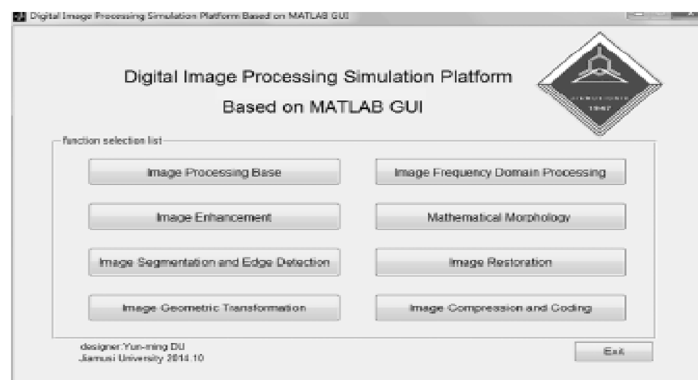

Figure 2.1: Graphical interface in MATLAB [16].

These options define geometric transformations, enhancement in images, and many more, all previously developed in MATLAB. They are made so the user doesn't need to develop any code, it basically shows the user how an image is changed when applied some operation. We can see in Figure 2.2 an example of the user choosing the option of apply a histogram operation to the original image, and then see the result. It also has a description of the operation, what it does and the outcome of it [17].

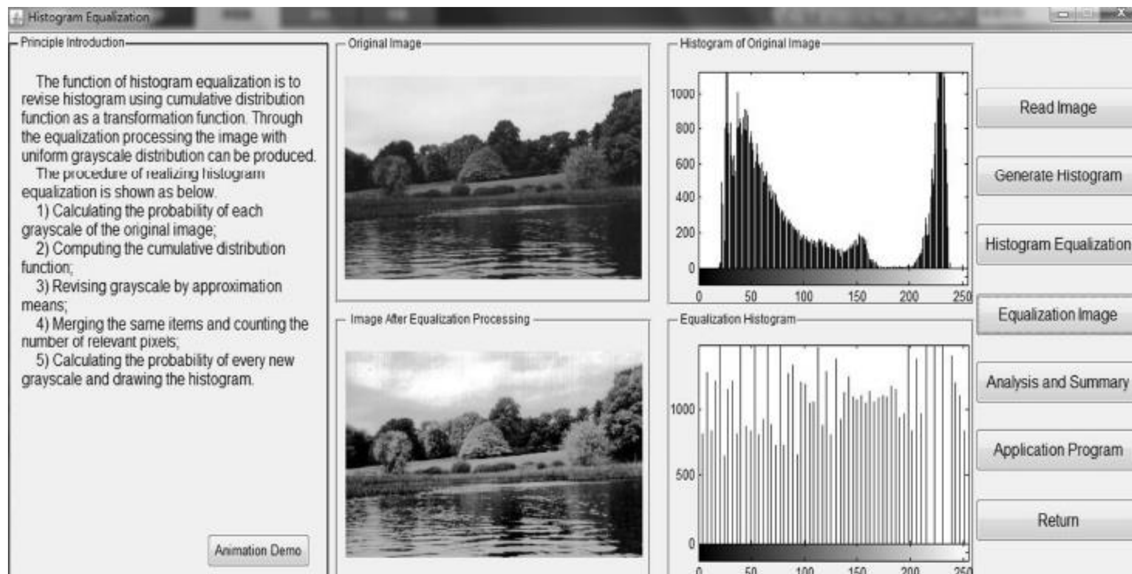

Figure 2.2: Graphical interface for histogram equalization in MATLAB [17].

This project was developed with the purpose of being an auxiliary tool to teaching how image processing it is done, as well as a way of help students to consolidate their theoretical knowledge of image processing by watching its application, and to stimulate their interest on the subject [16].

Another example of the application of MATLAB for image processing involves a robot that must move to the direction chosen by the user. The user will be using a red cap in one finger, recorded in real time by a webcam in a laptop. If the red cap points to the right, the robot must go right, if it points to the top the robot must follow forward. The user chooses where the robot must go, and then the frame is processed in MATLAB where it takes the colour of the red cap, converts the whole frame to grayscale and then black and white. Finally, it puts the red colour in the frame. Then it is tracked the red cap with a square, so it can detect the changes of it, which means detect the changes of direction made by the user [18].

Another project like this, that can be applied to the medical and robotic industries is the control of a wheelchair via Iris movement using MATLAB for image processing. It's like the previous project, there's a camera that gets frames from the iris of the human eye and MATLAB process's the movement of it, sending then the information to the Arduino that controls the motors of the wheelchair. This project allows people with mental or physical disabilities to move without any effort, while using MATLAB to process their iris frames [19].

2.2.1.4 Alternatives to MATLAB

There are some alternatives to MATLAB. One of them, that is going to be described after this MATLAB section is the OpenCV. OpenCV is a popular open source computer vision library with a lot of image analysis and video analysis. In an experiment made, applying the

Sobel filter to an image with resolution 1024x768 it can be concluded that OpenCV is faster in this situation as expressed in Figure 2.3, so it is worth discussing it [20].

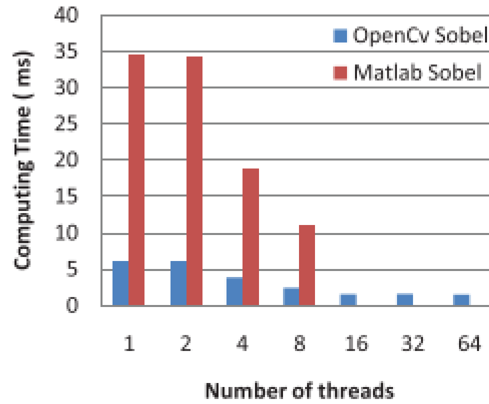

Figure 2.3: Image of oil processed in MATLAB [20].

2.2.1.5 MATLAB, final overview

To end this discussion about MATLAB, we can conclude that it can be used to process images. It can be used in several projects in a lot of different industries with different requirements as we saw, and it can escalate and its user friendly having a graphical interface.

Next sub section makes an overview about OpenCV.

2.2.2 OpenCV

This sub section makes a general overview about OpenCV, how it can be used for image processing and its application has an image processing tool in real life projects. It is also made a comparison with MATLAB in terms of image processing.

2.2.2.1 Brief introduction to OpenCV for image processing

OpenCV was started and developed by Gary Bradsky at Intel in the year 1999 and released in 2000, having its continued with the support of Willow Garage. OpenCV. All algorithms are implemented in C++, but those algorithms can be used for different languages like Python. OpenCV has a binding generator that enables uses to call C++ functions from Python [21].

OpenCV has the same way to access images than MATLAB has, using for example the imread function that stores the image (of any type, JPEG, PNG, etc) in a matrix. Of course, it also has ways to deal with matrices and access single or multiple positions on it. It is also available geometric operations too, like translation, rotation, reflection and many more [21].

OpenCV also has filters to highlight borders and other tiny details within images. Some examples are the Sobel and Laplacian filters. Both use derivatives to highlight borders and details, and the code and its output are displayed in Figure 2.4, with the Sobel output on the left and Laplacian on the right [21].

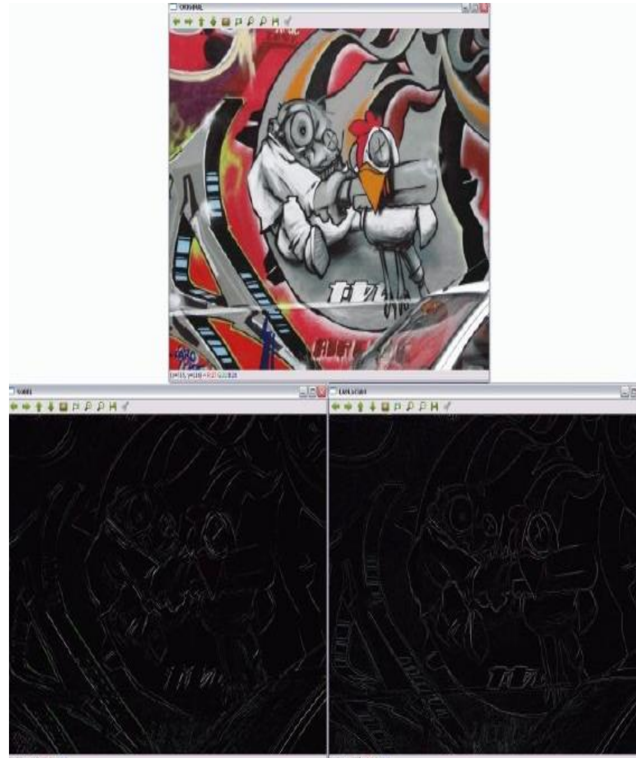

Figure 2.4: Sobel and Laplacian filters applied on an image in OpenCV [21].

It can be concluded that OpenCV has all the features to be a good tool for image processing. It has a way to get an image, process it by putting it on a matrix and applying geometric transformations and filtering, like MATLAB. In the next sub section, an overview about the use of OpenCV in real life projects with the purpose of performing image processing is made.

2.2.2.2 Use of OpenCV for image processing and acquisition in real life projects

In this sub section it is going to be introduced the use of OpenCV in real life projects, with the purpose of using it for image processing. The first example is the recognition of shapes like triangles, squares and rectangles in a given image. Toys have the described shapes that are going to be recognized and separated. In Figure 2.5 it is revealed the steps taken to differentiate the different types of shapes [22].

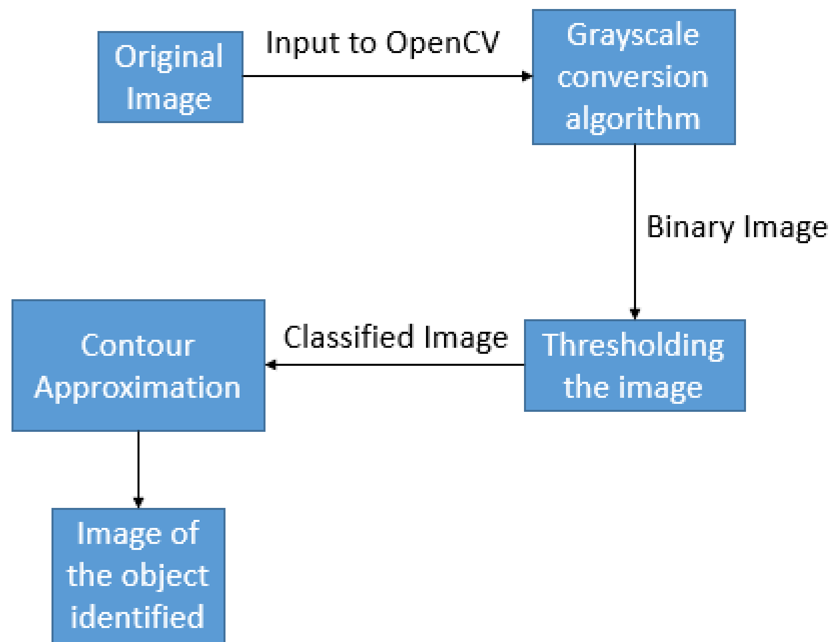

Figure 2.5: Sample flowchart for the recognition of shapes in OpenCV (adapted from [22]).

Those steps consist on a previous phase where the image is already stored in a file and is the input to the OpenCV interface. It is used the OpenCV image processing library in Python to perform a grayscale conversion, adjusting the threshold of the image and the contour approximation. The grayscale conversion has the objective of transforming the image in shades of gray, each pixel having its own intensity of gray (different level of shade). The thresholding is adjusted so the value of the pixels can be classified, and after this classification the contours are defined, which means a curve it's defined joining all the points with the same intensity in a continuous manner, giving us the shape of the object. Finally, the number of different shapes is counted, each one being an object like a rectangle, square, triangle or unknown if none of the others matched [22].

Another example of OpenCV applied for image recognition is in an intelligent traffic management system. The proposed methodology starts with video acquisition, composed by multiples cameras and processing the input through some filters and algorithms. The next step is video pre-processing where the video frames are processed until the user wants to stop this process. The Video feed is converted from RGB to Grayscale, so the amount of data to be processed is reduced. To end this step, a histogram is made to represent the pixel intensities with correspondent values in a chart [23].

The step of image recognition is next, and it has the point of differentiate cars from trucks, buses, wagons, and many more. There's an interface designed to test the classifier in toy cars and in with real cars in real time, as observed in Figure 2.6. There are more steps that consists in analysis in the number of cars detected by all cameras, so the user can now with

precision if the traffic is congested and the kind of vehicles in it. As observed, OpenCV does an excellent job recognizing image patterns and interfacing with image acquisition devices [23].

Figure 2.6: Application of the algorithm for the recognition of vehicles in OpenCV [23].

The next project where OpenCV was applied to image processing is the detection of different corn grain morphology. Corn is one of the most important and fundamental crops in China, one of the most developed a complex country in the world, so it's quality must be up to the standards made by China agriculture's organization. This project consists on getting an image from the corn grain taken from a video, zoom it to the area of interest, convert it to gray and finally looking for sharp points of grain to make a final image which consist of an ellipse fitting of the corn grain. Then the main parameters for quality control are calculated like the deflection angle, the diameter of it, and many more, all of this available in an interface with the user [24].

To conclude this sub-section, OpenCV can be applied to image processing, having a great interface to use video cameras. In the next sub-section, OpenCV is compared to alternative technologies like MATLAB.

2.2.2.3 OpenCV, final overview

To end the discussion about OpenCV, we can conclude that it can be used to process images, and it is efficient and widely used to do so. It's one of the most popular software used to do this task in the industry of robotics and it's currently free. It can be used in several projects in a lot of different industries with different requirements as we saw, and it can escalate.

The next sub section describes one of the programming languages used for OpenCV implementation, C++, and how it can have its uses for image processing.

2.2.3 C++

This sub section makes a quick overview to the C++ programming language and if it can be sued for image processing in real life.

2.2.3.1 Brief introduction to C++ for image processing

C is one of the most popular high-level languages and used around the world. It was developed in the 70's by Dennis Ritchie and 90% of the Unix operating systems was developed with this language.

Like the previous programming languages described above C enables the use of functions used to processing images, like histograms and edge detections, among others. Also, the simplest image transformations are allowed like rotation, inversion, scaling, and more, which makes it a programming language suitable for the task of image processing [25].

About C++, it's an upgraded version of the C language. It is widely used because is it a more complex and upgraded version of C, although in terms of image processing has the same features as C but grouped in libraries. There are some examples of use of C++ for image processing in real life projects, as described in the next sub section.

2.2.3.2 Use of C++ for image processing and acquisition in real life projects

The C++ language can be used in several projects with the purpose of image processing. The first example is its application in the medical industry. It was developed a system in C++ for reconstruction and enhancement of a 3D medical image with the objective of enhance the visual effects on the image and surgery precision with the final goal of drawing a 3D human organ. The code was developed using Microsoft Visual Studio 2012 as the chosen development tool in Windows 8 OS [26].

The image is acquired and inserted in the system with the image format of JPEG, PNG, or others, and after that it is processed using some 2D image processing algorithms to adjust the brightness and other, and finally reconstruct it using an 3D reconstruction algorithm, all these algorithms developed in C++. Some options of visualization are given in an interface for the 3D reconstruction operation like zooming and rotating. After that, a low pass filter to smooth the image is applied, and the subtraction of the smooth image is done to the original, giving as final result a clear image [26].

Finally, the image is reconstructed to the 3D format, using a lot of transformations and a pre-defined C++ algorithm, which lets us conclude that C++ was useful as an image processing language.

This was an example of how to use the C++ language to the task of processing images. It can do it, but it's more used in OpenCV. So, it can be stated that C++ can be used alone as an image processing language.

2.2.4 Some applications used for image processing

For the task of image processing there are other solutions in the market besides the solutions presented in this chapter so far. These solutions can be expensive, so this sub-section is going to cover just their main features.

Agisoft Photoscan it is a software application designed to generate 3D models from pictures taken in different perspectives (angles). It is one of the most popular in this area and its main goal is to process aerial images and it needs a better hardware than the hardware used by their competitors, and it needs more RAM than them [27].

Another software application, one that displays the best performance results is Pix4D, because the post-processing can be done inside the software without the need to export to other software like the others do [27].

Other example of application for image processing is Menci APS, that has a CAD and a 3D view, differing from Pix4D in a more customizable result [27].

2.2.5 Conclusions

It can be stated that the described software can be used for the task of image processing and can be combined with other technologies the same task. All the technologies presented are software languages and are among the most popular used for image processing. The next section describes an alternative to these solutions, the FPGA, which is the chosen solution to implement the image processing framework described in this document.

2.3 FPGA

In this section it is presented the FPGA. It is going to be made a little historic introduction about it, followed by an overview of its uses in image processing and followed by some examples in real life projects. To conclude the section, the FPGA and the other software's mentioned in the last section are compared when used in a real-life image processing system.

2.3.1 Origins and introduction to the FPGA

Any digital logic circuit that performs a task in a system can be represented by 3 types of logic gates, AND, OR and NOT, where AND is the logical conjunction of all signals composed by 0's and 1's (0 represent the absence of signal and 1 represent the existence of it), OR the logical disjunction, or the NOT that inverts the signal representation (applying NOT to the value 0 turns it into 1 and vice-versa). So, in 1980's this led to the development of PLDs (Programmable Logic Devices) by the company Xilinx and many more, which is thousands of these gates in a single chip with the goal of implementing combinational logic. Later, grouping PLDs on a single chip was called CPLDs (Complex Programmable Logic Devices) [28].

A different architecture was proposed in the mid 1980's that consists in RAM-based lookup tables instead of the previous gates described before to implement combinational logic, being these devices called FPGA (Field Programmable Gate Array). These devices contain an array of CLBs (Configurable Logic Blocks) and an array of I/O blocks. As an example, the family of FPGA Spartan-3E from the company Xilinx, in which each CLB has four slices, and each of these slices has 16 x 1 RAM look-up tables (LUTs) (array that replaces runtime computation with a much simpler array indexation operation to save in processing time) that allows the implementation of four variables. Each slice still has 2 flip-flops with type D to act as storage devices for bits [28].

Xilinx Part	No. Of Gates	No. Of IOs	No. Of CLBs	No. of Flip-flops	Block RAM(bits)
CPLDs					
9500 family	800 – 6 400	34 - 192		36 - 288	
FPGAs					
Spartan	5000 – 40 000	77 - 224	100 - 784	360 – 2 016	
Spartan II	15000 – 20 0000	86 - 284	96 – 1 176	642 – 5 556	16 384 – 57 344
Spartan IIE	23000 – 60 0000	182 - 514	384 – 3 456	2 082 – 15 366	32 768 – 294 912
Spartan 3	50 000 – 5 000 000	124 - 784	192 – 8 320	2 280 – 71 264	73 728 – 1 916 928
Spartan-3E	100 000 – 1 600 000	108 - 376	240 - 3 688	1920 – 29 505	73 728 – 663 552
Virtex	57 906 – 1 124 022	180 - 512	384 – 6 144	2 076 – 26 112	32 768 – 131 072
Virtex E	71 693 – 4 074 387	176 - 804	384 – 16 224	1 888 – 66 504	65 536 – 851 968
Virtex-II	40 960 – 8 388 608	88 - 1108	64 – 11 648	1 040 – 99 832	73 728 – 3 096 576

Figure 2.7: Comparison of logic composed CPLDs and FPGA (adapted from [28]).

The old way of designing digital circuits was to draw logic diagrams that contain the previous gates and some logic functions. But with the number of gates and logic functions growing exponentially, this process started to become problematic. Drawing thousands or millions of gates would be a long and hard task. So, to better describe digital circuits, the HDLs

were proposed. These languages allow the user to design a digital system by writing a program that describes the behaviour of the digital circuit [28].

The most widely used HDLs are VHDL and Verilog. The program wrote in these languages can be used to both simulate the operation of the circuit and make the synthesize process to perform an implementation of the circuit. This implementation can be done in a CPLD, FPGA or an ASIC (Application Specific Integrated Circuit). But it is also needed an HDE (Hardware Description Editor) [28].

Since VHDL and Verilog are the most widely used HDLs, it is interesting to find their characteristics and compare them in terms of efficiency. In terms of modelling capability, Verilog does not have the same level of abstraction as VHDL. However, Verilog can model Gates and VHDL cannot without using the VITAL language as described in Figure 2.8 [29].

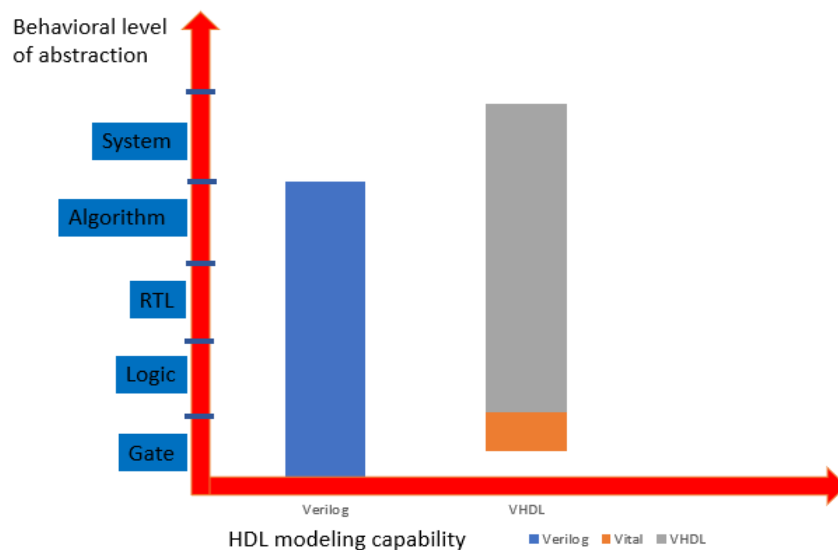

Figure 2.8: Modelling capability between Verilog and VHDL (adapted from [29]).

In terms of compilation VHDL has multiple design units (pairs compost by entity/architecture) that reside in the same file, which allows the units to be compiled in separately if desired. Verilog is rooted in the native interpretative mode, which means that it must be kept wisely the compilation order of the code written in one file and the compilation order of several files. In terms of data types, in VHDL are chosen by the user, but in Verilog's chosen by the Verilog language which is simpler. Finally, in terms of reusability, VHDL has procedures and functions that may be encapsulated in a package, and a sort of packages can be assimilated in a library to become available to any design-unit that needs to use them. In Verilog there's not present the concept of package, functions and procedures must be used within a model that must be defined in a module [29].

It was made an experiment of implementing the greatest common divider in both HDLs, and the final conclusion was that either one of them can get to the solution very fast, which means that the hardware that is modelled in one HDL can also be modelled in the other, that the

choose of the HDL (VHDL or Verilog) it is not based on technical properties but in the company's culture [29].

In [30], it is compared the speed of ASICs and FPGAs by measuring the maximum clock frequency for each of them. The power consumption is measured in the static and dynamic components. The results of the speed test were that in average the ASICs is 3.1 times faster than the FPGA. For circuits with logic, the FPGA is 3.2 times slower than the ASIC's, for circuits that use the block memory it's 2.3 times slower and those that use both are 2.3 times slower as well. In terms of power consumption, the FPGA consumes 12 times more than the ASIC using multipliers, 9.2 times when it is used memories and 9 times when both.

So why use the FPGA? Because it is reconfigurable. It is also one of the reasons for its high consumption in terms of power. ASICs cannot be reconfigured as FPGAs can [31].

So, the conclusion to take out of this discussion between ASICs and FPGA is that FPGAs are usually used to develop prototypes, due to their reconfigurable properties, that later can be implemented in ASICs, since ASIC's performance is usually higher. In the next sub section, it's presented the reasons why the FPGA can be used for image processing.

2.3.2 Why the FPGA can be used for image processing

The FPGA used HDL instead of a programming language, and as described earlier, it can achieve faster processing times. As an example, if we need a surveillance system and it is needed a performance of processing speed less than 40 milliseconds and we can use the programming languages C++, MATLAB or an HDL language in the FPGA, which uses a low-level programming model oriented for hardware. For MATLAB is used the simulator Simulink, and it allows the FPGA hardware to be included by adding the XGS (Xilinx System Generator) block sets in the process of the simulation. This generator converts the Simulink model to be used in the environment Xilinx, so the developer doesn't have to be familiar with the HDL language used, the systems converts the model to it [32].

The image is inserted in a matrix for MATLAB and C/C++, but in the FPGA, there is needed some extra blocks to deal with it since it can't perform multiple dimensional signal processing. The FPGA used is from the family Virtex-5 (ML-555 board). There are available images with size 4KB, 45 KB and 119KB, and the same image processing algorithms are applied. In Figure 2.9 are displayed the execution times for each. The FPGA achieved faster execution times in comparison to MATLAB's and C++'s. and with the increasing of the image size it keeps faster [32].

Image Size & Coding	Minimum Time (ms)	Average Time (ms)
Image1 (119KB)		
C++	46	47
MATLAB	1435.9	1440.3
FPGA	30.72	30.72
Image2 (45KB)		
C++	23	24.5
MATLAB	664.8	666.95
FPGA	14.75	14.75
Image3 (4KB)		
C++	0	0.9
MATLAB	8.1	8.32
FPGA	0.488	0.488

Figure 2.9: Processing time in milliseconds for MATLAB, C/C++ and the FPGA for images of different sizes (adapted from [32]).

The image can be put in the FPGA in a matrix defined by the user or in the memory of the FPGA like the RAM [33].

