MASTER'S THESIS

Morphological Image Reconstruction Implementation Using a Hardware-Software Approach in FPGA

Felipe Regis Gonçalves Cabral

Brasília

2018, June

UNIVERSIDADE DE BRASÍLIA

FACULDADE DE TECNOLOGIA

UNIVERSIDADE DE BRASÍLIA
FACULDADE DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA MECÂNICA

MASTER'S THESIS

Morphological Image Reconstruction Implementation Using a Hardware-Software Approach in FPGA

Felipe Regis Gonçalves Cabral

THESIS SUBMITTED TO THE DEPARTMENT OF MECHANICAL ENGINEERING IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER IN MECHATRONIC SYSTEMS

Examination Board

Prof. Ricardo Pezzuol Jacobi, CIC/UnB ______________________
Supervisor

Prof. Carlos H. Llanos Quintero, ENM/UnB ______________________
Co-Supervisor

Prof. Alba Cristina M. A. de Melo, CIC/UnB ______________________
Chair Member

Prof. Daniel Mauricio Muñoz Arboleda, ENM/UnB ______________________
Chair Member

FICHA CATALOGRÁFICA

Cabral, Felipe Regis Gonçalves

Morphological Image Reconstruction Implementation Using a Hardware-Software Approach in FPGA / Felipe Regis Gonçalves Cabral. –Brasil, 2018.

75 p.

Orientador: Ricardo Pezzuol Jacobi

Co-Orientador: Carlos H. Llanos Quintero

Dissertação (Mestrado) – Universidade de Brasília – UnB
Faculdade de Tecnologia – FT
Programa de Pós-Graduação em Sistemas Mecatrônicos – PPMEC, 2018.

1. FPGA. 2. Reconstrução Morfológica. 3. Processamento de Imagem. 4. Sistemas Embarcados 5. Hardware. 6. Software. I. Ricardo Pezzuol Jacobi, orientador. II. Carlos H. Llanos Quintero, co-orientador. III. Universidade de Brasília. IV. Faculdade de Tecnologia.

REFERÊNCIA BIBLIOGRÁFICA

Cabral, F. R. G. (2018). *Morphological Image Reconstruction Implementation Using a Hardware-Software Approach in FPGA*. Dissertação de Mestrado em Sistemas Mecatrônicos. Publicação ENM.DM-144/2018, Departamento de Engenharia Mecânica, Faculdade de Tecnologia, Universidade de Brasília, Brasília, DF, 75p.

CESSÃO DE DIREITOS

AUTOR:Felipe Regis Gonçalves Cabral

TíTULO: *Morphological Image Reconstruction Implementation Using a Hardware-Software Approach in FPGA*

GRAU: Mestre ANO: 2018

É concedida à Universidade de Brasília permissão para reproduzir cópias desta dissertação e para emprestar ou vender tais cópias somente para propósitos acadêmicos e científicos. O autor reserva outros direitos de publicação e nenhuma parte desse trabalho de conclusão de curso pode ser reproduzida sem autorização por escrito do autor.

____

Felipe Regis Gonçalves Cabral
Departamento de Eng. Mecânica (ENM) - FT
Universidade de Brasília (UnB)
Campus Darcy Ribeiro
CEP 70919-970 - Brasília - DF - Brasil.

Dedication

This thesis work is dedicated to my beloved girlfriend, Rayssa, who has been an incredible source of support and encouragement. This would not be possible without you.

Felipe Regis Gonçalves Cabral

Acknowledgements

First and foremost I would like to thank God for this accomplishment. All the honor and glory goes to Him. I could not forget to say thanks to my supervisors at the University of Brasilia, Ricardo Jacobi and Carlos Llanos, for the valuable feedback and all the help during the course of the project. Special thanks to my mother Jandira, my father Carlos, and my brothers Fabio, Fabricio and Matheus for all the support and encouragement throughout the years of my life. I also must be deeply grateful to my best friend and love Rayssa Luna for all the companionship and love she has given to me. This MSc thesis literally would not be possible without your advises and encouragement. Finally, I would like to thanks to the Brazilian National Council for Scientific and Technological Development (CNPq), whose financially supported this research through a master degree scholarship.

Felipe Regis Gonçalves Cabral

RESUMO

A presente dissertação de mestrado implementa um algoritmo de reconstrução morfológica de imagens com uma abordagem hardware/software utilizando um sistema embarcado em FPGA. A parte de hardware foi desenvolvida em VHDL em uma FPGA da família Cyclone® V da Altera que possui um processador ARM® Cortex™A9 no mesmo chip, possibilitando a execução da parte de software em linguagem C. O algoritmo em si recebe como entrada duas imagens chamadas *marker* e *mask*, e entrega como saída uma imagem que é a reconstrução morfológica baseada no conteúdo das duas. Este trabalho implementa o algoritmo através de uma estratégia de particionamento de imagem em imagens menores, executando a versão sequencial (SR) do algoritmo em hardware simultaneamente à reconstrução usando fila nas fronteiras das sub-imagens em software. A proposta do trabalho é a utilização de propriedades inerentes à implementações em hardware, como a paralelização e alto desempenho, em conjunto com a flexibilidade e rapidez de um projeto de software para alcançar uma solução final que ao mesmo tempo possua um bom desempenho, tendo em vista as limitações de um sistema embarcado com pouca memória, e ofereça uma solução final flexível, permitindo ao usuário escolher o tamanho da imagem a ser processada via software. A arquitetura do hardware roda em frequência de 150 Mhz, utiliza protocolo Avalon para se comunicar, possui uma memória DDR3 externa de 1GB para armazenamento temporário das imagens e está conectado a um processador ARM Cortex-A9 através de um barramento AMBA® AXI3. A parte de software roda nesse processador a 925 Mhz e possui outra memória DDR3 de 1GB. O programa primeiro configura os registradores internos do hardware de acordo com os parâmetros escolhidos (tamanho da imagem, endereço de memória das imagens entre outros), ordena o hardware a realizar o algoritmo de reconstrução morfológica diversas vezes em imagens menores e depois executa a propagação do algoritmo entre as bordas dessas sub-imagens processadas pelo hardware. Como método de verificação funcional da solução, as imagens resultantes foram comparadas com as produzidas pelo MATLAB®. A melhor solução proposta por este trabalho alcançou uma melhoria em torno de 2x em comparação com a melhor solução teórica possível do algoritmo de reconstrução morfológica sequencial (SR) implementado em um hardware de 150 Mhz, uma melhoria de até 12x em relação ao algoritmo de reconstrução morfológica *Fast Hybrid* (FH) proposto por Vincent [1] rodando em um ARM® Cortex™A9 e uma melhoria de até 2x em comparação com o mesmo algoritmo rodando em uma CPU Intel® Core™ i5. Os testes finais para validar a solução mostram resultados corretos para imagens em escala de cinza com resolução de 8 bits de até 8192x8192 pixels (imagens 8k). Até onde se sabe, este é a primeira implementação hardware/software do algoritmo de reconstrução morfológica de imagens descrito na literatura.

ABSTRACT

This MSc dissertation implements a morphological image reconstruction algorithm using a hardware/software approach in an FPGA based embedded system. The hardware part was developed in VHDL in a Cyclone® V FPGA from Altera that has an ARM® Cortex™A9 processor in the same chip, allowing the execution of the software part in C language. The algorithm