Another example that shows the FPGA use for image processing is object detection and classification. There is a C++ based class library, SystemC, which can be converted to VHDL. But the implementation of these algorithms can also be made in OpenCV or in VHDL for example. So, it was made a comparison between the time spent in the execution of the image recognition and processing algorithms in OpenCV, in VHDL and in SystemC, which is converted to HDL and downloaded into the FPGA [34].

Why the conversion from SystemC to HDL in the FPGA? Because the image processing algorithms that are used in this problem are too complex to develop efficiently in HDL. The results are presented in Figure 2.10. The implementation in VHDL achieved better performance in this situation than the previous alternatives, OpenCV and System C. From this experiment, it can be concluded that the FPGA can be used for image processing and get good results in some situations [34].

	Demosaicing	Binary morphology	Canny Algorithm
VHDL Spartan-6	**205.7**	**228.7**	**136.9**
SystemC Spartan-6	52.63	175.7	27.4
OpenCV on Intel Core i7 2800 MHZ	**191.6**	**225**	63.9

	Demosaicing	Binary morphology	Canny Algorithm
VHDL Virtex-5	**250.3**	**309.7**	**143**
SystemC Virtex-5	151.1	192.8	49.7
OpenCV on Intel Core i7 2800 MHZ	191.6	225	63.9

Figure 2.10: Comparison of the performance of the FPGA using VHDL, SystemC and OpenCV [34].

The next sub section presents some real-life projects with the goal of performing image processing where the FPGA is used has the primary solution.

2.3.3 Use of the FPGA in real life projects

Now that it is established that the FPGA can be used for image processing and has a good performance, some examples of its use in real life projects are presented.

The first example is to process images taken by a set of cameras in a robot. The experiment consists in having a set of 2 cameras and 2 VGA monitors, being the task of the first camera to detect the shape, the colour blue of an object and position of the object. The object is displayed in real time in the first monitor plus the background, and the colour and position of the object consists in a different task for the robot to perform, being this case to drive the motor to rotate and change the robots direction (left to right for example) according with the position of the blue object. The second camera has a global monitoring trolley robot to monitor the entire area and displays it in monitor 2. The task of the FPGA is to process the image using VHDL to develop the image processing algorithms. The images are received in real time by the cameras and displayed it after prior processing in the monitor trough the VGA interface. In Figure 2.11 it is displayed the robot recognizing the blue object and grabbing it with a claw [35].

Figure 2.11: Robot grabbing a blue object after the recognition and processing of the objects image by the FPGA [35].

The second example that combines the use of the FPGA and OpenCV for image processing is a system with the task of processing video from several cameras in real time. The footage contains cars in the traffic, and the goal is to find out if the traffic is congested and warn the user about it. The video footage is processed by using OpenCV, because of its interface with the cameras, and to choose the area of interest that is going to be processed by the FPGA. For example, the trees and building must be filtered from the original image, so the processing made in the FPGA deal only with the cars in the road. The processing algorithm that deals only with the vehicle detection and counting is made in OpenCV but is converted to VHDL using a MATLAB library. It’s then compiled and executed in the FPGA. Finally, the warning is sent to the user if the traffic is congested. An example of the algorithm detection is shown in Figure 2.12 [12].

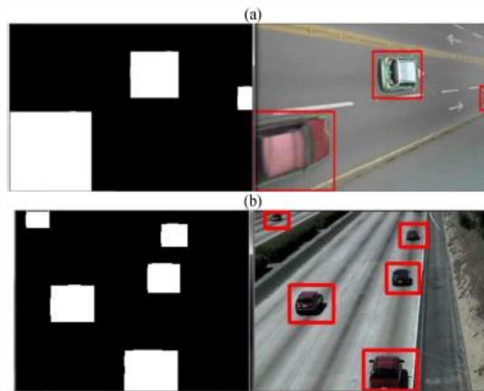

Fig. 9 The algorithm of detection vehicles is applied to the sequence of images.

Figure 2.12: Algorithm for car detection made in OpenCV and converted to VHDL, being executed in the FPGA [12].

There is still one more important tool to discuss, LabVIEW. LABVIEW is a graphical programming and design language that has the same image processing functionalities the text-based languages have. The main advantage of graphical programming is that is not needed a background in text-based programming. It allows system modelling using graphics. Another advantage is that its performance is compared to that achieved by text-based languages because LabVIEW compiles to machine code [36].

LabVIEW FPGA is the part of LabVIEW that integrates its functionality with the FPGA. Block diagrams made in LabVIEW are converted to digital hardware circuits and deployed to the FPGA [36].

An example for the use of the LabVIEW FPGA is acceleration of the image processing using LabVIEW FPGA and a FPGA co-processor to process video that is being recorded by a camera. After some images are recorded by a camera, the host uses LabVIEW as an interface to download them into the FPGA via USB 2.0, which uses a co-processor to speed the process. The image processing then consists in applying a noise filter and an edge detection algorithm. After application, the image is displayed in a monitor using the interface VGA [37].

The next sub section shows other uses for the FPGA besides its use for image processing.

2.3.4 High Level Synthesis

This section as the goal of explain what high-level synthesis is and how can it be applied to the FPGA, using Xilinx Vivado HLS (High Level Synthesis).

2.3.4.1 Introduction to high level synthesis

Software algorithms tend to be developed with the goal of abstracting the developer from the details of the computing platform. There are processors specialized in digital signals (DSP)

and in graphics processing (GPU), being both capable of executing any algorithm written in a high-level language. But they are bounded to use accelerators with a specific function that allows an improved execution of the algorithm in their target software applications [38].

So, the objective is to design algorithms with parallelization in mind, efficient parallelization so the performance of the algorithm can increase. But the techniques used for algorithm design use the same elements of the FPGA design, making it clear that the principal difference between a FPGA and a modern processor is the programming model. So high-level synthesis gives abstraction to a developed solution, being an automated design process that processes an algorithm description of some wanted behaviour and ends up creating hardware with the sole purpose of implementing that behaviour [38].

2.3.4.2 Xilinx Vivado HLS for image processing

There are compilers from C and other high-level languages for different processor architectures, but here it's introduced the Xilinx Vivado HLS (High-Level Synthesis) compiler. It has the same functionalities that some other compilers of C have, but is target is the Xilinx FPGAs. The FGA traditional flow uses RTL (Register Transfer Level) as the main method for design capture. In Figure 2.13 it is shown the importance of the programming model in implementation time and performance in different computation platforms [38].

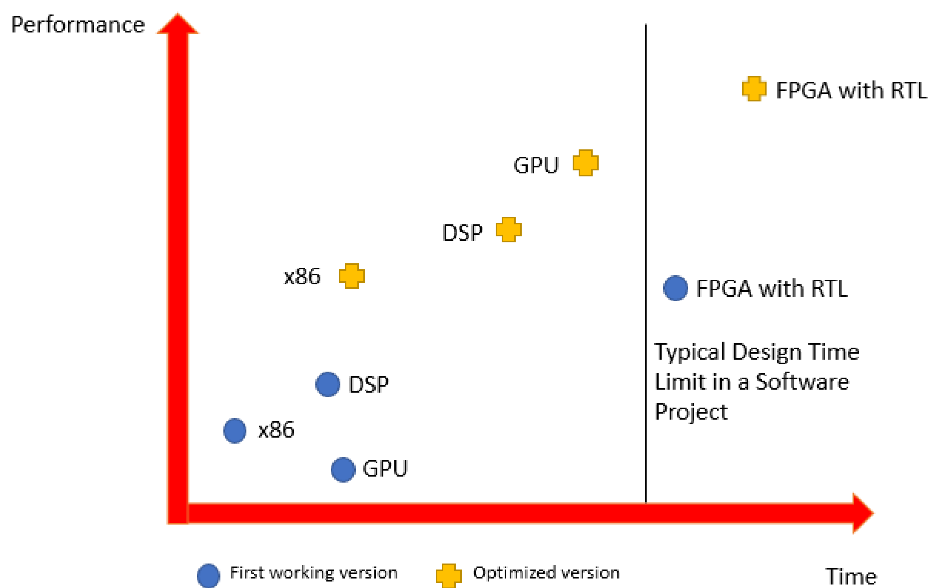

Figure 2.13: Use of RTL to implement programming model in different platforms (adapted from [38]).

In Figure 2.14 is displayed the use of the Xilinx Vivado HLS. It makes the process of transforming a C specification into a RTL implementation that can be later synthetized into the FPGA [38]. The Xilinx Vivado HLS design flow is descripted in Figure 2.15.

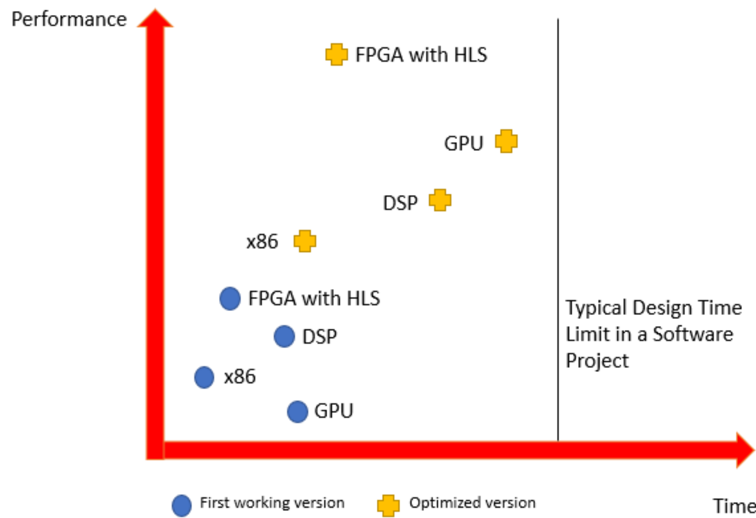

Figure 2.14: Use of Xilinx Vivado HLS to implement programming model in different platforms (adapted from [38]).

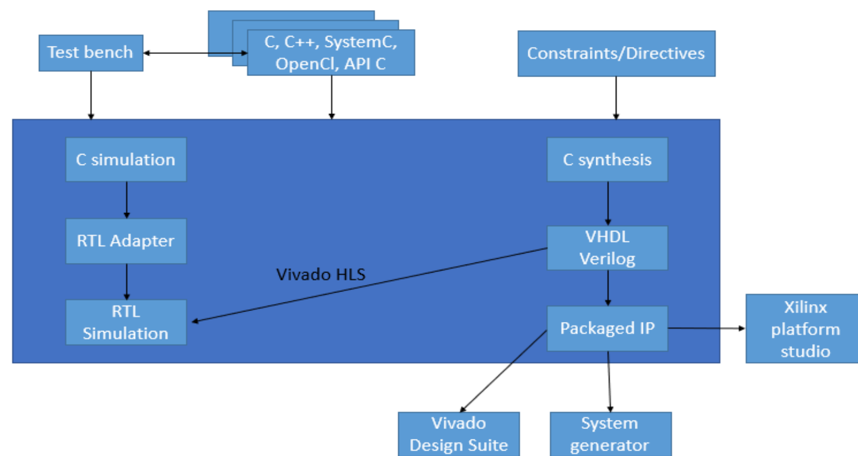

Figure 2.15: HLS process in Xilinx Vivado HLS (adapted from [38]).

The resume of processes used in the HLS on Xilinx Vivado HLS is present in Table 2.3.

Table 2.3: HLS detailed process in Xilinx Vivado HLS.

Sub-process	Description
Compile, simulate and debug the C code.	It compiles, executes and debugs the algorithm that has been developed in C.
Synthetize C into RTL.	Synthetises the previous developed C algorithm into an RTL implementation.
Generate reports and analysis of the design.	Generate reports about the synthesis and it analyses the design.
Verification of the RTL implementation.	Verify the RTL implementation.
Packaging the RTL implementation.	Turns the RTL implementation into a package and selects IP formats for it.

For the image processing task, Xilinx Vivado HSL tool can be used to allow hardware acceleration of image processing algorithms. The goal is to develop image processing algorithms in C and then use HLS to implement these algorithms in the FPGA more efficiently and faster. As an example, to implement a histogram (representation of the frequency of pixels intensity values in an image, giving information about if the image is dark or light) the flowchart to implement the algorithm in C++ is described in Figure 2.16 [39].

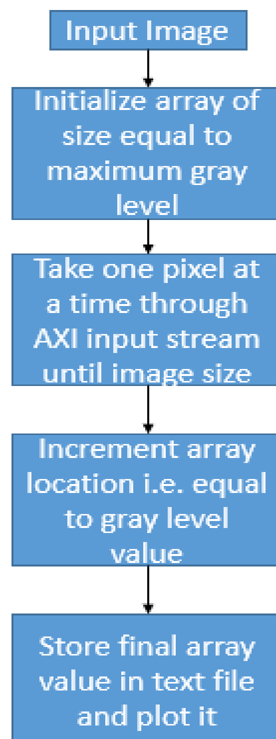

Figure 2.16: Histogram flowchart for its implementation if C++ (adapted from [39]).

There is also implemented the Histogram equalization without pipelining and with pipelining in hardware resources, being the pipelining process introduced by the Xilinx Vivado HLS tool. Figure 2.17 shows the final output of the image [39].

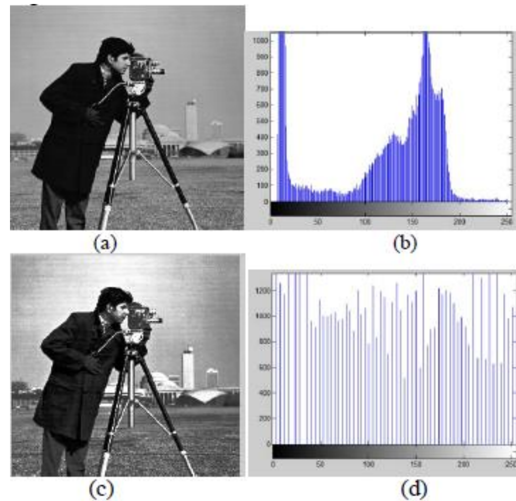

Figure 2.17: Output of the Histogram and Histogram Equalization implemented using Xilinx Vivado HLS tool [39].

To conclude this overview about the Xilinx Vivado HLS tool, it can save time in developing image processing algorithms in VHDL. The synthesis of C code to VHDL may not be perfect, but it can allow a faster execution of the algorithm and less time developing and testing the algorithm.

2.3.5 Other uses for the FPGA

The example described is the use of the FPGA to implement an architecture to digitally control a motor that is connected to an Arduino. It also has a Java graphical interface for monitoring and control the whole process. Basically, a control system is a set of inputs and outputs. There's a controller to lead these processes, and it must operate at a high speed, so the FPGA will do this task and control how the motor should operate. The controller parameters can be chosen and changed in the Java graphical interface. The Arduino is connected to the motor, by USB to the computer and to the FPGA by pins [40].

The FPGA can also be used with another popular device, the Raspberry Pi. A webpage is made with PHP to ask the user which pre-defined operations (like the average filter, sharpening filter) should be done to an image uploaded in the web page. That page is interfaced with the Raspberry Pi, which provides IP address to communicate with the web page. These

operations and the image are given to the FPGA that performs the task of applying the filters [41]. The next sub section deducts conclusions about the use of the FPGA for image processing.

2.3.6 Conclusions about the use of the FPGA for image processing

It can be concluded that the FPGA can in fact be an alternative in developing the task of image processing when compared with high-level programming languages. The use of an HDL can be an alternative, and sometimes achieve best performances than high level programming languages can. The use of the FPGA for other tasks besides image processing unveils some of its potential as a processing tool.

The next section describes the DS-Pnets and IOPT-Flow in detail and how can both be used in the image processing subject.

2.4 DS-Pnets and IOPT-Flow

The DS-Pnet (Dataflows, Signals and Petri nets) modelling formalism and its associated tool framework IOPT-Flow, proposed in this document to support the development of image processing systems, is described in this section.

2.4.1 DS-Pnets

The DS-Pnets, proposed in [42] [43] [44], are an intuitive graphical formalism that combines Petri nets and dataflows. These Petri nets inherits the execution semantics and the main characteristics of the IOPT Petri net class [5]. In the dataflow part, it is considered that the operations are instantaneous and that there is no propagation delay between operations. It has the Petri net class main components like a Petri net place, transition and arc, also an input and output as a signal, or an event. It has a dataflow arc and an operation. The external interface of a model is characterized by inputs and outputs, some can be reading of a sensor or setting of some values in actuators. These signals are associated with I/O pins that are connected to physical devices [5]. The elements presented in Figure 2.18 are available:

- Petri net Place – represented by a yellow circle with the marking, which can be null (zero) or a positive integer, at the centre;
- Petri net Transition – represented by a cyan rectangle;
- Petri net Arc – represented by an arrow, it connects a place to a transition or a transition to a place;
- Read Arc – represented by a blue dashed line with a circle at the end, when the target is a transition, or with a triangle at the end, when the target is another DS-Pnet element;

- Input signal – represented by a green circle, it is used to specify a system input, and can be Boolean or integer range;
- Output signal – represented by a red circle, it is used to specify a system output, and can be Boolean or integer range;
- Events, input or output – represented by green or red diamonds, are only active during a single execution step (in hardware: during a clock);
- Operation – represented by a trapezoid (by default) with a set of inputs and one output, perform mathematical operations;
- Component – represented by a gray rectangle with inputs and outputs, it contains a DS-Pnet model or an external component.

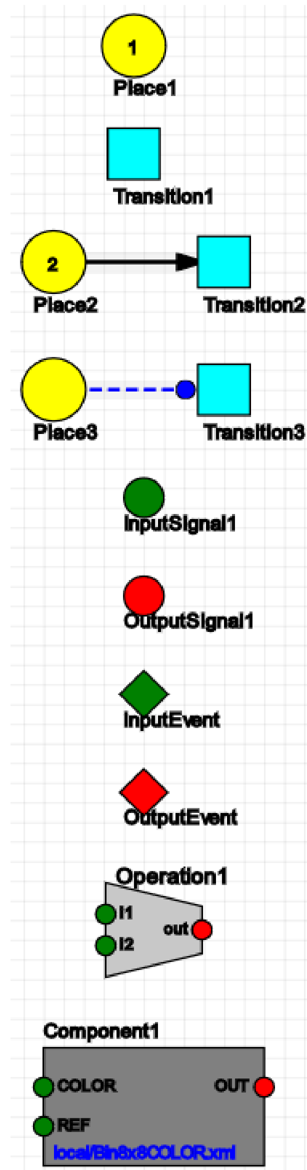

Figure 2.18: DS-Pnet elements.

Components support hierarchical structuring, simplifying models' creation and reading. Complex systems can be specified through multiple models, which are then instantiated in a

main model through components. Components can be instantiated several times in the same model, or reused on future projects, accelerating their development.

2.4.2 IOPT-Flow

To support the use of DS-Pnets in model-driven development approaches, the IOPT-Flow tool framework was developed [45]. This framework has many similarities with the IOPT-tools framework [46], which supports the IOPT Petri nets [5]. Both are Web-based tools that support models' edition, validation, and translation into implementation code. The IOPT-Flow framework supports the development of embedded and cyber-physical systems, generating Javascript, C, and VHDL code. The main IOPT-Flow tools for hardware development are:

- Editor – to create and edit DS-Pnet models;
- Simulator – to validate the developed models, it provides an interactive token-player and produces the associated timing diagrams;
- Automatic VHDL code generator – automatically translates the DS-Pnet models into VHDL code, to support systems implementations in reconfigurable hardware platforms, such as FPGAs.

The editor, presented in Figure 2.19, is a web-based editor that uses the Ajax principles. The editor is divided in three main areas. At the left are available icons to draw elements, rotate, multiply or delete them, create, open, and save models, among other options. Models are saved, in XML format, in the Webserver, but can also be downloaded. The area at the centre is the drawing area; whereas at the right is the properties area, which is displayed when an element is selected.

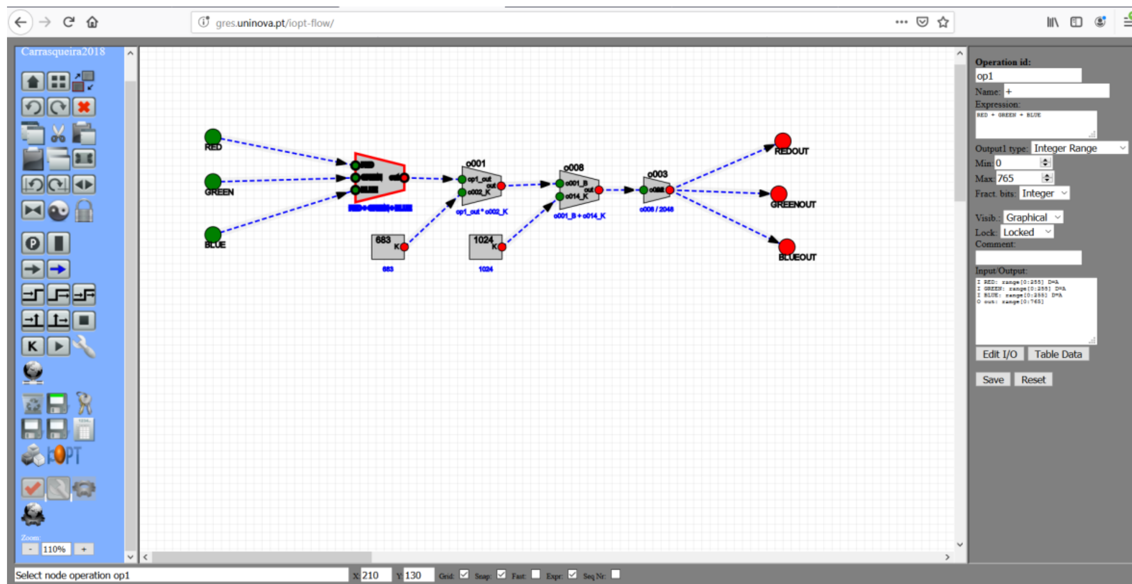

Figure 2.19: IOPT-Flow editor.

The simulation tool is presented in Figure 2.20 and a timing diagram is presented in figure 2.21. At the left the icons that control the simulation execution, save and load simulations, launch the timing diagram, and more. The simulator keeps track of every simulation step. At the centre the simulated model at a specific simulation step. Finally, at the right the model inputs, outputs, place marking and fired transitions, where it is possible to change the inputs.

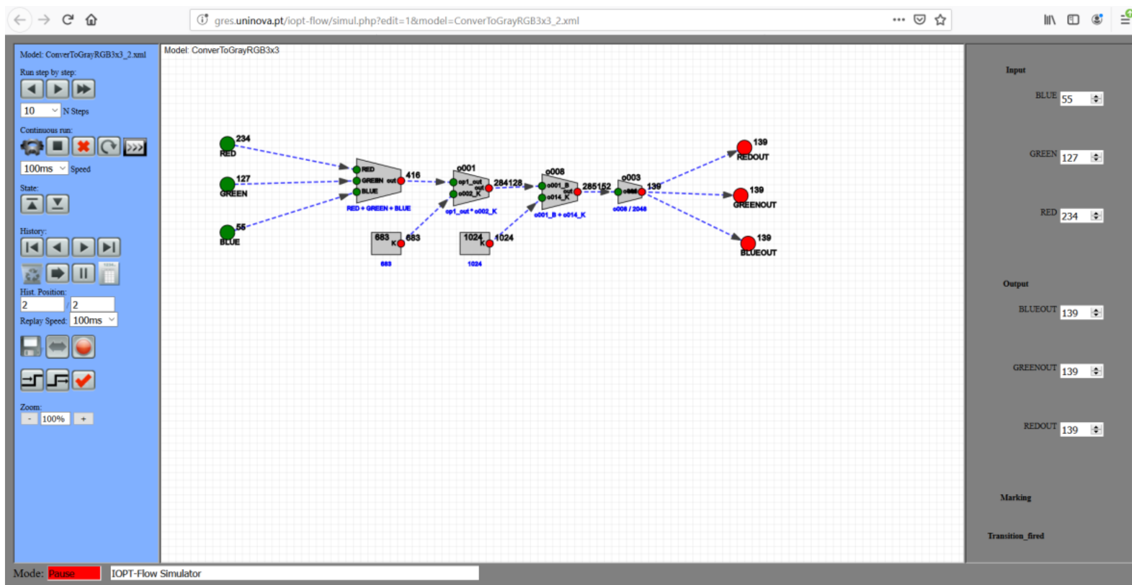

Figure 2.20: IOPT-Flow simulation tool.

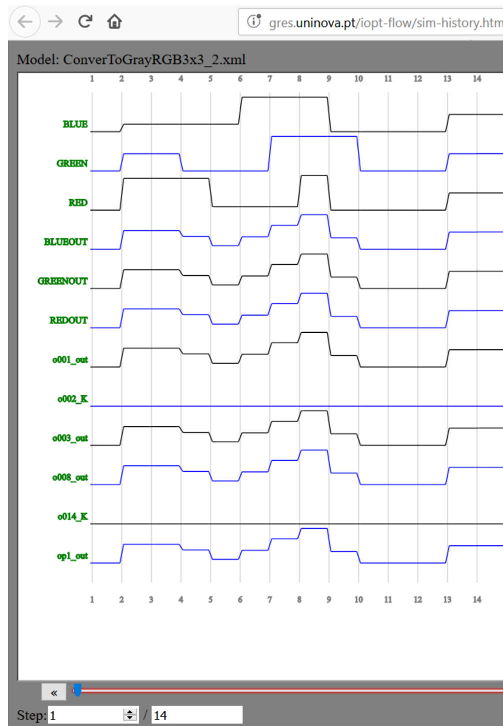

Figure 2.21: Timing diagram.

The automatic code generators do not have a specific user interface. They run on the Webserver and are launched when their icons, presented in Figure 2.22 and available at the left area of the IOPT-Flow editor, are pressed.

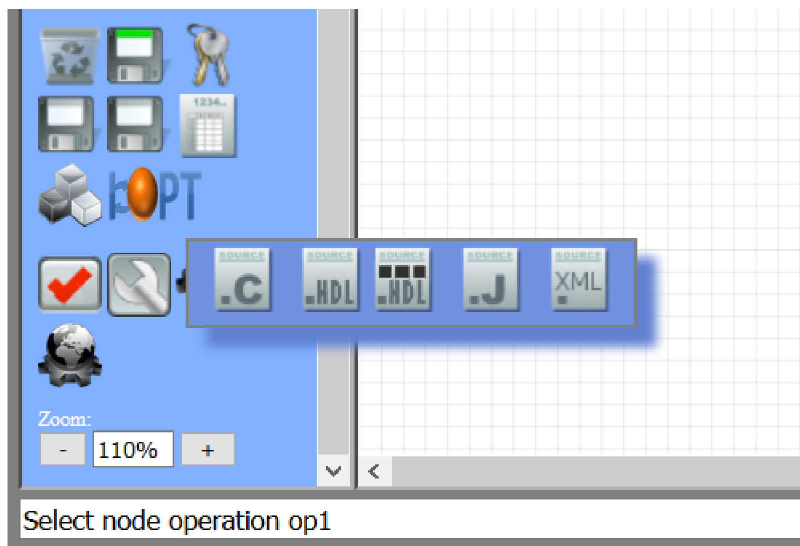

Figure 2.22: Automatic code generators.

2.4.3 Final remarks

We can conclude that the DS-Pnets and IOPT-Flow framework are useful tools to specify and develop embedded and cyber-physical controllers. However, their characteristics look

appropriate for the development of image processing algorithms or even for the full image processing system, to be implemented in hardware platforms, such as FPGAs.

The next chapter describes in detail the developed validation framework, to achieve the initial stated objectives.

3 Image processing validation framework

This chapter makes an argumentative approach on the proposed architecture and the how it was validated.

Section 3.1 describes the proposed architecture with a block diagram. Section 3.2 describes the validation platform developed to test and validate the proposed architecture. Finally, section 3.3 describes the technologies used to implement the validation platform and section 3.4 the implementation of the validation platform.

In order to implement the validation platform, extra technologies need to be used. In this section is made an overview of the used technologies and the configurations needed to make the best use of them for the validation platform needs.

To better illustrate the system implementation and concepts behind each module, the UML editor StarUML was used. UML is an established modelling language for developing systems and software, it was chosen to better model the system. This editor allows the creation of UML statechart and UML flowchart models. The UML flowchart is used to display the flow of a specific module. The UML state chart, which can be defined by being a finite set of state machines with hierarchy, orthogonality and broadcasting, was used to describe all the actions in the system.

The following section will introduce the validation platform, giving an overview about the framework itself from a high-level point of view.

3.1 Introduction

An FPGA-based platform can be used to implement the complete image processing system, or just a part of it. FPGAs are often proposed for image pre-processing, receiving images from the camera and providing processed images and/or their features to another system, usually a computer. The block diagram of a common image processing FPGA-based system is displayed in Figure 3.1.

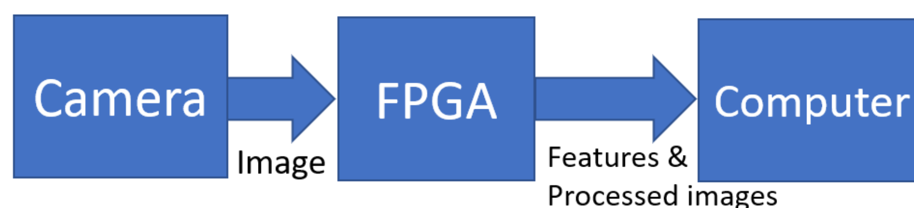

Figure 3.1: The block diagram of an image processing FPGA-based system.

To develop an image processing system in hardware, it is required to validate if it meets the necessary requirements to produce meaningful results. Those results need to be valid in the context of image processing and comparable with similar results from other image processing systems. Section 3.2 presents the proposed validation platform for FPGA-based systems.

3.2 Validation platform

To validate FPGA-based image processing systems, a validation platform was developed. Its simplified block diagram is displayed in Figure 3.2. The platform consists in an FPGA, a computer and a DDR2 SDRAM (Double Data Rate type 2 Synchronous Dynamic Random-Access Memory), hereinafter referred as DDR2. Here, the image is sent from the computer to the FPGA, where it is processed. In the FPGA, it is stored in a DDR2, and the processing algorithms are executed. After the extraction of the image features, the FPGA sends to the computer the processed images and/or their features.

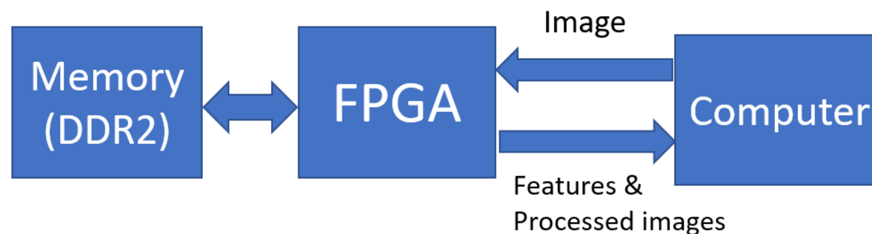

Figure 3. 2: Simplified block diagram of the validation platform.

3.2.1 FPGA-based implementation board

The FPGA-based implementation board, of the proposed validation platform, is the Avnet Spartan-3A DSP 1800A Video Kit, displayed in Figure 3.3. It includes the Xilinx Spartan-3A DSP 1800A Starter Board, the Avnet EXP PS Video Module and an LCD panel. The Xilinx Spartan-3A DSP 1800A Starter Board contains a 128MB (32M x 32) DDR2 SDRAM, which will be used to store the images and their features.

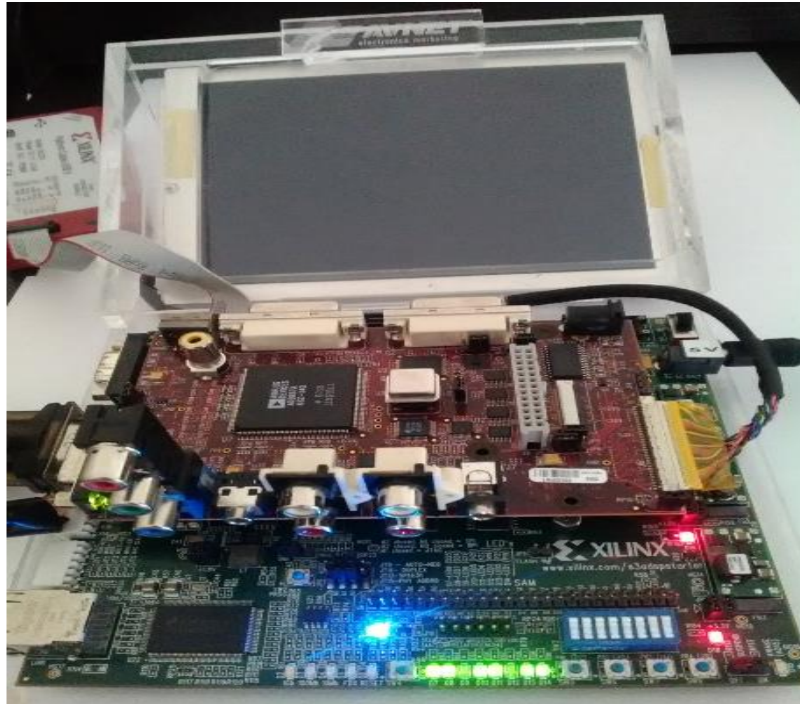

Figure 3.3: Avnet Spartan-3A DSP 1800A Video Kit.

3.2.2 FPGA development environment

The Xilinx ISE 14.7 is the software tool chosen to synthetize and analyse HDL projects, as well as setting a target device to program. It supports the VHDL language, which is used to program the chosen Xilinx FPGA for the validation platform. It is important to highlight another Xilinx ISE tool, the IP Core Generator, which was used during this work, for example, to generate the DDR2 interface. The used version is the latest; however, Xilinx ISE it is not updated since 2013. The Xilinx software environment that replaced ISE is the Vivado Design Suite, but it does not support the device being used. A snapshot from Xilinx ISE is displayed in Figure 3.4.

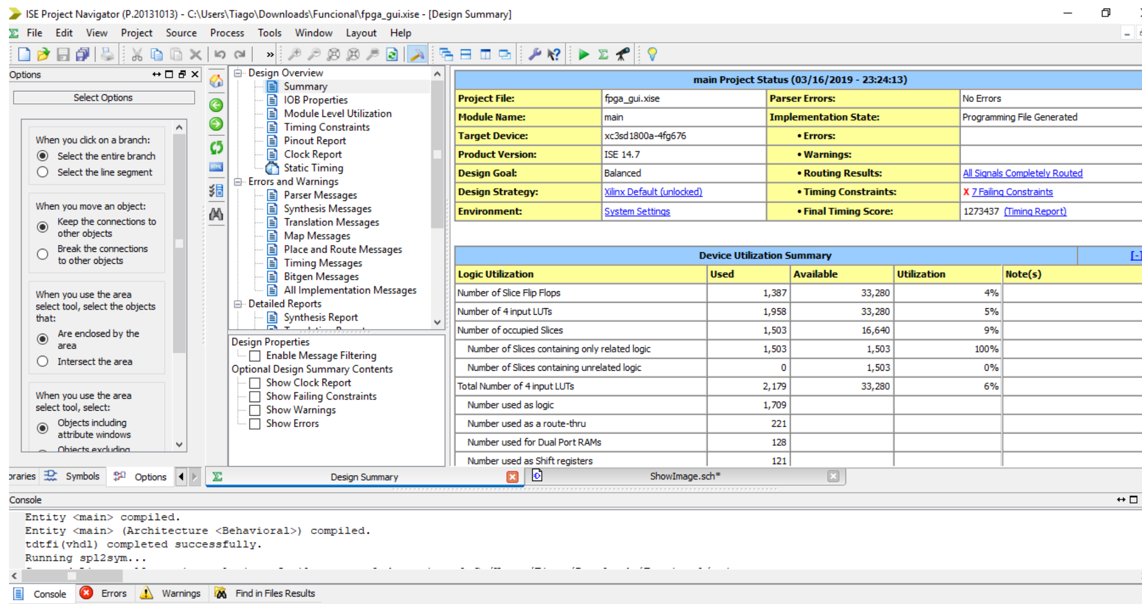

Figure 3.4: Xilinx ISE 14.7.

3.2.3 Serial Terminal

To send images from the computer to the FPGA-based system and receive the processed images and/or their features, the Realterm software application is used. It is a serial terminal program that allows the user to control which images are sent and at what baud rate. It is the UI (User Interface) of the validation platform, not only used to send images, but also to receive data and present it to the user, or even to save it in a file. The version of the program used is the 2.0.0.70 and is displayed in Figure 3.5.

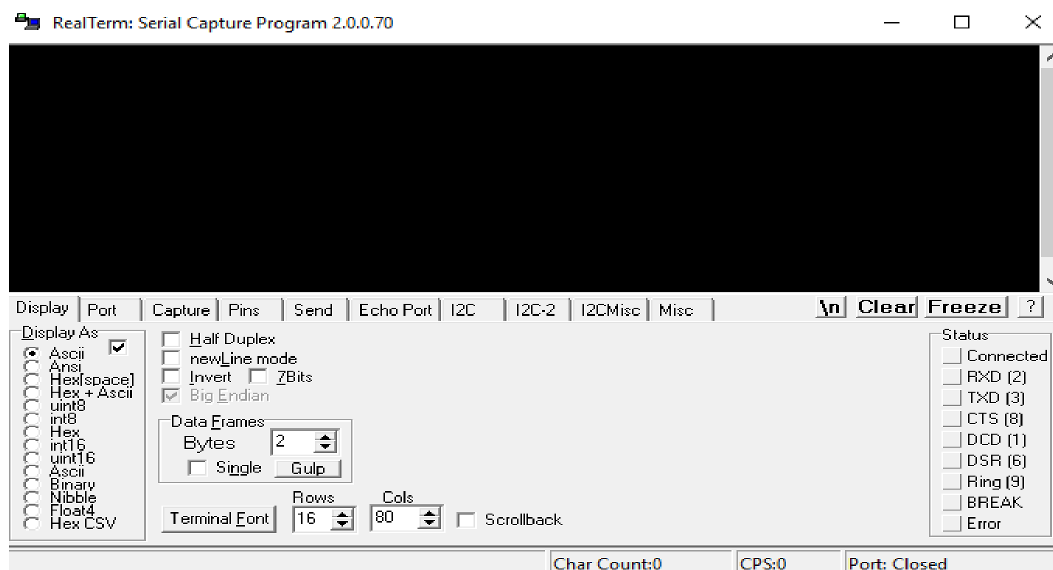

Figure 3.5: Realterm.

For the validation platform, the baud rate (rate at which the serial data is transmitted in bits per second) was set to 9600 bps (bits per second) initially for initial tests, but since Realterm allows a maximum of 921600 this was the value chosen in the end, to reduce the transmission time. It is also required to select the serial port, although it is selected by default when the serial to USB cable is detected. The other parameters were set to default. In Table 3.1 are the configuration parameters and in Figure 3.6 a screenshot of the configuration tab in Realterm. The UART (Universal Asynchronous Receiver/Transmitter) parameters needs to be configured in the Realterm and in the FPGA. In both, the data bits are set to 8 bits, the parity bit is disabled, which is used for transmission errors detection, 1 stop bit is enabled and no flow control is done.

Table 3.1: Configuration Parameters for UART (Realterm).

Baud rate (bits per second)	921600
Number of Data Bits (7 or 8)	8
Parity Bit (On, Off)	Off
Stop Bit (0, 1, 2)	1
Flow Control (None, On, Hardware)	None
Port	Number of the serial port where the cable is connected

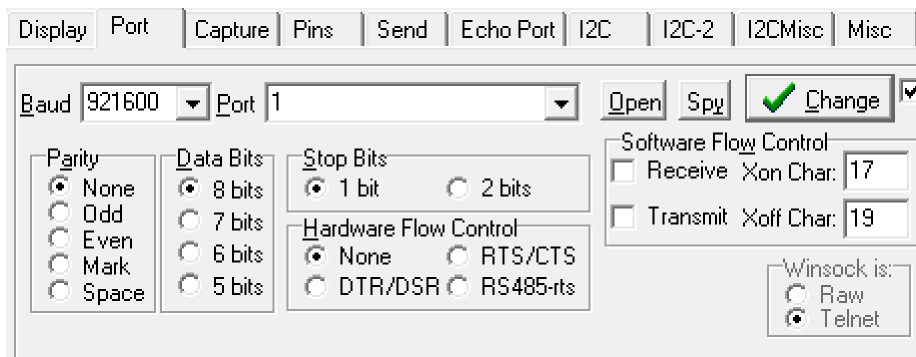

Figure 3.6: Realterm configuration tab.

3.2.4 Operating system

The Operating System (OS) used is the Windows 10. This choice was made by personal choice and experience in the OS, and because the Xilinx development environment ISE was already installed in a PC with this OS. Additionally, Visual Studio 2017 is compatible with Windows 10.

Having described the validation platform, section 3.3 describes the validation flow of the validation platform.

3.3 Validation flow

The validation platform block diagram is presented in Figure 3.7, providing more detail than Figure 3.2. The computer executes two applications, the developed C# program and the Realterm. The FPGA includes:

- the UART modules to receive ("UART_RX") and transmit ("UART_TX") from/to the Realterm;
- the DDR2 memory interface automatically generated by the Memory Interface Generator (MIG) from the IP Core Generator of the Xilinx ISE tool;
- the image processing algorithms to be validated; and
- the main algorithm that controls the entire system, ensuring proper communications, data exchanges, and the execution of the image processing algorithms.

The DDR2 memory, as illustrated in Figure 3.9, is used to store imagens and their features.

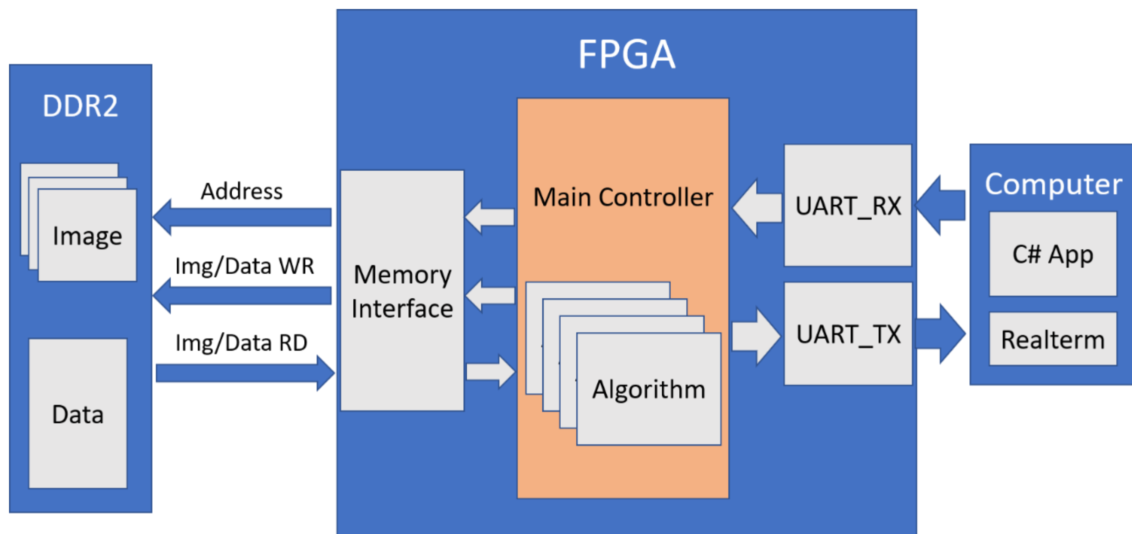

Figure 3.7: Validation platform block diagram.

The validation flow, using the developed validation platform, is presented in Figure 3.8. The flow starts with the user picking an image stored in the computer memory. This image can have several image extensions as .jpg, .png and others. The image must be transferred from the FPGA to the computer via serial port. To do this, a serial terminal program (such as the Realterm) that uses the UART to exchange data between components (in this case the FPGA and the computer) is used. Realterm has the option to send a file. Before sending it, it is required to convert the original image, which is usually a compressed image, to its correspondent bitmap (.bmp), saving the image pixel by pixel in a binary file. After that, the data is not compressed, and the pixels are represented by the RGB colours, each having 24 bits representing them, the FPGA can understand the image and start to process it.

The serial terminal program sends the file with the image to the FPGA, where the data is received and stored in the DDR2 memory. The image processing algorithm can be executed now. From this processing, the images and/or their features are stored in the DDR2. When all the features are obtained, they are sent to the computer. The serial terminal program acts like the interface and receives the image features via UART. The processed images and/or their features are then evaluated by the developed C# program. One of the developed algorithms produces a Gray-Level Co-Occurrence Matrix (GLCM), which is received by the computer and stored in a csv file to be compared with the matrix obtained in the software implementation.

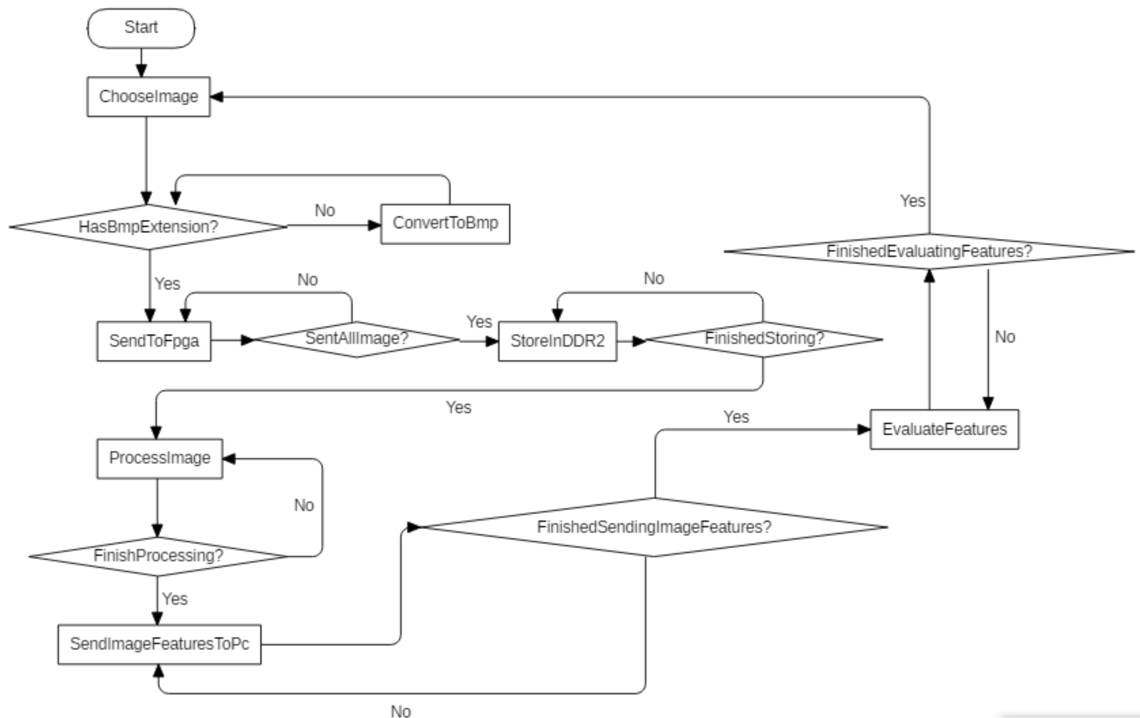

Figure 3. 8: Validation flow.

In the validation flow, the user chooses an image to be processed or to have its features extracted. This pick happens in “ChooseImage” process. The “ConvertToBmp” process is the execution of the C# program to convert the image to its bmp equivalent, if it is not already in it, and store it in a binary file. The process of “SendToFpga” is the transfer of the text file to the FPGA using the serial terminal program as an UI. When all the data from the text file is received, it starts to be stored in the DDR2, which is described by “StoreInDDR2” process. After that, the image process algorithms are executed in order to extract the desired image features. The image features are stored in the DDR2, and when are all determined, the FPGA sends them to the PC (“SendImageFeaturesToPc” process). The image features received in the serial terminal program are then evaluated in the C# program, which is described by the “EvaluateFeatures” process. When all of them were evaluated, the flow resets, allowing the user to choose a new image to extract its features.

The next section describes the main controller and the modules to interface with the computer and DDR2.

3.4 Implemented modules

This section describes, with technical detail, the modules that allow the interaction of the image processing modules with the computer and with the DDR2 memory. These interface modules were developed using manually written VHDL code; however, one of these modules, one that interfaces with DDR2 memory, was generated by the Xilinx CORE Generator tool.

The block diagram of the validation platform modules is displayed in Figure 3.9. In the current section, only modules that do not implement image processing algorithms are described. These modules include the interface with the DDR2, the communication with the computer via UART and a main module to control all the system actions. The blocks that implement image processing algorithms are explained in the next two chapters.

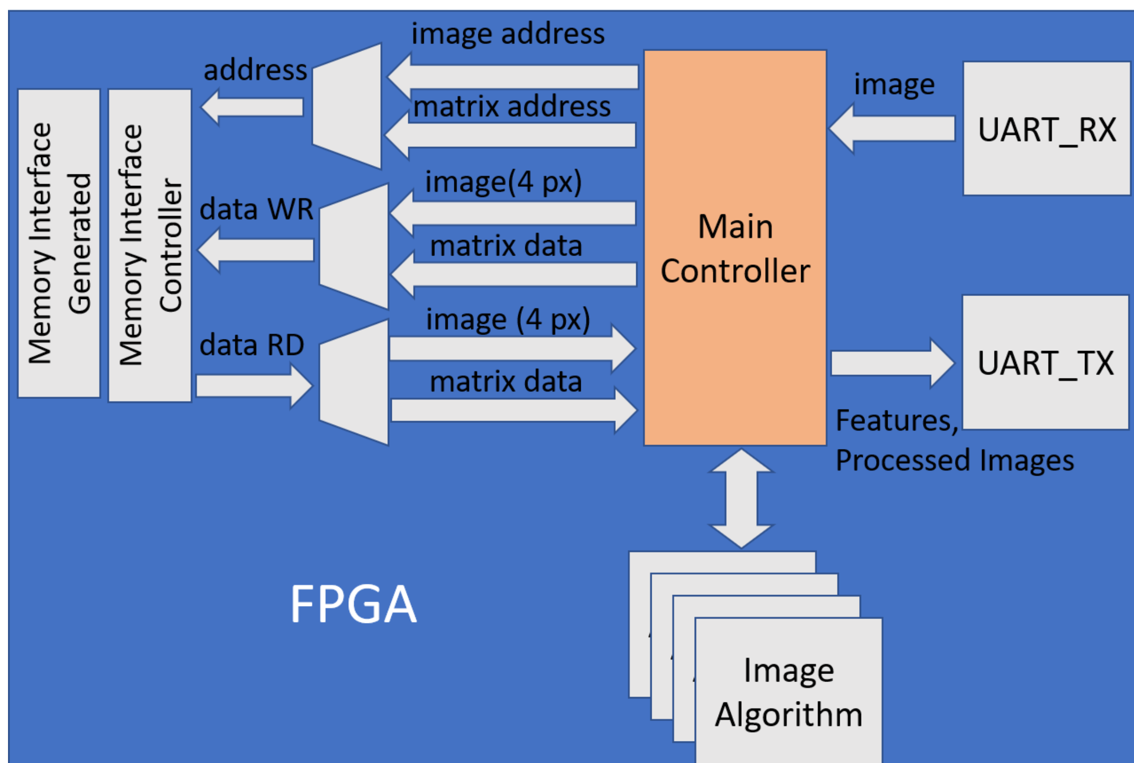

Figure 3.9: Validation platform modules block diagram.

The next sub sections explain the components presented in Figure 3.9. As a note, the "Main Controller" module is called "Control Imig" module. The second name was used for the implementation.

3.4.1 Memory Interface Generated module

This module is an interface with the DDR2 memory, which is used to store the images and their features. It was generated using the Xilinx CORE Generator tool. This tool allows the generation of several modules like dividers, RAM or ROM memories, and more. So, we need to specify what kind of component we want to generate. For the DDR2 interface, we must choose a RAM memory and an interface to interact with it. In the tool, the folder memories & storage elements are chosen and in there is a folder called Memory Interface Generators. This folder and then the option MIG are selected. In Figure 3.10 is displayed a snapshot of the Xilinx CORE Generator tool.

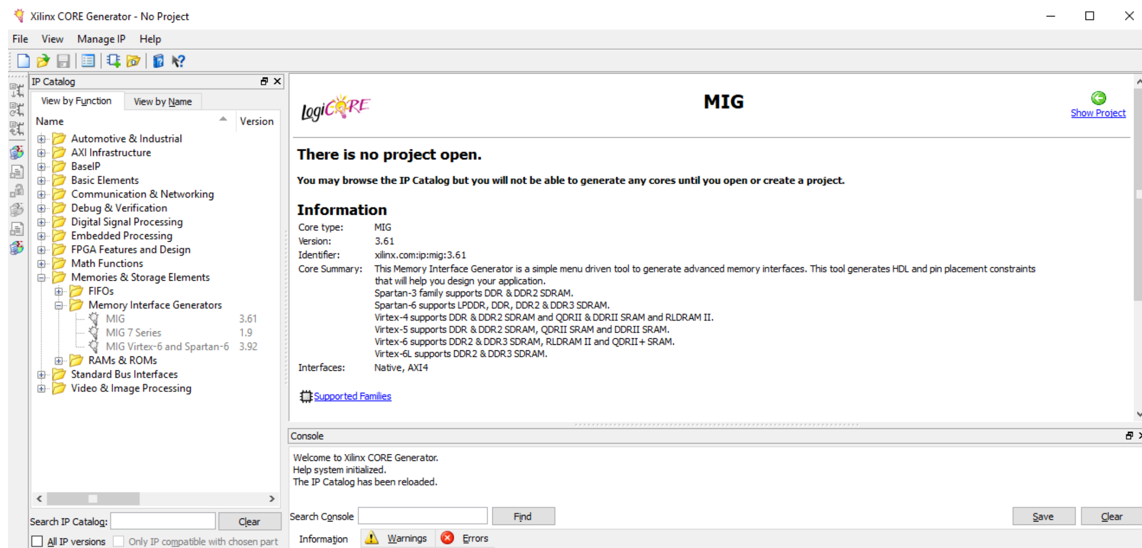

Figure 3. 10: Memory interface generation for DDR2.

To finish the generation of the DDR2 interface, some additional configuration parameters must be provided. In Table 3.2 are displayed the configuration parameters:

Table 3.2: DDR2 interface configuration parameters.

Frequency	8000 ps = 125 MHz
Data Width	32 bits
Burst Length	4

The frequency was set to 125 MHz to match the clock frequency of the FPGA. The data width was set to 32 bits, since two DDR2 of 16 bits are used. The burst length of 4 allows the read and write of 128 bits, which are 4 words of 32 bits.

In order to understand the reason behind this configuration, the behaviour of the DDR2 must be explained. The DDR2 has 3 main actions to be performed, the initialization of itself, the read and the write operation. The initialization is defined by the following Figure 3.11.

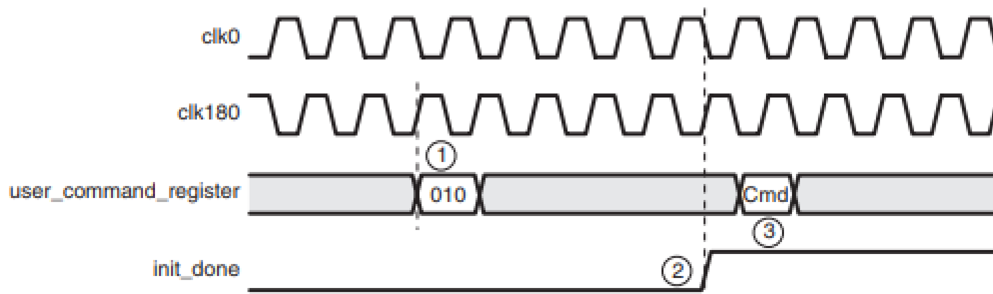

Figure 3.11: DDR2 initialization [46].

For the initialization of the DDR2, the initialization command must be issued in the falling edge of clk0 (1). After a certain number of clocks, the signal “init_done” is high (2), which means the initialization was done successfully (3). Next in the Figure 3.12 is presented the write operation.

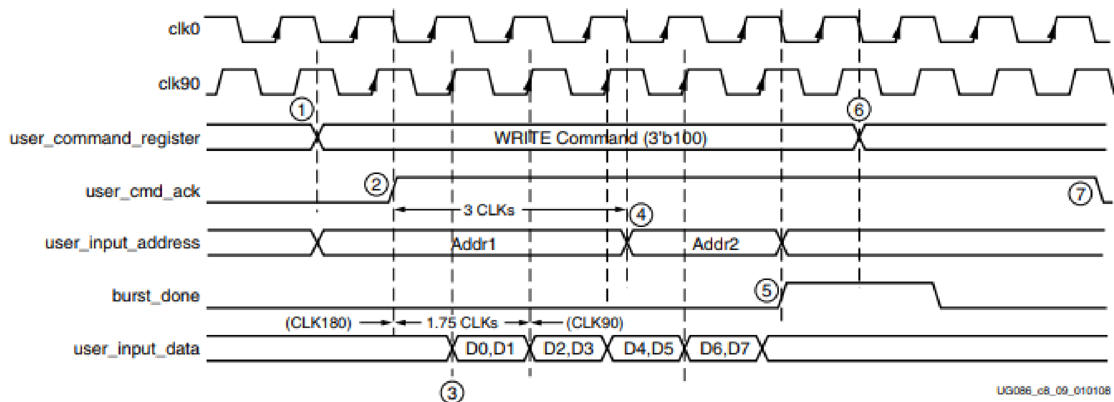

Figure 3.12: DDR2 write operation [46]

The write operation starts when the write command is asserted in the rising edge of the clk90 signal together with the first address (1). The command is acknowledged in the falling edge of the next clock (2). After that, the user must introduce 32 bits of data, in each rising edge of clk90, to be written in the DDR2 (3). 3 clocks after the command acknowledge, the next address must be provided (4). After writing all the words, the “burst_done” signal is asserted for 2 clocks, signalling the end of the operation (5). The command buffer is then cleaned after the first burst done (6). A certain number of

clocks later, the user command acknowledge is set to low, telling the user that another command can be introduced (7).

Next in Figure 3.13 is presented the read operation. The read operation also operates in the falling edge of the clk0. The read command is asserted together with the address (1). 3 clocks after the acknowledge (3) the next address can be set. After the read latency (4), the “user_data_valid” signal is asserted, meaning the data is available to be accessed from the given addresses (5). A word is ready for clock, and the “burst_done” signal is asserted for 2 clocks after the data was signalled as available to be accessed (6). 1 clock after the “burst_done” signal assertion, the “user_command_register” is cleared (7), which means another command can be given, and some clocks later the “user_command_ack” is set low (8), signalling the end of the read operation.

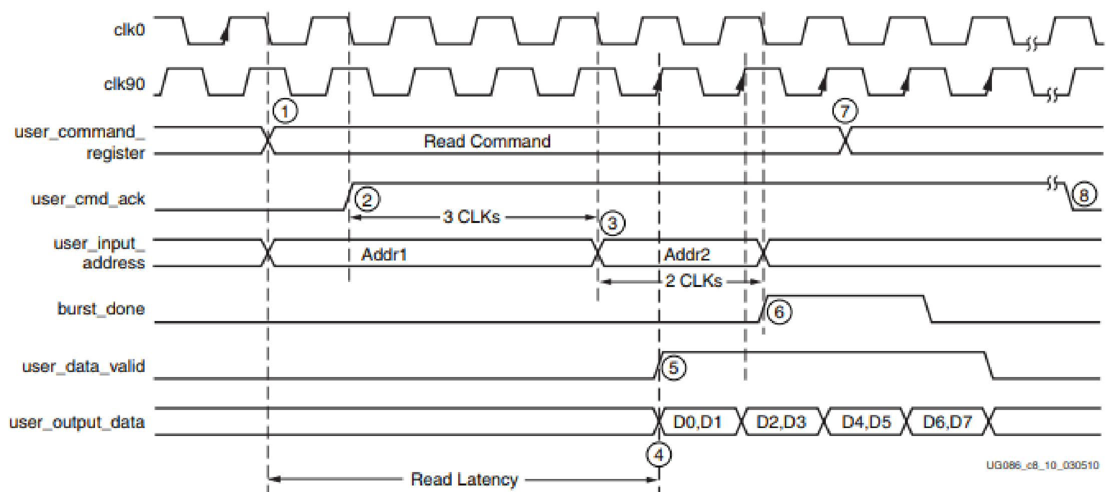

Figure 3.13: DDR2 read operation [46]

The DDR2 memory is used in this validation platform to store images and data about the images. The memory was divided in two areas, one to save the image to be processed or the processed image, and the other area to save data. All the implemented image processing algorithms use the image area; however, just one these algorithms use the data area, to save a matrix with image features. The developed validation platform supports the processing of images with dimensions of 640x480, which means the DDR2 must provide 307200 addresses (represented by 19 bits) in order to store the image, since it stores each pixel at a 32-bit position. The matrix with the image features has a dimension of 256x256, which demands the DDR2 to have 65536 addresses (represented by 16 bits) where the matrix can be stored. Summing the 2 values, the DDR2 must have 372736 memory positions (represented by 19 bits). The DDR2 itself use up to 25 bits to represent memory positions, so the storage of both the matrix and the image can be done.

It is also required to ensure that the storage of the image and the matrix do not override each other, which means, a boundary between them is needed. This is achieved by dividing the DDR2 into 2 sets of addresses, one to store the image and another to store the matrix. Since the image only needs up to 19 bits to be stored, the address for the storing the image starts at 0 and ends up in 16777215 (0111111111111111111111111 in binary or FFFFFF in hexadecimal). The remaining of the addresses are then reserved to the storage of the matrix, which starts at 16 777216 (1000000000000000000000000 in binary or 1000000 in hexadecimal) and ends in 33554431. Since each set of addresses can address up to millions and the image and the matrix only need hundreds of thousands addresses, there is space to have them in the address sets.

The DDR2 interface generation comes up with the discussed options of reading and writing in it. The fact of the amount of memory it has available meets the needs of the validation platform, storing the image and the matrix with the image features without filling up entirely the memory, leaves free space to be used for other algorithms. The requests of reading and writing are documented, but the DDR2 interface module does not perform any kind of control over them, it just executes the requested commands. So, it is needed a module to ensure that the commands are issued correctly. This module is called “Imig”.

3.4.2 Memory Interface Controller Module

The memory interface controller, named “Imig”, is the module that ensures that the init, read and write operations are performed correctly in the DDR2, as described in Figures 3.11, 3.12, and 3.13. The module implements 5 parallel processes, described in Table 3.3:

Table 3.3 Parallel processes in Imig module and their description.

Process	Description
Init Ram	Inits the DDR2.
Write Ram	Write 128 bits in the DDR2 in the given address.
Fill Write buffer	Fill the buffer to be written in the DDR2.
Read Ram	Read 128 bits of the DDR2 in the given address.
Fill Read buffer	Fill the buffer with the information read from the DDR2.

In Figure 3.14 there is the statechart of the Imig module. The first process in Table 3.4 is the Init Ram, which goes to the initial state “InitIdle” when the initial reset occurs. This process as the goal to signal the Init command to the DDR2 interface, to initialize it. If the DDR2 was not initialized yet, the state shifts to “SetInitCmd”. If it was, the state goes to “InitDone”. The state “SetInitCmd” sets the init command and the next state “ClearInitCmd” clears it. Finally, the state shifts to “InitDone” when the initialization of the DDR2 was successful. After initialization, the DDR2 is ready to accept write and read commands.

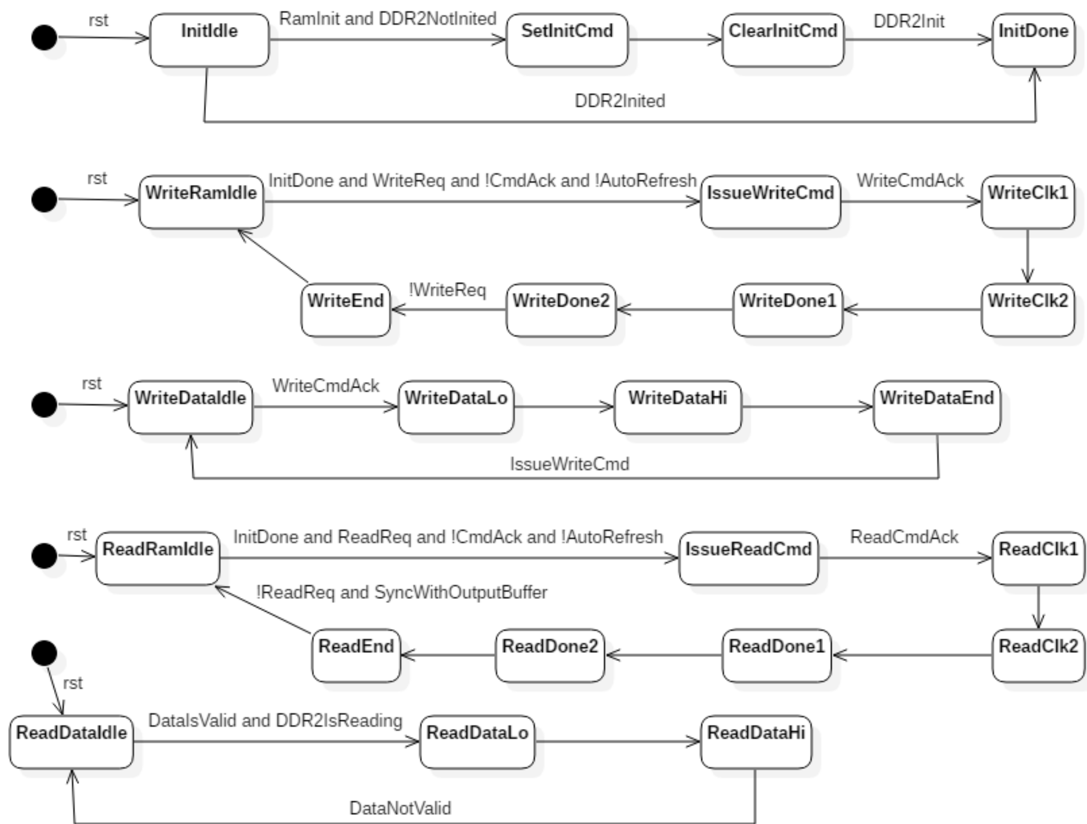

Figure 3.14 : Imig statechart.

The second process in this module is the Write Ram process. This process ensures that a write operation in the DDR2 is executed. When the reset is done, the state shifts to "WriteRamIdle". When a write request is signalled to this module, if the DDR2 was already initialized and no other command is being executed and the DDR2 was refreshed, the state shifts to "IssueWriteCmd". The refresh is the process of periodically reading information from an area of the memory and immediately rewriting that same information in the same area without any modification. If the previous conditions meet, the write operation is accepted by the DDR2, with a certain address. Once the DDR2 acknowledges the write command, the state shifts to "WriteClk1" and from "WriteClk1" to "WriteClk2". These two states are just 2 clocks the Imig module must wait for the DDR2 to end the writing of the 128 bits or 64 bits per clock. The next states are "WriteDone1" and "WriteDone2. In these 2 states, the burst done command is set high and the write command is set low. The burst done command is used to signal the ending of the operation. The burst done then signals the end of the write command, and it needs to be set high for 2 clocks. The next state is "WriteEnd" and is reached when the write request made is not being signalled anymore. This state also signals that the writing operation is no longer happening and another operation can be made. It shifts then to state "WriteRamIdle".

The third process is called Fill Write Buffer. This process is used to fill the buffer with data to be written. When the initial reset is done the state is "WriteDataIdle". When the write command acknowledge is high, the state shifts to "WriteDataLo" state, which fills the 64-bit buffer to be sent to the DDR2 with the 64 less significant bits of the 128 bits message. The state then shifts to "WriteDataHi", which then fills the buffer with the 64 most significant bits. The actual writing of the data is made in the previous process (Write Ram). The next state is "WriteDataEnd". This state waits for the write request ends, and then it shifts back to "WriteDataIdle".

The fourth process is called Read Ram, and, as the name states, it has the job to control the read request operation and ensure it is performed correctly. After the initial reset, the state reached is "ReadRamIdle". When a read request is signalled, the DDR2 has already been initialized, if there is no other command signalled, and the DDR2 is refreshed, the state shifts to "IssueReadCmd" and the read operation starts. The read request to the DDR2 is made and it is signalled that the read operation is being done. The next state is "ReadClk1", which is achieved by being in the previous state and the read command acknowledge being signalled by the DDR2. Like the Write Ram process, the next state is "ReadClk2". After getting them, the 2 next states are "ReadDone1" and "ReadDone2", where the burst done is high to signal the end of the read operation. Then the state switch to "ReadEnd". This state only shifts to "ReadRamIdle" when Control Imig set the read request to low and the state is synced with the state that uses the read buffer. When the buffer connected to ControlImig has all the 128 bits stored, it signals that to this state, and after that a new read request can be made without the problem of losing information in the process.

Finally, the last process of this module is the Fill Read buffer process. Analogous to the Fill Write Buffer, it fills a buffer with the 128 bits read from the DDR2 given a specific address. When the initial reset happens, the state shifts to "ReadDataIdle". If the read operation is happening and the information is ready to be read (is valid, signalled by the DDR2), the last significant 64 bits are stored in the read buffer to be sent to ControlImig. The state also shifts to "ReadDataLo". Now the state goes to "ReadDataHi" and the next 64 bits read are stored in the 64 most significant bits of the buffer. When DDR2 signals that data is no longer valid to be read, the state goes back to "ReadDataIdle" and syncs with the "ReadEnd" state from Read Ram process, so there is not loss of information and a new read request can be made again.

The next sub-section presents the communication between computer and the FPGA, which is made through the UART module.

3.4.3 UART modules

The UART (Universal Asynchronous Receiver-Transmitter) is used for asynchronous serial communication. It sends bytes over a single wire. Because it is asynchronous, it is not controlled by a clock. Can operate in Half-Duplex, which means there at two transmitters sharing a line, or Full-Duplex (two transmitters having their own line). It has several parameters

that can be set by the user, such as the baud rate, number of data bits, parity bits, stop bits and flow control. The baud rate is the rate which the serial data is transmitted in bits per second. The number of data bits is the number of bits to be transmitted. The parity bit can be appended to the data; it is used for error detection. The number of stop bits, which signals the end of operation, can be configured. Flow control is the process of managing the rate of data transmission between two nodes in order to prevent the drowning of a slow receiver by a fast transmitter. All these parameters are configured in the serial terminal program Realterm.

In the developed validation platform, both the computer and the FPGA use UART modules to interact. The FPGA samples the line continuously, and once the line transits from high to low, it means the data is coming from the computer. The first transition indicates the start bit, in a falling edge, and once it is detected, the FPGA must wait for one half of a bit period, which is specified by the baud rate. This ensures the sampling of the middle of the data bit, so no data bits should escape the sampling. The FPGA is sampling until the stop bit is found.

In order to have the UART modules to work properly, the same configurations must be set in the computer serial terminal program (Realterm) and in the FPGA.

3.4.3.1 FPGA UART configuration

The UART must be configured with certain parameters has previously described, as baud rate, number of data bits and more. The baud rate is configured in Realterm, and so it is configured the same value here, 921600. It is also needed to configure the generic clocks per bit = (Frequency of internal Clock)/ (Frequency of UART). The generic clocks per bit syncs the UART with the other modules that share DDR2's clock, so the sampling of the line can be done properly by the FPGA. Since the clock used is the clock of the DDR2 (125MHZ) is the internal clock, and the frequency of the UART is 921600 (the baud rate), the Generic clock configured in the UART receiver and transmitter is 136. The configured value is the generic clocks per bit and is 136.

The FPGA UART module was divided in 2, the UART receiver, which handles the transfer of data form the PC to the FPGA and the UART transmitter, which handles the communication in the opposite way. The next sub section discusses the UART receiver implementation.

3.4.3.2 UART receiver

The UART receiver module receives each byte of the image and make it available to the FPGA. One of the main requirements for the UART receiver is to receive the entire image with the fastest baud rate possible, so the processing operation can also be faster. In Figure 3.15 is a statechart of the UART receiver behaviour.

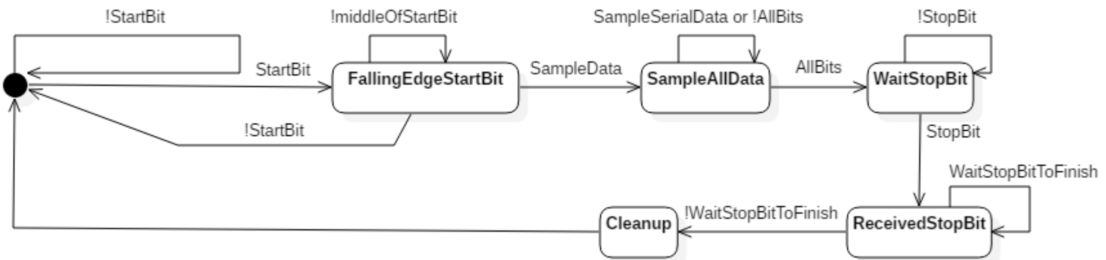

Figure 3.15: UART receiver statechart.

To summarize, the UART receiver stays in its initial state if hasn't detected the start bit. When it does, the initial state shifts to state "FallingEdgeStartBit". In this state, the FPGA waits for one half of a bit period, which depends on the baud rate chosen. This verification ensures that the middle of the data bit gets sampled. After this sample, the state changes to "SampleAllData". In this state, the FPGA wait for one-bit period (to ensure it received all the information), that depends on the baud rate, and then samples the 8 bits of data. When it is done, the state shifts to "WaitStopBit". This means that the FPGA will wait for the stop bit to show up, signalling that all the data was sampled, and the operation has ended. It stays in "ReceivedStopBit" state to ensure stop bit has ended. The next state shift is to state "Cleanup" where the states and signals used are reset. After that, the state shifts to the initial state, allowing new data to be sampled.

As stated in section 3.2, the user interface used for the UART is the Realterm. It has the options to set the baud rate, set the display format of the information that is being sent or received and allows to send all the information of a file trough the UART to the FPGA. And that's exactly what is being done, all the information of the 640x480 image, which is saved in a file, is sent through the UART.

Although this implementation allows the FPGA to receive data, it is not quite compatible with the amount of data the DDR2 allows to be written, since this module allows receiving 8 bits at a time and the DDR2 allows writing and reading 128 bits. To optimize this process, a new component, UART receiver aux 128 bits, was built. Its implementation is described in the next sub section.

3.4.3.3 UART receiver aux 128 bits

Since the DDR2 interface, as described above, writes data in bursts of 128 bits, it is required to have a module that get this amount of information and sends it to the DDR2. The module that does this is the UART receiver aux 128 bits. It is constantly waiting for the UART receiver to send 8 bits until it has a 128 bits buffer full. After this, it sends the information to be written in the DDR2 and keep waiting for more. In Figure 3.16 is the respective statechart of this module.

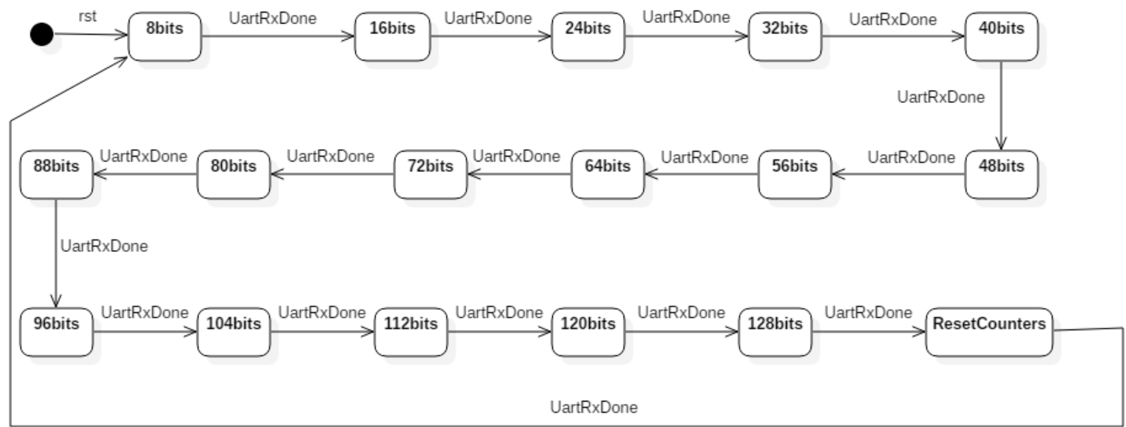

Figure 3.16: UART receiver aux 128 bits state chart.

The statechart shifts to state “8bits” when reset is triggered. Any time UART receives a byte (UartRxDone), the state shifts. The “8bits” state shifts to the “16bits”, the “16bits” state to the “24bits” and so on. As the name of the states indicate, the “8bits” state means the first 8 bits were stored in the 128bits buffer, the “16bits” state means that we have already stored 16 bits and so on. This goes on until the “128bits” state is reached, that means the buffer is full and the information can be sent to the DDR2. So, the “128bits” state shifts to the “ResetCounters” state, and as the name states, it resets the counters needed to have this module prepared to receive another 128 bits from the UART receiver.

The next sub section explains the implementation of the UART transmitter, which is like the implementation described so far.

3.4.3.4 UART transmitter

The UART transmitter module has the goal of sending the processed image and/or their features from the FPGA to the PC. Its behaviour is analogous to UART receiver, having a start bit to signal the beginning of the transmission of 8bits and a stop bit signalling the end. In Figure 3.17 can be found its statechart.

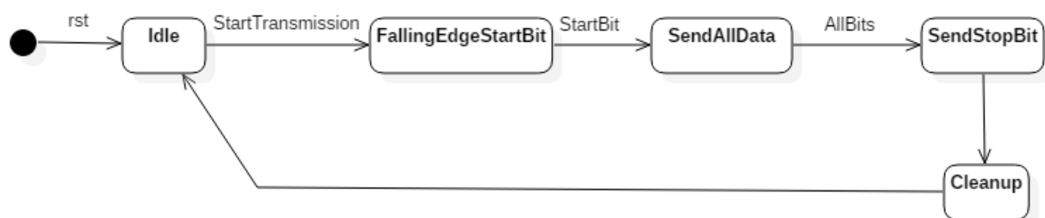

Figure 3.17: UART transmitter statechart.

The UART transmitter stays in its initial state if it has not been commanded to start. When it is, the initial state shifts to state “FallingEdgeStartBit”. In this state, the FPGA sets the transmission line (TX) to zero to send the start bit. After, the state changes to “SendAllData”,

where the FPGA sends the 8 bits with a frequency that depends on the baud rate. When it's done, the state shifts to "SendStopBit". The FPGA will wait for the stop bit to be sent, and the operation has ended. The next state shift is to state "Cleanup" where the states and signals used are reset. After that, the state shifts to the initial state, allowing new data to be sent.

As described, the UART transmitter behaves like UART receiver. The main goal here is to send the new processed image to the PC via Realterm.

Although this implementation allows the FPGA to transmit data, it is not suited with the amount of data the DDR2 allows to be read, since this module allows sending 8 bits at a time and the DDR2 allows reading 128 bits. To optimize this process, a new component, UART transmitter aux 128 bits, was built. Its implementation is described in the sub section 3.4.5.3.6.

3.4.3.5 UART transmitter aux 128 bits

Since the information to be processed is stored in the DDR2, read, processed and stored again in it we need an auxiliary module to send this information the UART transmitter. We cannot send to the UART transmitter directly because the DDR2 gives 128 bits of information and the UART transmitter only accepts 8 bits per clock. So, the UART transmitter aux 128 bits module connects these two by storing 128 bits of information and send 8 bits at a time to the UART transmitter. In Figure 3.18 is the respective state chart of this module.

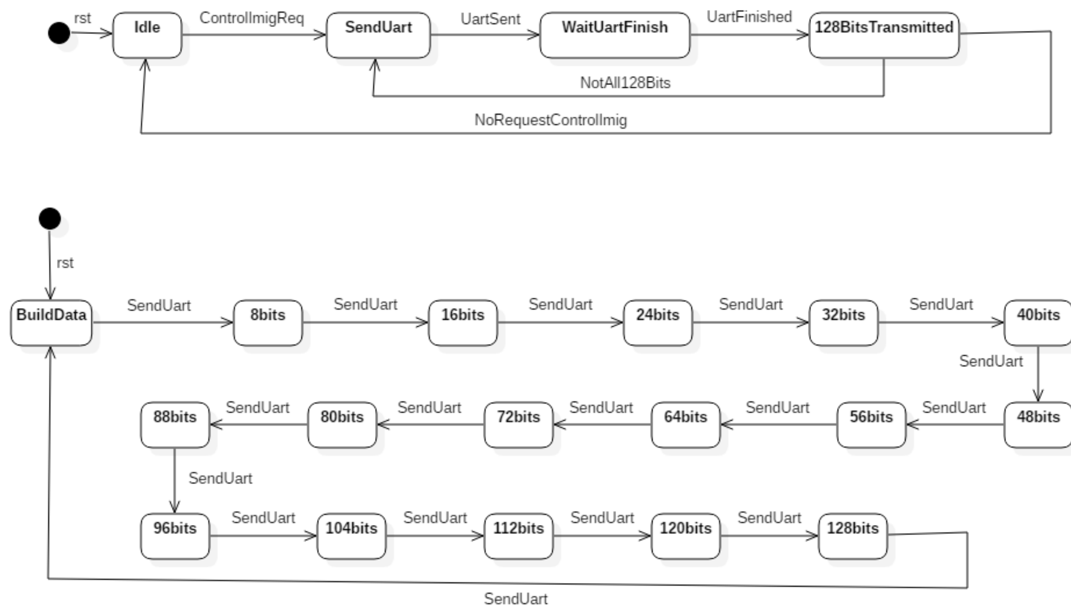

Figure 3.18: UART transmitter aux 128 bits state chart.

The initial state is the state where the module is prepared to receive requests to transmit. This state is called “Idle”. It expects request because it should only send information when it is ready, and since all modules are controlled by the module ControlImig it is going to be controlled by it as well. When ControlImig sends a request, the state shifts to state “SendUart” and it is sent an acknowledge to ControlImig signalling that the request has been accepted. This state has a sub state machine associated with it, since it needs to send 8 bits at a time to the UART transmitter. The first state of this sub state machine is called “BuildData” and is reached when reset is on. This sub state machine is triggered whenever the “SendUart” state is reached, making it go through sub state “8bits” to “128bits” state, and ending in the “BuildData” state. The analogue behaviour is found in the UART aux receiver 128 bits module.

Whenever the UART transmits 8 bits, the state shifts to “WaitUartFinish”, where it requests the UART transmitter to not transmit anymore until a new request is made. The state shifts to “128BitsTransmitted” when UART transmitter signals it has ended the transmission operation. In this state it is evaluated if the 128bits were transmitted or not. If not, the state shifts back to state “SendUart” in order to keep sending another 8 bits till it reaches the 128bits sent. If yes, the state shifts to “Idle” when ControlImig is not making any request to this module. Essentially this flow is repeated each time the ControlImig makes a request.

In the next sub section, the module that controls the general flow of the validation platform is presented, the Main controller module.

3.4.4 Main controller

The global system, which includes the validation platform and the image processing algorithms, is composed by several modules, as presented in Figure 3.7. To manage these modules, a main controller, named here as “ControlImig”, was implemented.

The interaction between the Control Imig and the other modules is mainly supported by a 4-phase handshake protocol. This protocol is not related with message communication, but it is related with synchronization between the modules. It is a 4-phase handshake protocol that uses pairs of Request/Acknowledge signals [47]. Using this protocol, the transmitter component makes a request to another component, setting the Request line. The receiver (other component) receives the request and sends back an acknowledge (setting the Acknowledge line). When the acknowledge is received, the transmitter component stops making the request, resetting the Request. But it waits until the receiver component ends, which is signalled with the Acknowledge reset. So, this way the components are synched with each other and the transmitter only makes another request after the receiver ends.

This protocol is used to control/synchronize operations such as read from the DDR2, write, apply image processing, control how information comes and goes to UART, and more. It has control over all components and communicates with them by making requests and expecting

acknowledges to those requests. In Table 3.4 is shown the main actions controller by the ControlImig flow.

Table 3.4: Main operations controlled by ControlImig.

Name	Description
Initialize DDR2 positions	Initializes the DDR2 positions value with 0.
Receive image from UART	The PC sends and image via UART to the FPGA.
Write image in DDR2	An image is stored in the DDR2.
Read image from the DDR2	Retrieves an image form the DDR2.
Algorithms	Applies the algorithms to the image stored in the DDR2.
Read data from DDR2 to UART	Reads data from the DDR2 and sends it to the PC via UART.

ControlImig has 3 main processes running simultaneously. It has the Main process, where all the requests for certain operations are made, the Write process, where all the write operations are controlled, and the Read process, where all the read operations are controlled.

The first operation controlled by Control Imig is the “Initialize DDR2 positions”. It consists in initializing the DDR2. The DDR2 initialization command is issued, but its memory positions come with a default value different from zero. Since the image and the data are stored in the DDR2, those positions should to be initialized to zero to ensure data integrity.

The second operation that the Control Imig is responsible for, is “Receive image from UART”. Having the DDR2 initialized and with its memory position values cleared, the system is ready to receive an image. The transmission of the image to the FPGA is made through data exchange through the UART.

The next operation (“Write image in DDR2”) consists in writing the image data in the DDR2. Since the DDR2 allows writing a maximum 128 bits in a write operation, 128 bits of data image are fetched and stored.

Control Imig also controls the “Read image from the DDR2” operation, which fetches 128 bits per read operation from the DDR2. Whenever image data is read, Control Imig executes the operation “Algorithms”. This operation consists in applying the developed image processing algorithms and store their result in the DDR2. One of those algorithms, which saves a matrix in the DDR2, is described in detail on Chapter 4.

The last operation Control Imig performs is to send the stored information in the DDR2 back to the computer. This is achievable trough the UART module.

3.4.5 Main File

The Main file instantiate all the modules and is responsible to interconnect their inputs and outputs and the global system inputs and outputs. It is implemented as the top module of the Xilinx project. The connections between modules could be done via schematic, but since the quantity of modules and connections are quite big, it was chosen to do it in VHDL. The connections are made by defining each module with its own port map, which maps signals in an architecture to ports on an instance within that architecture. To deploy the code to the FPGA, Xilinx ISE uses the tool Impact to load a .bit file to the FPGA. Since the bit file generated is the top module file, the file loaded into the FPGA is the main.bit file. The next sub section explains what the UCF file is and how it relates to the main file.

3.4.6 UCF File

The User Constraints File (UCF) file is dedicated to connecting the inputs and outputs to the FPGA pins, to constrain clocks, paths, and more. The inputs connected to the board push buttons were 3, an input to make the initial reset, an input to init the DDR2 and an input to start the program flow.

The next section provides a detailed description about the application developed in C# to perform the pre-processing task and to compare the output of the FPGA with the initial input.

3.5 Pre-processing and evaluation application

To get the image ready to be sent from the PC to the FPGA and to evaluate the extracted features and/or processed images, a C# program was developed in Visual Studio 2017. The flow of getting the image from the PC into the FPGA goes through some main steps. The first one is the conversion of the image to its correspondent bitmap (BMP) with 24 bits per RGB pixel (8 bits per colour), since it has an extension .jpg, .png, or other, which means the data is compressed and the FPGA cannot decompress it, without specific IP (Intellectual Property) cores.

The second step is more related to implementation details. The validation platform was developed to read and write, in each memory access, 128 bits in the DDR2. To make it simple, the developed platform only saves 4 RGB pixels in each 128 bits (1 RGB pixel in each 32 bits). So, the C# program includes one byte (the ASCII code of the char "0") after every 3 bytes of the image. Nevertheless, this fourth byte is ignored by the image processing algorithms. Additionally, because the first 54 bytes of the BMP file are the header, not containing image pixels, they are not copied to the new binary file by the C# program. The file is then closed, and it is ready to be sent to the FPGA via Realterm. When the processed images and/or their

features are sent from the FPGA to the Realterm, the Realterm saves them into a file with extension “txt”.

The developed C# program also supports file comparison. This is useful to compare images processed in the FPGA-based platform with the images processed in software using the OpenCV image processing library, enabling the validation of the image processing algorithms implemented in FPGA.

Additionally, the C# application provides the option to copy image features into a CSV file, which is referred in this document as “MatrixToCSV”. This option is required to validate an implemented algorithm that calculates the Gray-Level Co-Occurrence Matrix (GLCM) [48]of an image. In Figure 3.19 there is the state chart of this C# program.

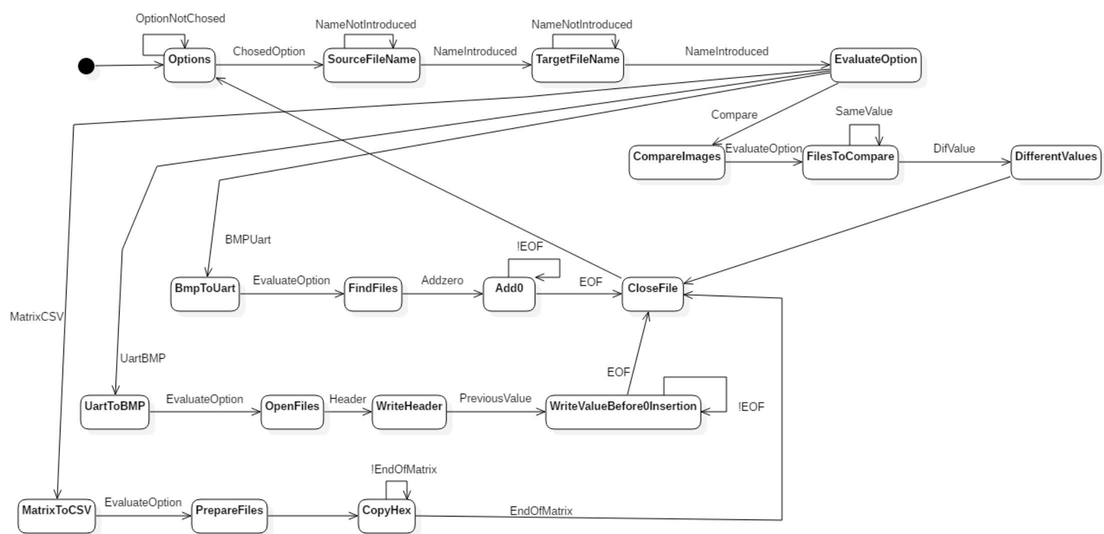

Figure 3.19: C# program state chart.

The program starts in the initial state, which makes a shift to “Options” state. While an option is not chosen the state doesn’t shift. If it is, the state can shift to 2 out of 5 states. The first state is achieved when “Options” shifts to “SourceFileName”, which happens when any option is chosen. In this state, the user introduces the file name from where it is going to copy or change its content. When it is introduced, the target filename is requested. So, the state shifts to “TargetFileName”. When the name is introduced, it shifts to “EvaluateOptions”. This state allows any option to be executed.

Now there are 4 states to choose from according to the option the user chose. If selected the option to add the zeros every 3 bytes, the state goes to “BmpToUart”. Here it shifts to “FindFiles” where the source file and the target file are searched according to the given names and opened in binary reader (source file) and in normal file mode (target file). If the target file already exists, it is replaced by a new one with the same name. The state than goes to state “Add0”, where it is evaluated in a cycle every multiple of 3. After each multiple of 3, a new

position with the value 0 is introduced. When the end of file (EOF) is reached, the files are closed the and the state returns to “Options”.

If the user chose the option to convert the received file from the UART to Bmp, the “Options” state shifts to “UartToBMP”. Here the flow is similar, but the headers must be added so the original file remains the same. The state shifts to “OpenFiles” and then to “WriteHeader” to effectively write the headers in the opened file. We also should ignore the zero insertion since it wasn’t in the initial file in the beginning. The state shifts to “Ignore0Insertion3 and when the EOF is reached it shifts back to “CloseFile” to close all the files and after that to “Options”.

Another option available is to compare the 2 images. When the user selects it, the state shifts from “EvaluateFiles” to “CompareImages”. Here the 2 files are opened, which makes the state go to “FilesToCompare”, and then in this state the comparison is made byte to byte. If the values are different, a counter is incremented and printed in the console. When it terminates, it switches to “CloseFile” state, and comes back to “Options”.

The final option is to convert matrix to csv. This option makes the state “Options” to reach “MatrixToCsv”. Here, a CSV file is also opened, besides the source file, and every value of the matrix (image features) is converted to hexadecimal and copied to the CSV file. The state shifts to “PrepareFiles” and then to “CopyHex”. When the matrix is all copied the state goes to “CloseFile” and the file are closed. Finally goes back to “Options”.

The use of the C# program to prepare the image to be sent from the PC to the FPGA and to evaluate and store in files the image features is really handy and it does not consume FPGA resources, which allows the FPGA to apply the image processing algorithms without the need to make extra verifications and changes like the addition of a new position with value 0 after very multiple of 3.

The Visual Studio 2017 is the editor chosen to develop the C# application. It also allows the use of the Emgu CV framework, which grants the project image processing functions from OpenCV to be called in the project. Emgu CV is a cross platform .NET wrapper for the OpenCV image processing library. The wrapper is compiled by other programs, which allows calling OpenCV functions from languages compatible to .Net, like C#. A snapshot is displayed in Figure 3.20.

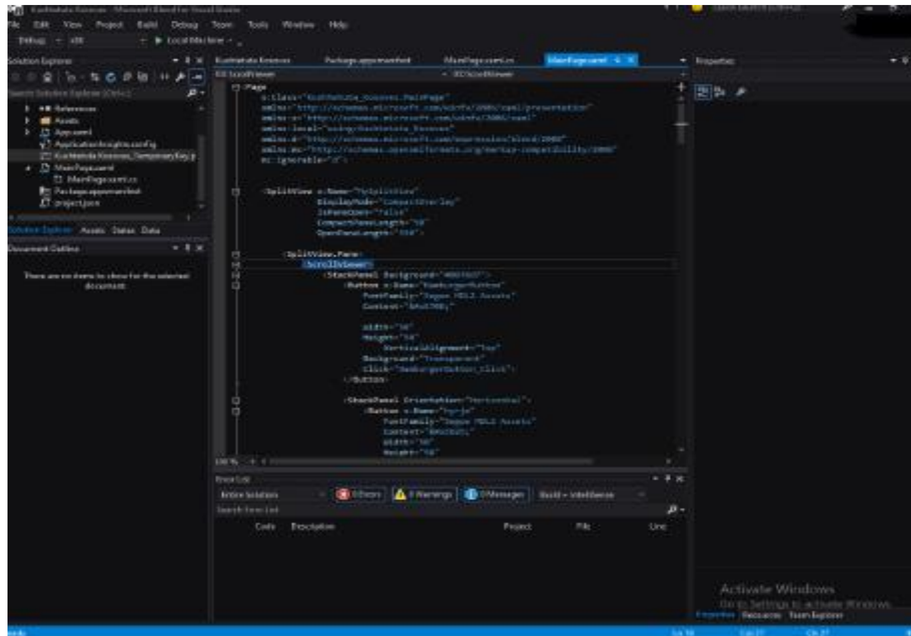

Figure 3.20: Visual Studio 2017.

4 GLCM algorithm module

4.1 Introduction

During this work, it was developed an FPGA-based image processing system that generates the Gray-Level Co-Occurrence Matrix (GLCM) [48] and calculates a pair of image features. The GLCM algorithm is a static texture algorithm, for image classification, that is used to generate the Gray-Level Co-Occurrence Matrix of an image and extract its features, such as the image contrast and energy.

In this image processing system, the algorithms were specified in VHDL and implemented in the validation platform presented in Chapter 3. This system structure and behavior are described in the current chapter. This image processing algorithm starts after the image has been received through the UART module and saved in the DDR2. The algorithm performs a set of tasks, but the task implemented in hardware are summarized in Table 4.1. The grayscale algorithm is applied to the image to get the different grey tones it has. After that, the quantization of grey tones is made. This process has a lot of operations to perform that are computationally heavy. A matrix is made to store the results of these calculations. Next is made the sum of the matrix with its transpose, so the discrepancy between the discrete values is minimized. Finally, the contrast and energy are calculated.

Table 4.1: Implemented GLCM algorithm equations.

Name	Equation
Grayscale	Gray = 0.299×Red + 0.587×Green + 0.114×Blue
Quantization of grey tones	Address = [gray_pixel_x][gray_pixel_y] Value = Value + 1
Sum the matrix with its transpose	Result = M + M ^T
Contrast	Result += |i-j||j-i|(i,j)/Total
Energy	Result += ((j,i)/Total)((j,i)/Total)

The structure of the system that generates the Gray-Level Co-Occurrence Matrix (GLCM) is described in section 4.2 and its behaviour is described in section 4.4.

4.2 System structure

The block diagram of the GLCM algorithm module is presented in Figure 4.1. This module receives 4 pixels at a time and calculates the grayscale of these 4 pixels.

To perform the rest of the operations of the GLCM algorithm, the 4 pixels value are concatenated. This means the concatenation is made with one-pixel data and the next pixel data, belonging to the same line. These generates two matrix addresses if x and y are different. The first is the appending of y to x and a second address with x being appended to y. If x and y are equal, only one address is generated. This will return to the algorithm module the matrix data. The sum of the matrix data with its transpose is performed, incrementing twice if the x and y are equal, once otherwise.

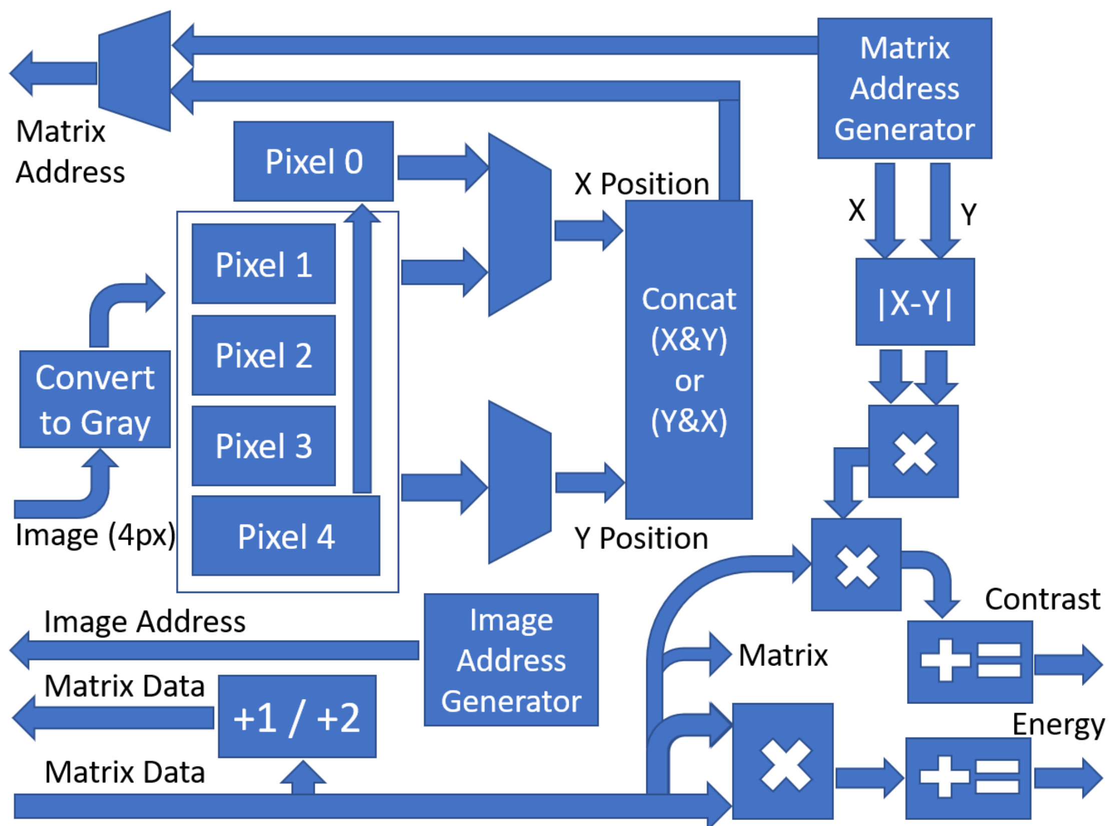

Figure 4.1: GLCM Algorithm block diagram.

The result stored in the DDR2 will be the sum of the matrix with its transpose. The transpose of a matrix is achieved by shifting the columns into lines and lines into columns. So, in the implementation, every time the column is equal to the line, the value of the matrix is incremented since it is the same as the original matrix. The other values are just shifted as stated.

Next, the image contrast and energy are simultaneously calculated. For each matrix position, the absolute value, for the difference between x and y, is calculated. The result is multiplied by itself and by the value of the position (x,y) of the matrix. The result is then stored in the contrast register, which will have in the end the accumulated contrast value of the matrix. For the energy operation, the value in the position (x,y) of the matrix is multiplied by itself and accumulated in the energy register. The energy is calculated by dividing the square of the position in the line and column by the total number of the matrix pixels square.

4.3 Convert to Gray module

The Convert to Gray module from Figure 4.1 contains four instances of the module presented in Figure 4.2. Each instance processes one pixel. This module calculates the weighted average of the three colour components (Red, Green, and Blue):

$$Gray = 0.299 \times Red + 0.587 \times Green + 0.114 \times Blue$$

The previous equation is implemented by OpenCV, which rounds the final result. To do it in VHDL (hardware), the numerator is summed with 512, and the result and the denominator are multiplied by 1024, obtaining the following equation:

$$Gray = (Red \times 306 + Green \times 601 + Blue \times 117 + 512) / 1024$$

Multipliers were user to perform the multiplications, adders to sum, and shifts to make the division by 1024 (2^10).

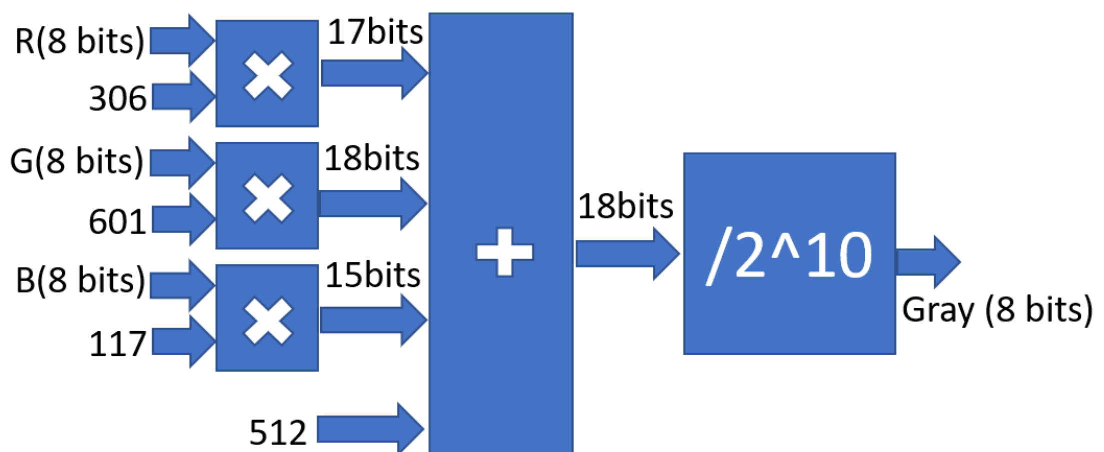

Figure 4.2: Grayscale module.

4.4 System behaviour

The behaviour of the GLCM algorithm module is represented in Figure 4.3 by an UML Activity Diagram. This module implements the main controller ("ControlImig"), described in subsection 3.4.4, extended for the GLCM algorithm.

For the GLCM algorithm to be applied, the system waits for the command Start algorithm. When the command is received, the matrix positions in the DDR2 need to be clear. To ensure data integrity all positions will have zero as value. The same operation is not performed for the image address subset because the image that is going to be received will overwrite the previous values.

The system is now ready to receive an image. The image is transferred from the computer to the FPGA via UART and stored in the reserved address subset from the DDR2. After that, 4 pixels are read and the GLCM algorithm starts to be executed with the application of the convert to gray algorithm. Then, if the pixels belong to a new line, this cycle will happen 3 times, with 3 concatenated values (Pixel1 & Pixel2, Pixel2 & Pixel3, and Pixel 3 & Pixel4), four otherwise (also with value Pixel0 & Pixel1). These values are used to build the matrix addresses (x & y) from where the matrix data is going to be read.

After reading the first matrix data, if the values x and y, used to build the address, are equal, the matrix data is incremented twice, else one time. These increments are needed in order to calculate the sum between the matrix and its transpose. After these increments, it is requested to write that value in the DDR2 in the same position it was read from.

The next condition to be verified is if the x and y are different. If yes, the address is built using x and y in the inverted order. The second matrix data is read, incremented once and stored in the DDR2. After that, an evaluation is made to confirm if all the cycles were processed.

If all cycles were not processed, the next two addresses are built. Otherwise, the fourth pixel is saved for the next address building (Pixel0 & Pixel1). If there are unprocessed pixels, the system will return to read the next 4 pixels. When all pixels are processed, the matrix is ready, and the energy and contrast can be calculated in parallel. To do it, the first position of the matrix is read, energy and contrast values are calculated and saved in the registers. It if is not the end of the matrix, the program returns to read the next matrix position. If it is the end, it terminates.

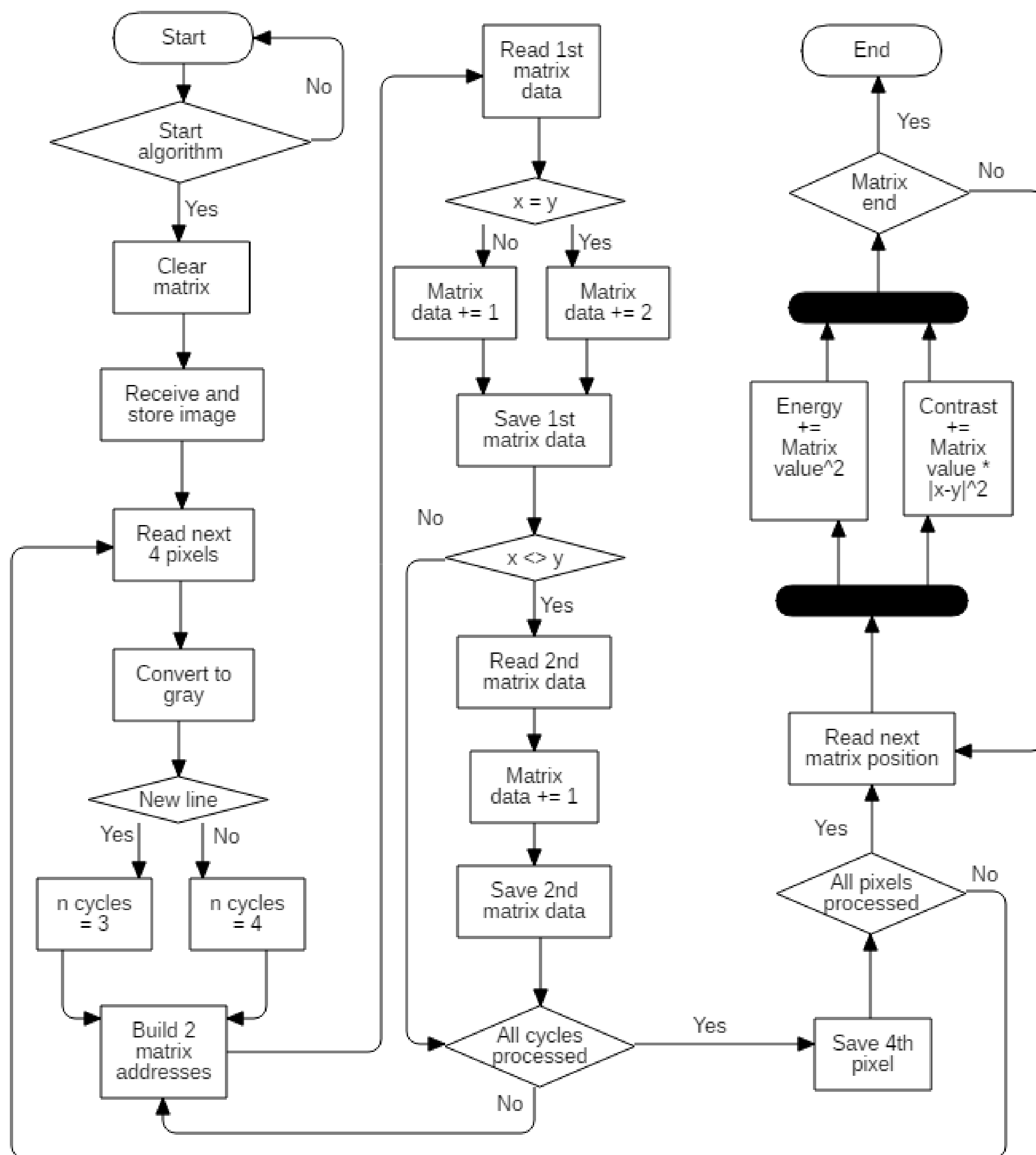

Figure 4.3: GLCM module activity diagram.

5 DS-Pnets and IOPT Flow in image processing

5.1 Introduction

This section proposes a set of image processing modules/components developed in IOPT-Flow. These modules were specified in DS-Pnets, a graphical and intuitive modelling formalism that combines Petri nets and dataflows.

The DS-Pnet models, created with the IOPT-Flow editor, can be validated with the IOPT-Flow simulation tools, and translated into VHDL code, using the IOPT-Flow automatic code generators. The proposed models were used to create a library folder for image processing in the IOPT-Flow framework, available online at *http://gres.uninova.pt/iopt-flow/*. The next sections present the developed modules using the IOPT-Flow tools in order to validate the use of the image processing platform for image processing

5.2 Green channel and Red channel

The red channel and green channel are algorithms that force the output to be the value of one colour. For the red channel, the output will be the colour red, for the green channel colour green. The DS-Pnets models of these modules are presented in Figures 5.1 and 5.2, respectively. Figure 5.2 presents the IOPT-Flow editor with the model in the centre, the icons area at the left, the properties area at the right, and the expression window at the bottom. In the expression window is presented the expression of the DS-Pnet operation, where the output is equal to the Green input.

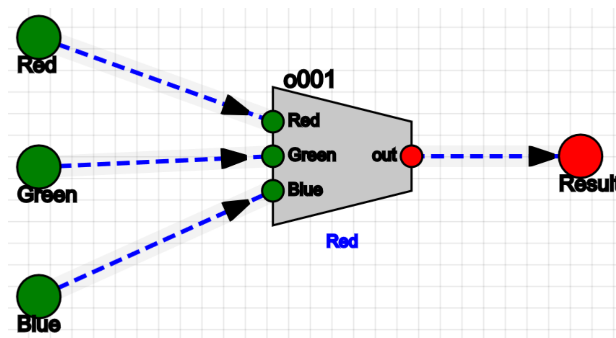

Figure 5.1: Red channel component.

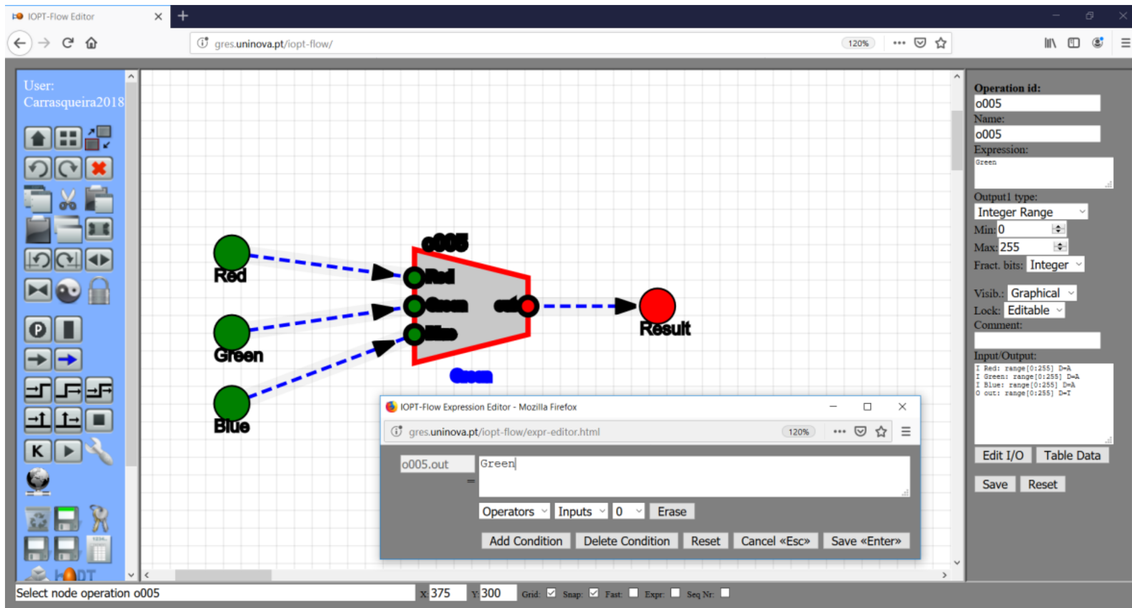

Figure 5.2: Green channel component in the IOPT-Flow editor.

In order to validate components behaviour, the IOPT-Flow tools provide a simulation tool. This simulation takes the green channel component (as an example) and accepts dynamic inputs. It also has the possibility of running individual steps or a continuous run. Individual steps can be used to return to the previous system state or to the next, which eases debug. A simulation step of the green channel component using the IOPT-Flow tools is presented in Figure 5.3. Like Xilinx ISE simulation tool, IOPT-Flow simulator also provides a timing diagram, as displayed in Figure 5.4.

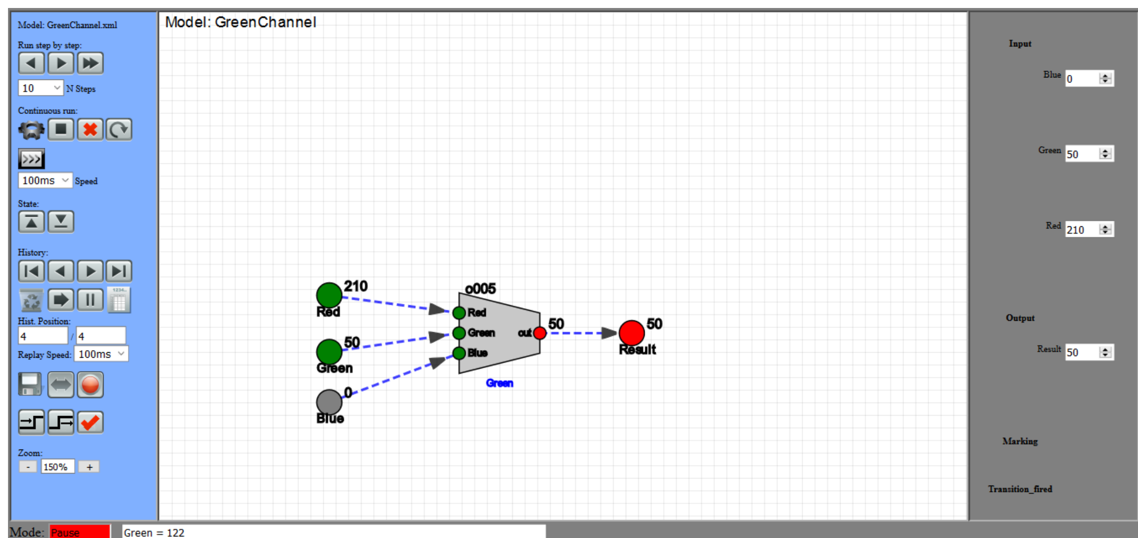

Figure 5.3: Green channel simulation.

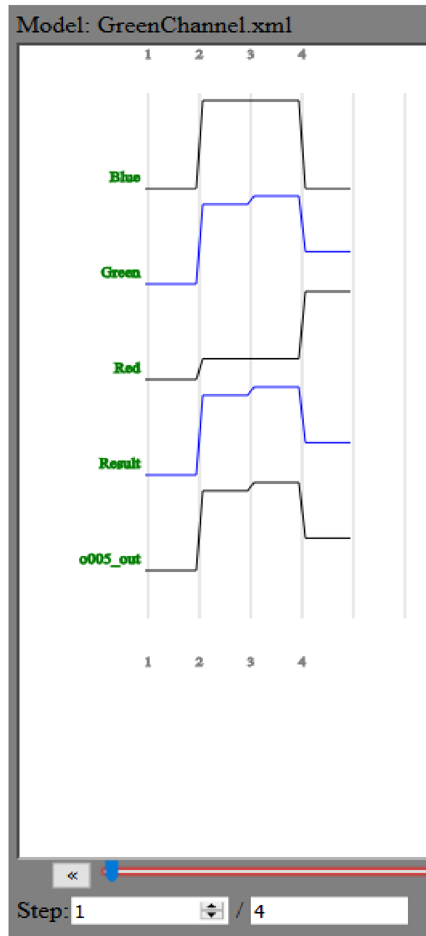

Figure 5.4: Green channel timing diagram.

The IOPT-Flow tools automatically generate VHDL code from the developed DS-Pnet model. For the green channel, the generated code is displayed in Figure 5.5. Some generated code is omitted. The presented VHDL code include the entity port with the component inputs and outputs and the process that makes the result equal to the green.

```
1  -- code omitted...
2  Entity GreenChannel Is
3  Port(
4      PF_Clk: IN STD_LOGIC;
5      PF_Rst: IN STD_LOGIC;
6      PF_Enb: IN STD_LOGIC;
7  
8      Blue: IN INTEGER RANGE 0 TO 255;
9      Green: IN INTEGER RANGE 0 TO 255;
10     Red: IN INTEGER RANGE 0 TO 255;
11     Result: OUT INTEGER RANGE 0 TO 255
12 );
13 End GreenChannel;
14 
15 Architecture Structural Of GreenChannel Is
16     -- code omitted...
17 Begin
18     -- code omitted...
19     ExecProc: Process(PF_Clk,sPF_Rst,sPF_Enb,s_Blue,s_Green,s_Red
20     Begin
21         If sPF_Rst = '1' Then
22             s_Result <= 0;
23         ElsIf sPF_Enb = '1' Then
24             s_o005_out <= s_Green;
25             s_Result <= s_o005_out;
26         End If;
27     End Process;
28     -- code omitted...
29 End Structural;
```

Figure 5.5: Green channel generated VHDL code.

5.3 Negative

The negative algorithm receives, as an input, a pixel value and a reference. The subtraction between each component of the pixel and the reference is made, ensuring the result is positive. If the reference if higher than the pixel component, the subtraction is made using the reference as the first operand; the pixel component otherwise. The implemented module in the IOPT-Flow tools is presented in Figure 5.6.

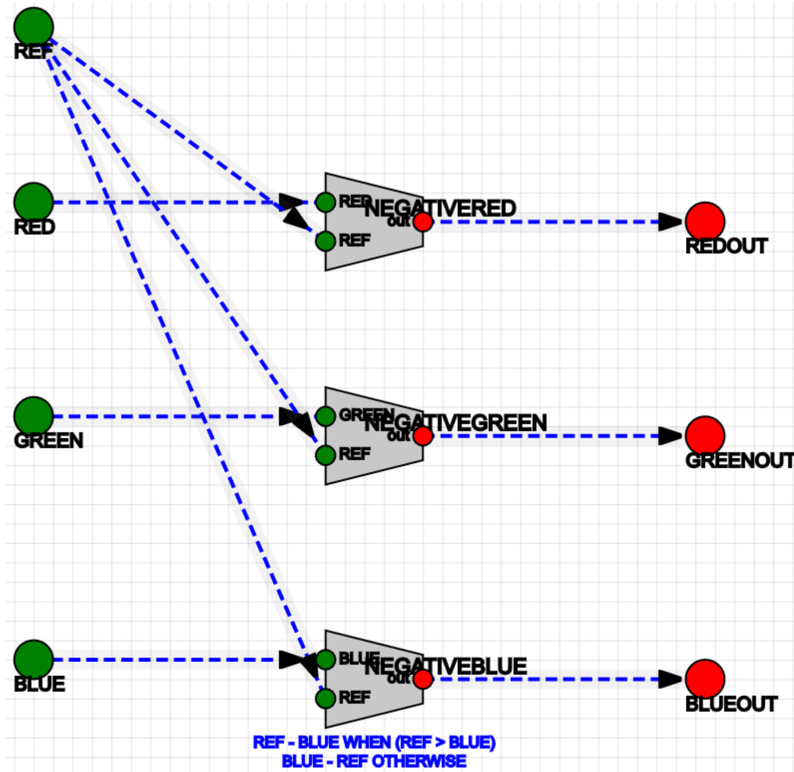

Figure 5.6: Negative module.

5.4 Bright and contrast

The bright and contrast algorithm consists in applying contrast and consequently bright to a given pixel. The pixel components are multiplied by a contrast value, followed by a sum of the bright value. After these operations, if a low reference and a high reference are provided, to ensure if the result surpasses one of the limits described, it stays with the reference value. Figure 5.7 displays the bright and contrast module.

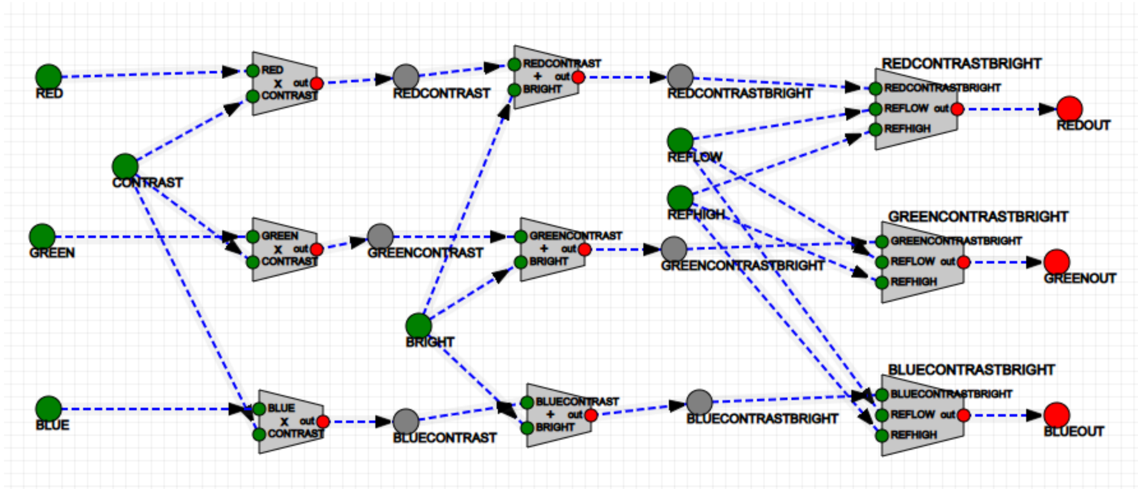

Figure 5.7: Bright and contrast module.

5.5 Simple grayscale

This grayscale module consists in getting the pixel components and calculate the average value of them. As it is not possible to make a division by 3, only by a multiple of 2, the following equation was implemented:

$$OUT = ((Red+Green+Blue) \times 683 + 1024) / 2048$$

The DS-Pnet model is displayed in Figure 5.8.

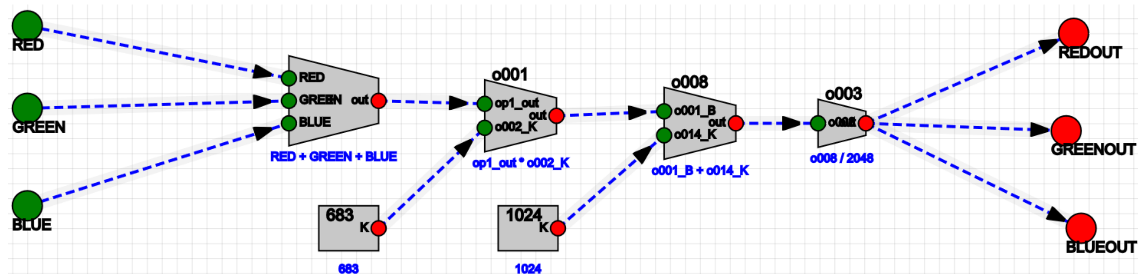

Figure 5.8: Simple Grayscale module.

5.6 Weighted grayscale

As described in section 4.3, the weighted grayscale algorithm consists in multiplying the red, green and blue components by constants, adding 512 and divide that result by 1024:

$$Gray = (Red\times306 + Green\times601 + Blue\times117 + 512) / 1024$$

The implemented module is represented in Figure 5.9.

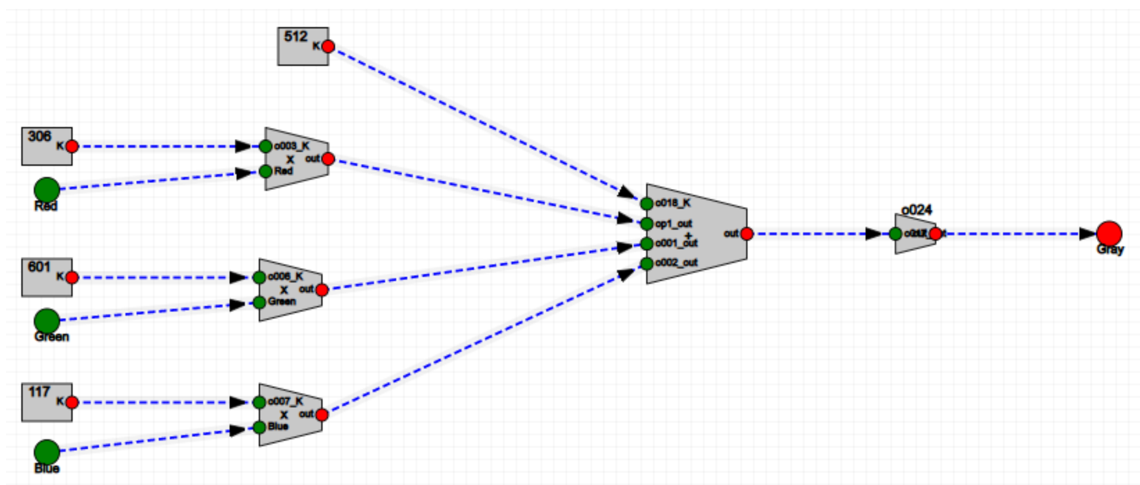

Figure 5.9: Weighted Grayscale module.

5.7 Binarization

The binarization module compares the colour input with a reference value. If the input is above the reference, the output will be the white colour (255), otherwise it will be black (0). The implemented module is displayed in Figure 5.10.

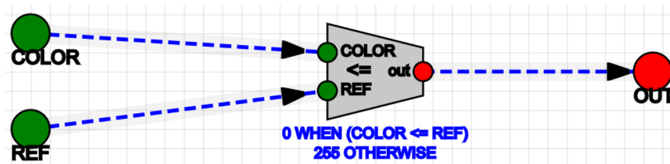

Figure 5.10: Binarization module.

5.8 Four weighted grayscales

The next component contains four instances of the weighted grayscale module, which means the grayscale is calculated for four pixels instead of one. It is presented in Figure 5.11.

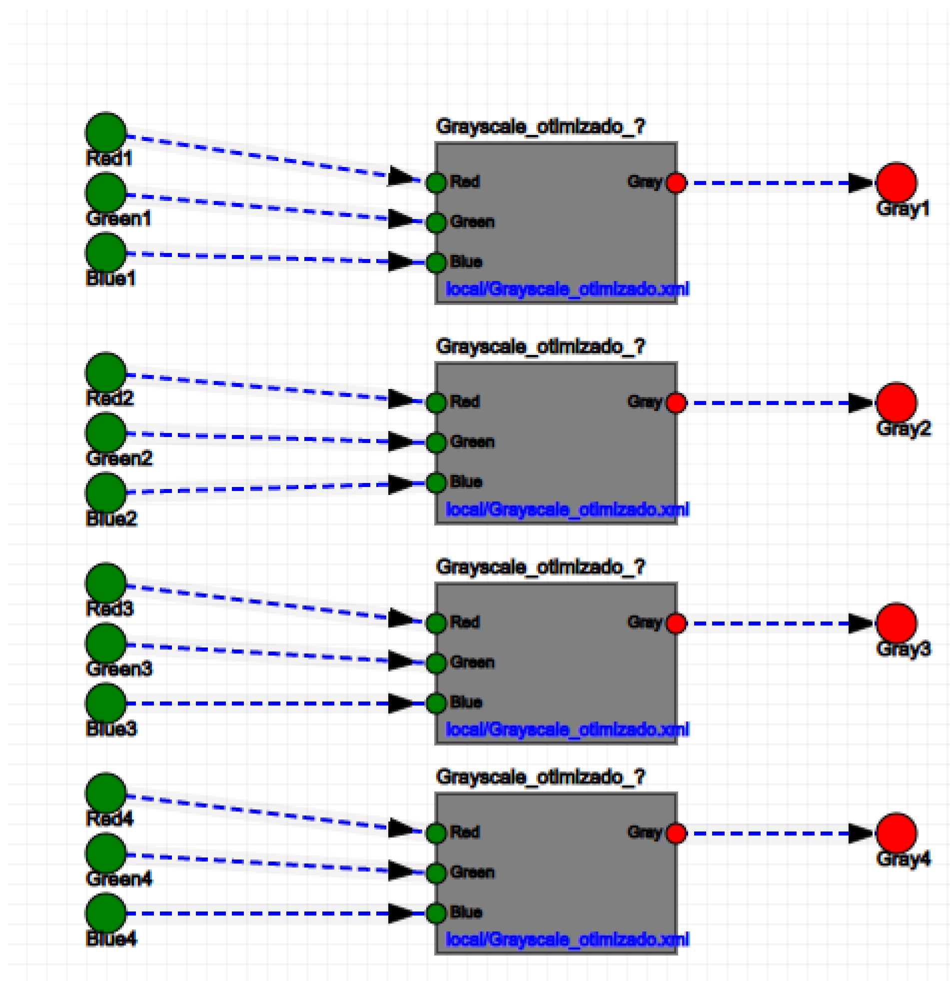

Figure 5.11: Four weighted grayscales module.

5.9 Four Weighted grayscale with register

The four weighted grayscales with register component are displayed in Figure 5.12. This component implements the grayscale algorithm for 4 pixels, and it has a load input. When this input is asserted, the values calculated are stored in 4 registers, which display the value in the output. When load is not asserted, the grayscale can be used to calculate new values, but the output will not change.

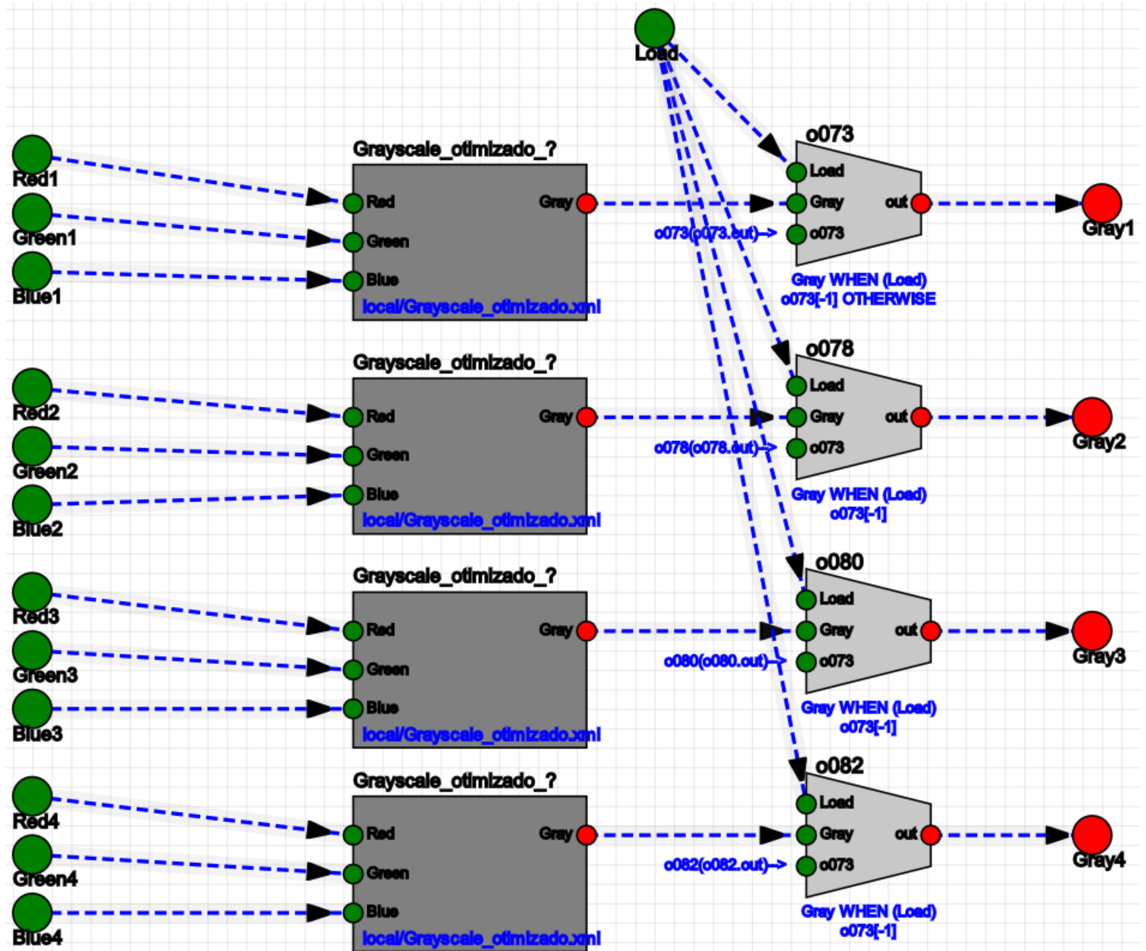

Figure 5.12: Four weighted grayscale with register module.

5.10 Horizontal projection

The horizontal projection counts the pixel intensity in a line and accumulates that result (the number of white pixels) for every line of the image. In the end, it is available the pixel intensity for each line of the image. In IOPT-Flow, this algorithm was implemented as displayed in Figure 5.13.

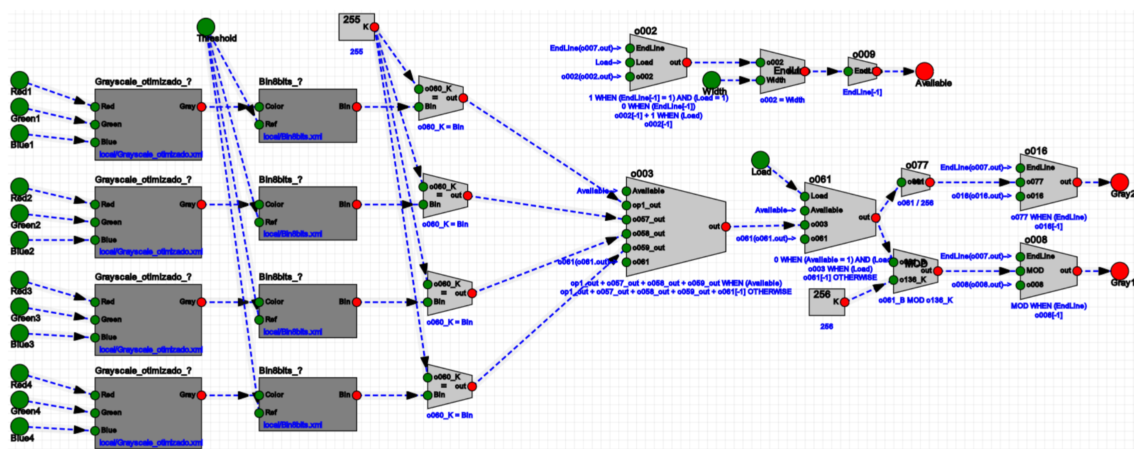

Figure 5.13: Horizontal projection component.

There are 4 weighted grayscale components. Each of these components receive 1 pixel, and the weighted grayscale algorithm is applied. The result is binarized, with a threshold input. The output of the binarization module is then compared with the constant value of 255. If it is equal, the output will be white, and it is taken as an input in the sum module, to be counted.

The sum module ("o003") has 6 inputs, the 4 outputs of the 4 binarization modules, an input called available, and the input "o061" that receives the previous value of operation "o061" (represented in the expression by "o061[-1]"). The available input controls if the sum module adds the previous "o061"value with the sum of the 4 outputs from the binarization modules, or not. The first situation happens when the line of the image has not ended and the second situation when it's the end of a line. When available is set, the second situation happens.

For the available input to be set, a few conditions must be met. There is a counter ("o002") to count until reaches the maximum image size divided by 4 (for example if the image size is 1920, it counts until 480), when the load input is active. This means a full line has been read. When this happens, the available input is set. The sum module outputs the added values.

The accumulated module ("o061") can store values until 1920, which uses more than 8 bits, so the output must be divided into 2, 8 bits, outputs. To get the lower 8 bits of the "o061" output, the MOD operation is used. To get the higher part of the number, "o061" output shifted to the right 8 times, using the operation "o077" that divides the output by 256.

The horizontal projection module has two outputs that represent an output value with a maximum value of 1920. This value is updated in the operations "o008" and "o016" when the end of line is reached. The available output signals the value update.

5.11 Closing remarks

It is important to note that the VHDL code automatically generated by the IOPT-Flow tools synchronizes all inputs and outputs with the rising edge of the clock. To make the generated codeto not be clock dependent, the code must be manually changed to remove the mentioned synchronizations.

The ability to simulate models in the IOPT-Flow tools simplifies the model creation. If the IOPT-Flow simulation tool was not presented, the generated code of the component had to be included, for example in a project in Xilinx, and then simulated there, which would lead to a slower development of the module.

The possibility to design DS-Pnet models, create the associated components, and instantiate then in other DS-Pnet models, makes development easier. For the example, to build the horizontal projection model, other models were instantiated/used. This makes the DS-Pnets very scalable.

The developed IOPT-Flow modules were described in this chapter. In the next chapter it is presented the validation of these modules.

6 Validation

6.1 Introduction

This section presents the validation of the Validation framework, GLCM system, and DS-Pnet modules, presented in chapters 3, 4, and 5. The proposed Validation framework enables the implementation and tests of the GLCM module and DS-Pnet modules. Each DS-Pnet module was simulated in the IOPT-Flow simulation tools and in the ISIM (ISE Simulator) from Xilinx ISE 14.7; however, only the timing diagrams from ISIM are presented here. Both the GLCM system and DS-Pnet modules were implemented in the FPGA. Equivalent algorithms were also implemented in software using the OpenCV library. The presented execution times depend not only on each image processing module but also on the validation frameworks.

The GLCM algorithm implementation mas made in software and hardware. In the software implementation, it was used OpenCV version 3.3.0 and C++, in the operating system Linux Ubuntu 18.04. The processor used was an Intel Core i7-8550U, 1,8 GHz and the computer have 16 GB of RAM memory.

The algorithms presented in chapter 5 that were implemented in hardware using DS-Pnets and IOPT-Flow, were also implemented in software. For the software implementation, it was used EmguCV version 2.4.10.1939 and C#, in the operating system Windows 10. The processor used was an Intel Core i7-4700MQ CPU, 2.4 GHz and the computer have 8 GB of RAM memory.

To validate the implementations, the same images were used in the software and hardware tests. To obtain the results presented in this chapter, two images were used. For the GLCM algorithm Figure 6.1 was used; whereas for the other algorithms Figure 6.2 was used. The image displayed in Figure 6.1, with dimensions 640x480, was provided by João Pedro Matos Carvalho, which was also used in the implementation of the GLCM algorithm in OpenCV. Figure 6.2 is just the first 48 lines from Figure 6.1 (640x48).

Figure 6.1: Image used for testing with dimensions 640x480.

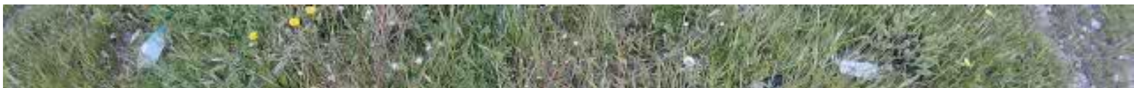

Figure 6.2: Image used for testing with dimensions 640x48.

6.2 Green channel and Red channel

The red channel algorithm was simulated, and the result is presented in Figure 6.3. Only 60 ns of simulation are displayed. The red channel algorithm puts the value of the red input in the output, called result, as confirmed in the simulation.

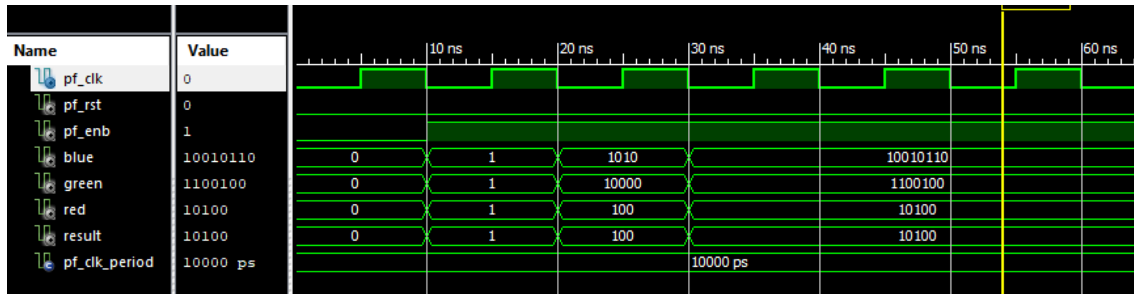

Figure 6.3: Red channel algorithm simulation.

After the simulation, the algorithm was executed in the validation platform, and took 364 217 clocks, which is approximately 2913.7 microseconds. The same implementation was made in OpenCV, resulting in a 184 microseconds of execution time. In Figure 6.4 is presented the result obtained in EmguCV, which is equal to the one obtained in the FPGA.

Figure 6.4: Red channel algorithm implementation in EmguCV.

The green channel algorithm also was simulated, and the result is presented in Figure 6.5. The first 120 ns of simulation are displayed.

After the simulation, the algorithm was executed in the validation platform. It took 364 227 clocks, which is approximately 2914.2 microseconds. The same implementation was made in EmguCV, resulting in a 183 microseconds of execution time. The result is displayed in Figure 6.6.

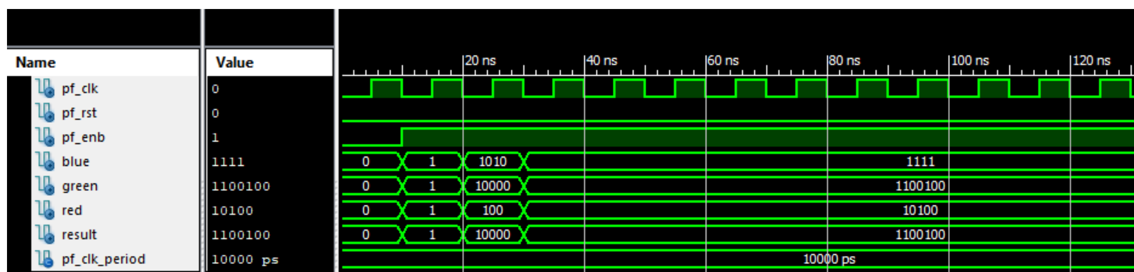

Figure 6.5: Green channel algorithm simulation.

Figure 6.6: Green channel algorithm implementation in EmguCV.

6.3 Negative

The result from the negative algorithm simulation is presented in Figure 6.7. For presentation, only 60 ns are displayed.

After the simulation, the algorithm was executed in the validation platform, and took 364 232 clocks (approximately 2913.9 microseconds). The same implementation was made in EmguCV, resulting in a 154.3 microseconds of execution time. The result is presented in Figure 6.8.

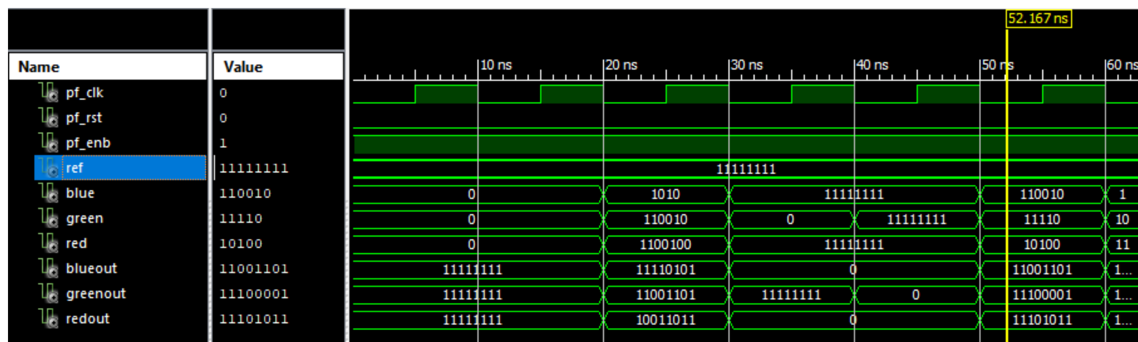

Figure 6.7: Negative algorithm simulation.

Figure 6.8: Negative algorithm implementation in EmguCV.

6.4 Bright and contrast

The bright and contrast algorithm was simulated, with the result presented in Figure 6.9. The first 60 ns of the simulation are displayed. The algorithm applies bright and contrast to the image, leaving the output, as confirmed in the simulation.

After the simulation, the algorithm was executed in the validation platform, and took 364 209 clocks, which is approximately 2913.7 microseconds. The same implementation was made in EmguCV, resulting in a 1156.8 microseconds of execution time. The result for contrast = 2 and bright = 64 is displayed in Figure 6.10.

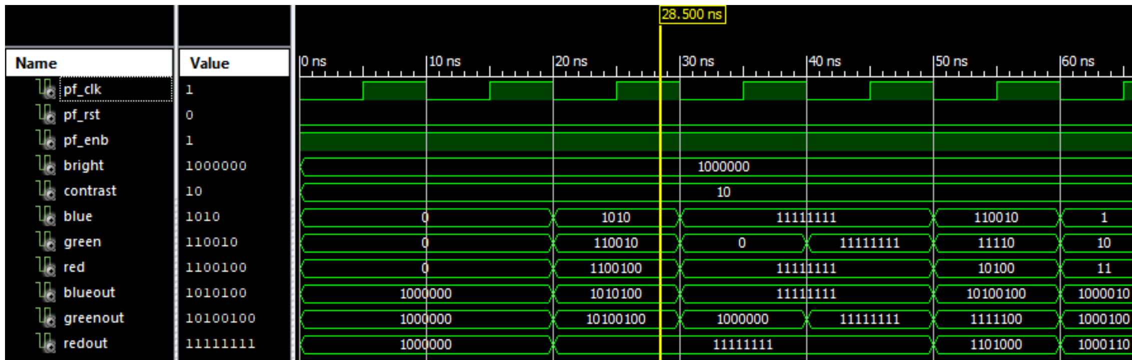

Figure 6.9: Bright and contrast algorithm simulation.

Figure 6.10: Bright and contrast algorithm implementation in EmguCV.

6.5 Simple grayscale

The simplest version of the grayscale algorithm was simulated, and the first 70 ns are presented in Figure 6.11. After the simulation, the algorithm was executed in the validation platform, and took 364 213 clocks, which is approximately 2913.7 microseconds. The implementation in EmguCV took 496 microseconds and it is displayed in Figure 6.12.

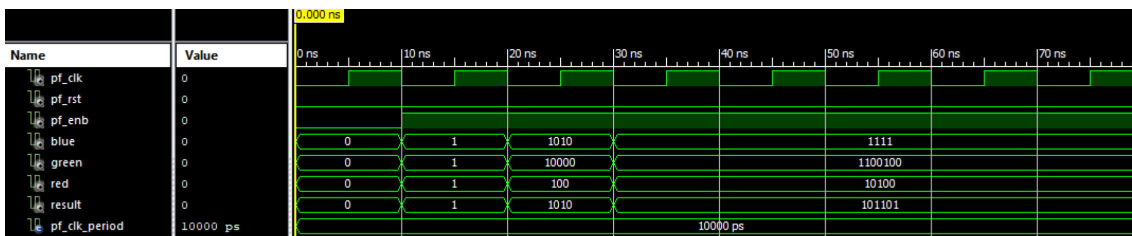

Figure 6.11: Grayscale simpler version algorithm simulation.

Figure 6.12: Grayscale simpler version algorithm implementation in EmguCV.

6.6 Weighted grayscale

The grayscale optimized algorithm descripted in previous chapters was simulated, and the result is presented in Figure 6.13. The first 140 ns of simulation are displayed. The algorithm applies the grayscale algorithm, leaving the output, as confirmed in the simulation.

After the simulation, the algorithm was executed in the validation platform, and took 364 179 clocks, which is approximately 2913.4 microseconds. The implementation in EmguCV took 478.5 microseconds. The result is displayed in Figure 6.14.

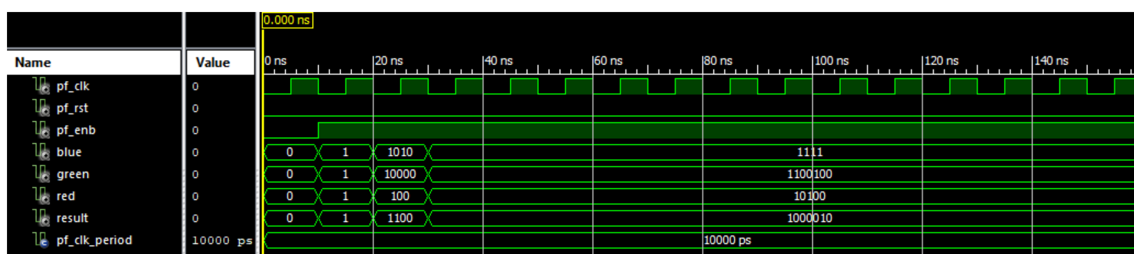

Figure 6.13: Grayscale optimized algorithm simulation.

Figure 6.14: Grayscale optimized algorithm implementation in EmguCV.

6.7 Binarization

The binarization algorithm was also simulated, and the result is presented in Figure 6.15. Only 120 ns of simulation are displayed. After the simulation, the algorithm was executed in the validation platform, and took 364 216 clocks, which is approximately 2913.7 microseconds. The execution time of the software implementation is 688 microseconds. The resulting image is presented in Figure 6.16.

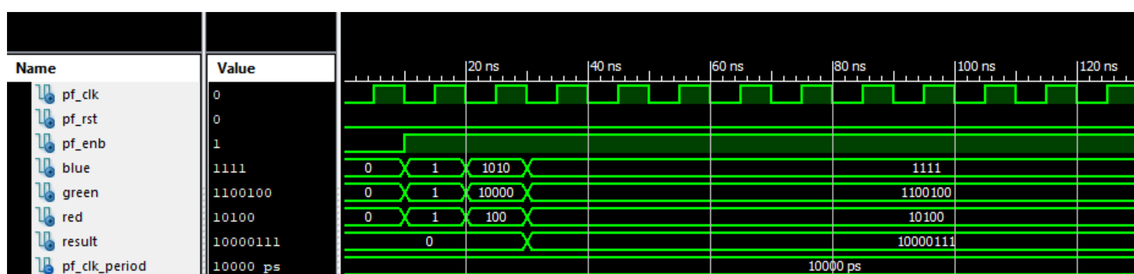

Figure 6.15: Binarization algorithm simulation.

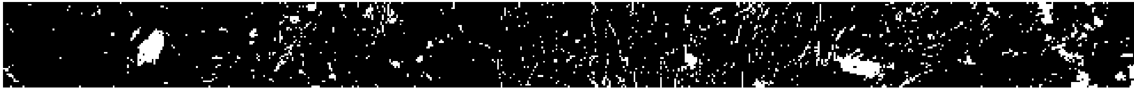

Figure 6.16: Resulting image from the Binarization algorithm.

6.8 Four weighted grayscales

The grayscale optimized algorithm descripted in previous chapters was developed for 4 pixels. The simulation is presented in Figure 6.17. Only 140 ns of simulation are displayed. The algorithm applies the grayscale algorithm to each pixel.

After the simulation, the algorithm was executed in the validation platform, and took 364 204 clocks, which is approximately 2913.6 microseconds.

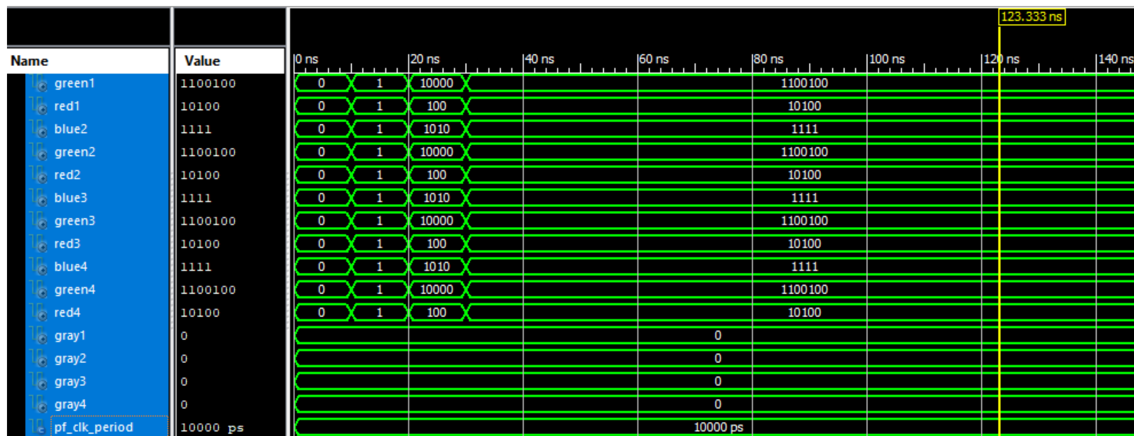

Figure 6.17: Grayscale optimized algorithm for 4 pixels simulation.

6.9 Four weighted grayscales with register

The grayscale algorithm with a register was simulated, being composed by 4 similar modules, and the result is presented in Figure 6.18. It is displayed 140 ns of simulation. The algorithm implements the previously descripted grayscale optimized version in earlier chapters with a register. When the load input is asserted, the outputs will receive the result of the application of the algorithm, as confirmed in the simulation.

After the simulation, the algorithm was executed in the validation platform, and took 364 205 clocks, which is approximately 2913.6 microseconds.

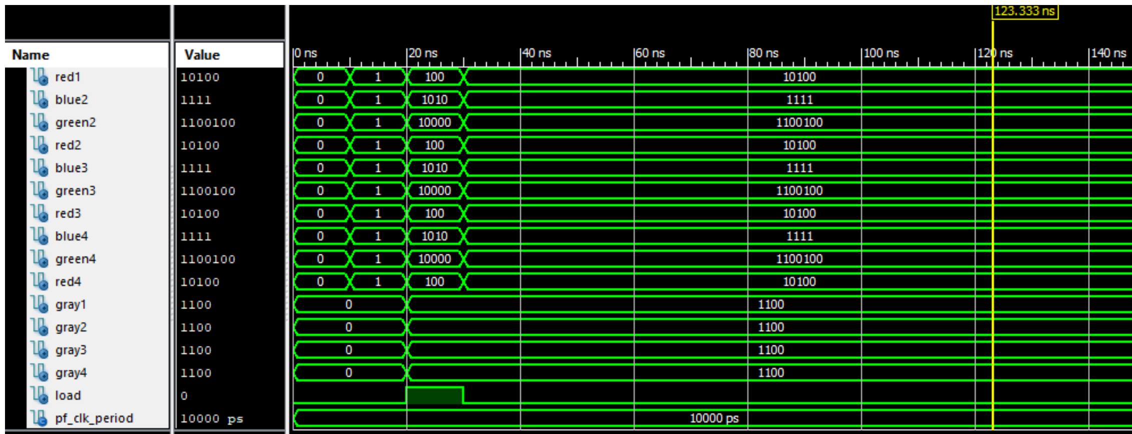

Figure 6.18: Grayscale with a register algorithm simulation.

6.10 Horizontal projection

The horizontal projection algorithm was simulated, and the result is presented in Figure 6.19. For the binarization, the threshold used was 127. It was displayed 140 ns of the simulation. After the simulation, the algorithm was executed in the validation platform, and took 364 217 clocks, which is approximately 2913.7 microseconds. The execution time of the software implementation is 739 microseconds. The values obtained per line by both FPGA and EmguCV are presented in a chart in Figure 6.20 for comparison. In the 48 lines, eleven lines have different projection discrepancies of 1 value and one line with a discrepancy of 3 values.

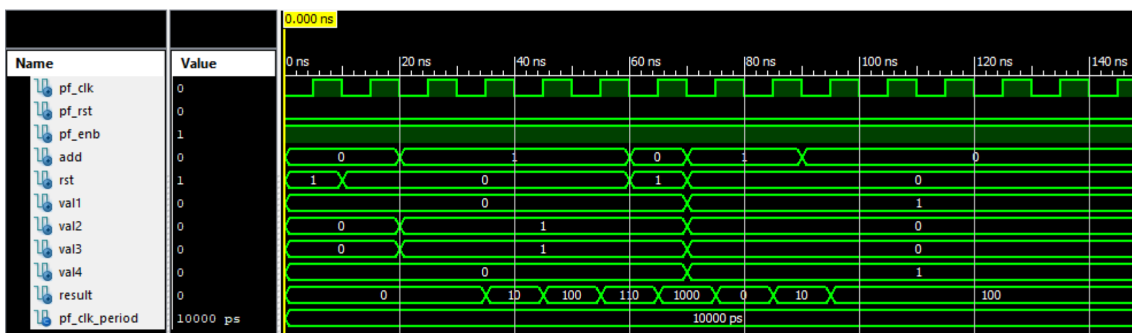

Figure 6.19: Horizontal projection algorithm simulation.

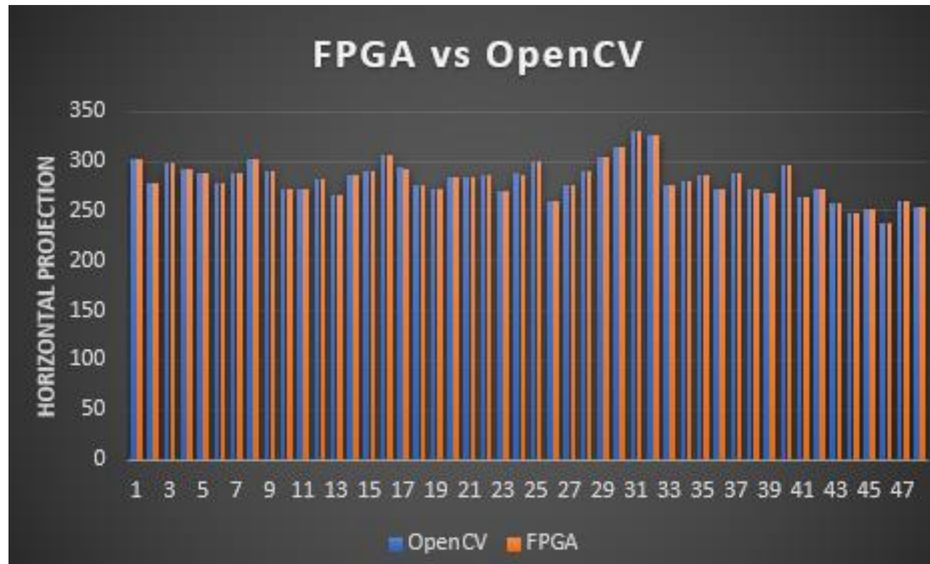

Figure 6.20: Horizontal projection algorithm simulation.

6.11 GLCM algorithm execution

The execution time of the GLCM algorithm was measured for the application of the contrast and the energy algorithms. It starts counting from the moment the image is stored in the DDR2 until it's processed. In Table 6.1 are displayed the correspondent execution times in both platforms, FPGA and software.

Table 6.1: GLCM execution time.

Platform	Execution Time (seconds)
FPGA	0.27
OpenCV	0.03

The image features extracted from the original image in the validation platform are very similar to the features extracted in the software implementation. In Table 6.2 is displayed the sum of all the matrix values obtained by the execution of the GLCM algorithm in OpenCV and in the FPGA. Were made trials for the full image, with size of 640x480.

Table 6.2: Results obtained from executing the algorithms in the FPGA vs OpenCV.

Full Image (640x480)		
Algorithm	**Implementation in FPGA**	**Implementation in OpenCV**
Contrast	451 169 734	451 179 094
Energy	52 432 710	52 042 888

6.12 Results discussion

This section discusses the results obtained in this chapter. The discussion focuses on the algorithms output and on their execution time. The FPGA implementations are compared with the same implementation developed in software.

6.12.1 Execution times

In this section, the image processing algorithms' execution times are compared for the FPGA and OpenCV implementations. It is important to note that these algorithms' execution times depend not only on their modules/functions, but also on the implementation platforms.

Figure 6.21 resumes the results presented in chapter 6 for the execution time. The algorithms implemented in the FPGA have similar execution times. This is because their operations are mainly implemented as combinational logic in the hardware, which can execute the algorithm in a clock period. In the OpenCV implementation, the execution time increases with the number of operations and on their complexity as presented in Figure 6.21.

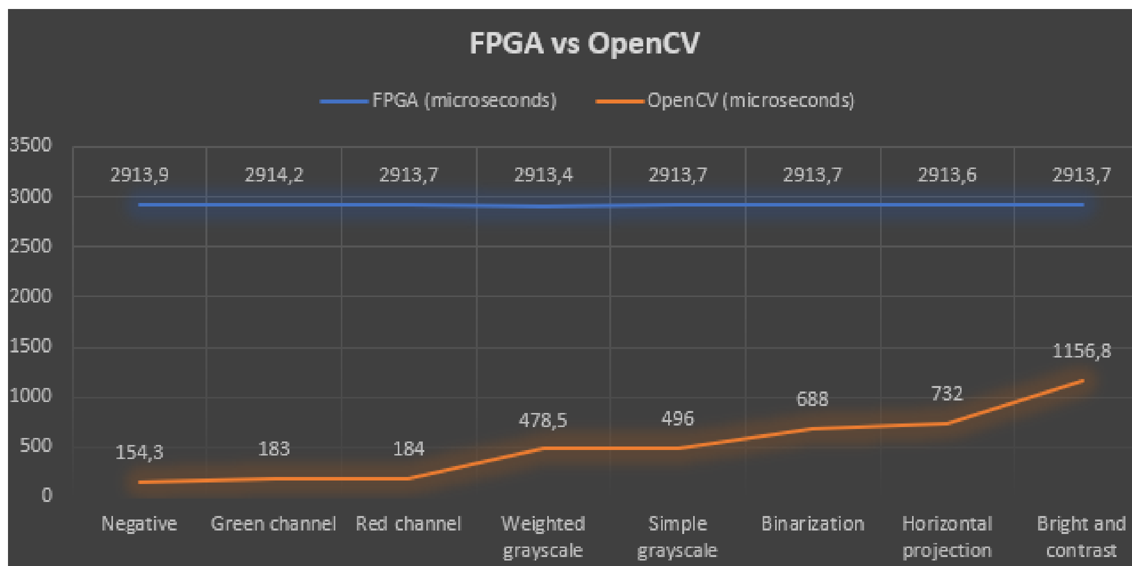

Figure 6.21: FPGA vs OpenCV execution times.

The presented execution times for the weighted grayscale and the four weighted grayscales are similar in the FPGA implementation. Since the validation platform is designed to processed 4 pixels in parallel, when the weighted grayscale is used, it is instantiated four times in the project; whereas, when the four weighted grayscales module is used, it is only instantiated one time, which leads to similar implementation codes.

The hardware implementation of the GLCM algorithm is approximately 10 times slower than the software implementation. In Figure 6.21 a similar result is presented. The algorithms

execution time tend to be 10 times slower in the hardware implementation. The high execution time of the hardware implementation is mainly due to the DDR2 memory. The use of the DDR2 slowed the execution time, since it needs several clocks to perform a read/write operation, as described in section 3.4

The DDR2 has the auto refresh operation, which explains the differences in execution time between similar image processing algorithms, as presented in Figure 6.21. This justifies that even several executions of the same algorithm have different execution times.

6.12.2 Outputs

The hardware and software implementations, of the image processing algorithms presented in chapters 4 and 5, have equal or similar outputs, depending on the implemented algorithms. There are some minor discrepancies in the algorithms that implement divisions, due to different rounding. The hardware implementation has less precision. Although it is possible to increase it, it is not required because the differences in rounding do not cause a negative impact on the results. The presented algorithms that implement divisions are those that calculate the grayscale: simple grayscale; weighted grayscale; four weighted grayscales; four weighted grayscales with register; horizontal projection; and GLCM algorithm. For example, the image from Figure 6.1 (with 307200 pixels) was processed by both implementations of the weighted grayscale algorithm. In the resulting images, 5460 were not the equal, which is less than 2%. Another example is the one presented in section 6.10, where the horizontal projection was calculated. In this example, the discrepancy was 12 pixels in 30720, which means 0.04%.

7 Conclusions and future work

The development of an image processing system in hardware is a challenging task. VHDL is a verbose language that makes the specification in VHDL a laborious and slow task. Simulations specification, execution, and analysis are also laborious and slow. Synthesis and design implementation take time. Finally, tests are not fast either.

The use of DS-Pnets and IOPT-Flow tools present several benefits. DS-Pnets have a higher level of abstraction, when compared to VHDL, which can speed up the specification. Additionally, they are a graphical formalism, which can make it more intuitive. The IOPT-Flow interactive token-player simulator tool, makes simulation easier and faster, reducing the development time. The VHDL automatic code generator avoids manual codification errors. The integration of the generated VHDL code was relatively simple, since the models were developed to fit the image processing system.

Additionally, it comes handy when members of development team are not familiar with VHDL. Compared to HLS, DS-Pnets and IOPT-Flow, should support optimized implementations in terms of resources and speed.

During this work, the DS-Pnets and the IOPT-Flow tools, were successfully used develop several image processing systems. The collection of the developed IOPT-Flow components is the beginning of an IOPT-Flow library for image processing. DS-Pnets and the associated tools should be considered to develop FPGA-based image processing systems.

Future extensions for the DS-Pnets and IOPT-Flow tools were identified. Currently, the VHDL code, automatically generated, is always synchronized by the rising edge of the same clock. In future, it should be possible to specify synchronizations with different clocks, in the rising edge and in the falling edge. For example, because the DDR2 interface operates in the falling edge of the clock, to integrate automatically generated code, it was required to manually update the edge from rising to falling. Finally, currently the inputs and outputs, of the automatically generated code, are synchronized with the clock. It should be possible to select if they are synchronized or not.

It was not possible to verify performance gains, as stated by other authors, in FPGA-based implementations, when compared to software implementations. This is due to the FPGA-based platform used during this work. The DDR2 memory available on the board was used to save images and their features; however, reading and writing operations are slow, which had a strong impact on the execution time. As future work, it is planned to use an FPGA with a large BRAM (Block RAM), which allows fast read/write operations.

From this work resulted one published paper at the time of writing this document. The paper, with the title “FPGA in image processing”, from the authors Tiago Carrasqueira, Filipe Moutinho, and Rogério Campos-Rebelo, was published in the proceedings of REC”2019 – XV Jornadas sobre Sistemas Reconfiguráveis, that took place from 14 to15 of February 2019 in Minho University, Guimarães, Portugal [6].

References

[1] S. Jin *et al.*, "The comparison of CPU time consumption for image processing algorithm in Matlab and OpenCV," *Proc. - IEEE 25th Annu. Int. Symp. Field-Programmable Cust. Comput. Mach. FCCM 2017*, vol. 606, no. 1, pp. 1–6, 2017.

[2] Y. Ma, *"The mathematic magic of Photoshop blend modes for image processing."* International Conference on Multimedia Technology, ICMT, 2011.

[3] T. Bailey, *"An Introduction to the C Programming Language and Software Design."* The University of Sydney, 2005.

[4] S. Palnitkar, *"Verilog HDL,"* Second., vol. 53, no. 9. Prentice Hall PTR, 2013.

[5] F. Pereira and L. Gomes, "The IOPT-Flow Modeling Framework Applied to Power Electronics Controllers," *IEEE Trans. Ind. Electron.*, vol. 64, no. 3, pp. 2363–2372, 2017.

[6] T. Carrasqueira, F. Moutinho, and R. Campos-rebelo, "FPGA in image processing," *REC'2019 – XV Jornadas sobre Sist. Reconfiguráveis*, p. 6, 2019.

[7] R. C. Gonzalez, R. E. Woods, and P. Hall, *"Digital Image Processing,"* Second. Prentice Hall, 2002.

[8] S. Chabchoub, S. Mansouri, and R. Ben Salah, "Biomedical monitoring system using LabVIEW FPGA," *2015 World Congr. Inf. Technol. Comput. Appl. WCITCA 2015*, pp. 3–7, 2015.

[9] A. P. D. Binotto, D. Doering, T. Stetzelberger, P. McVittie, S. Zimmermann, and C. E. Pereira, "A CPU, GPU, FPGA System for X-Ray Image Processing Using High-Speed Scientific Cameras," *2013 25th Int. Symp. Comput. Archit. High Perform. Comput.*, pp. 113–119, 2013.

[10] D. Sanjay, T. S. Savithri, and P. R. Kumar, "Person follower robotic system," *2014 Int. Conf. Control. Instrumentation, Commun. Comput. Technol. ICCICCT 2014*, pp. 1324–1327, 2014.

[11] S. Hashemi, H. Tann, F. Buttafuoco, and S. Reda, "Approximate computing for biometrie security systems: A case study on iris scanning," in *Proceedings of the 2018 Design, Automation and Test in Europe Conference and Exhibition, DATE 2018*, 2018, vol. 2018-Janua, pp. 319–324.

[12] Z. Moutakki, T. Ayaou, K. Afdel, and A. Amghar, "Prototype of an embedded system using Stratix III FPGA for vehicle detection and traffic management," *Int. Conf. Multimed. Comput. Syst. -Proceedings*, pp. 141–146, 2014.

[13] O. W. Ibraheem, A. Irwansyah, J. Hagemeyer, M. Porrmann, and U. Rueckert, "Reconfigurable vision processing system for player tracking in indoor sports," *Conf. Des. Archit. Signal Image Process. DASIP*, vol. 2017-Septe, pp. 1–6, 2017.

[14] C. Moler, "The Origins of MATLAB," 2004. [Online]. Available: https://www.mathworks.com/company/newsletters/articles/the-origins-of-

matlab.html. [Accessed: 09-Jan-2019].

[15] O. Marques, *Practical Image and Video Processing Using MATLAB®*. 2011.

[16] Mathworks, *Simulink User ' s Guide*. The MathWorks, 2015.

[17] Y. Du, L. Gai, J. Tian, and W. Liu, "Digital Image Processing Teaching Auxiliary System Based on MATLAB Graphical User Interface," *2015 7th Int. Conf. Inf. Technol. Med. Educ.*, pp. 434–438, 2015.

[18] S. Z. Murshed *et al.*, "Controlling an Embedded Robot through Image Processing based Object Tracking using MATLAB," *2016 10th Int. Conf. Intell. Syst. Control*, pp. 1–6, 2016.

[19] P. Arora, A. Sharma, A. S. Soni, and A. Garg, "Control of wheelchair dummy for differently abled patients via iris movement using image processing in MATLAB," *12th IEEE Int. Conf. Electron. Energy, Environ. Commun. Comput. Control (E3-C3), INDICON 2015*, pp. 4–7, 2016.

[20] S. Matuska, R. Hudec, and M. Benco, "The comparison of CPU time consumption for image processing algorithm in Matlab and OpenCV," *Proc. 9th Int. Conf. ELEKTRO 2012*, pp. 75–78, 2012.

[21] D. D. Bloisi *et al.*, *Learning Image Processing with OpenCV*, vol. 1. PACKT, 2015.

[22] J. Uma and P. Yuvarani, "Detection of shapes and counting in toy manufacturing industry with help of Phython," *Proc. - 2017 IEEE Int. Conf. Electr. Instrum. Commun. Eng. ICEICE 2017*, vol. 2017-Decem, pp. 1–5, 2017.

[23] I. Ali, A. Malik, W. Ahmed, and S. A. Khan, "Real-Time Vehicle Recognition and Improved Traffic Congestion Resolution," *Proc. - 2015 13th Int. Conf. Front. Inf. Technol. FIT 2015*, pp. 228–233, 2016.

[24] S. Jin, H. Zedong, and L. Yuan, "Software implementation of corn grain morphology detection based on OpenCV," *ICEMI 2017 - Proc. IEEE 13th Int. Conf. Electron. Meas. Instruments*, vol. 2018-Janua, pp. 412–415, 2018.

[25] D. Phillips, *Image Processing in C Second Edition*, vol. i, no. C. R & D Publications, 2000.

[26] Y. Chen and P. Sun, "The Research and Practice of Medical Image Enhancement and 3D Reconstruction System," *2017 Int. Conf. Robot. Intell. Syst.*, pp. 350–353, 2017.

[27] "Which image software should I use." [Online]. Available: https://horusaeronaves.com/qual-software-de-processamento-de-imagens-devo-utilizar/. [Accessed: 31-Jan-2019].

[28] R. E. Haskell and D. M. Hanna, *Introduction to Digital Design Using Digilent FPGA Boards*. 2009.

[29] D. J. Smith, O. Madison, and I. Estate, "VHDL & Verilog Compared & Contrasted - Plus Modeled Example Written in VHDL, Verilog," in *VeriBest Incorporated*, .

[30] I. Kuon and J. Rose, "Measuring the gap between FPGAs and ASICs," *IEEE*

Trans. Comput. Des. Integr. Circuits Syst., vol. 26, no. 2, pp. 203–215, 2007.

[31] A. Ehliar and A. Ehliar, *Performance driven FPGA design with an ASIC perspective*. LiU-Tryck, Linköping, 2009.

[32] M. Kiran, K. M. War, L. M. Kuan, and L. K. Meng, "Implementing image processing algorithms using 'Hardware in the loop' approach for Xilinx FPGA," *Int. Conf. Electron. Des. (ICED 2008), Penang*, pp. 1–6, 2008.

[33] A. Fazakas, M. Neag, and L. Feçtilă, "Block RAM versus distributed RAM implementation of SVM classifier on FPGA," *Int. Conf. Appl. Electron. 2006, AE*, pp. 43–46, 2006.

[34] T. Tiemerding, C. Diederichs, C. Stehno, and S. Fatikow, "Comparison of different design methodologies of hardware-based image processing for automation in microrobotics," *2013 IEEE/ASME Int. Conf. Adv. Intell. Mechatronics Mechatronics Hum. Wellbeing, AIM 2013*, pp. 565–570, 2013.

[35] X. Han and L. Xing, "Mobile robot based on FPGA," *Proc. - 3rd Int. Conf. Intell. Transp. Big Data Smart City, ICITBS 2018*, vol. 2018-Janua, pp. 576–579, 2018.

[36] M. B. McMickell, P. J. Tanzillo, T. Kreider, and K. Ilic, "Rapid development of space applications with Responsive Digital Electronics Board and LabVIEW FPGA," *2010 NASA/ESA Conf. Adapt. Hardw. Syst. AHS 2010*, pp. 79–81, 2010.

[37] J. A. Kalomiros and J. Lygouras, "A Host/Co-processor FPGA-Based Architecture for Fast Image Processing," *Proc. Int. Work. Intell. Data Acquis. Adv. Comput. Syst. Technol. Appl.*, no. September, pp. 373–378, 2007.

[38] W. Ye, "Efficient Implementation of FPGA Based on Vivado High Level Synthesis," *2016 2nd IEEE Int. Conf. Comput. Commun. Effic.*, no. 36, pp. 2810–2813, 2016.

[39] M. Adeel, "Hardware Acceleration of Image Processing," *Int. Conf. Intellegent Comput. Control Syst. 2017*, pp. 29–34, 2017.

[40] W. Ipanaque, J. Salazar, and I. Belupu, "Implementation of an architecture of digital control in FPGA commanded from an embedded Java application," *2016 IEEE Int. Conf. Autom. ICA-ACCA 2016*, no. Ic, 2016.

[41] A. Rupani, P. Whig, G. Sujediya, and P. Vyas, "A robust technique for image processing based on interfacing of Raspberry-Pi and FPGA using IoT," *2017 Int. Conf. Comput. Commun. Electron. COMPTELIX 2017*, pp. 350–353, 2017.

[42] F. Pereira and L. Gomes, "Combining data-flows and petri nets for cyber-physical systems specification," *IFIP Adv. Inf. Commun. Technol.*, vol. 470, pp. 65–76, 2016.

[43] F. Joaquim and G. Pereira, "The DS-Pnet modeling formalism for cyber-physical system development," FCT UNL, 2017.

[44] F. Pereira, F. Moutinho, J. Ribeiro, and L. Gomes, "Web based IOPT Petri net Editor with an extensible plugin architecture to support generic net operations," *IECON Proc. (Industrial Electron. Conf.*, pp. 6151–6156, 2012.

[45] F. Pereira and L. Gomes, "The IOPT-Flow framework pairing Petri nets and

data-flows for embedded controller development," *IECON Proc. (Industrial Electron. Conf.*, pp. 4832–4837, 2016.

[46] U. Guide, "UG086 Xilinx Memory Interface Generator (MIG), User Guide," *ReVision*, vol. 086, 2009.

[47] A. Peeters and K. van Berkel, "Single-rail handshake circuits," in Proceedings of the 2Nd Working Conference on Asynchronous Design Methodologies, 06 1995, pp. 53 – 62..

[48] R. M. Haralick and K. Shanmugam, "Haralick Texture," *IEEE Trans. Syst. MAN Cybern.*, vol. SMC-3, pp. 610–621, 1973.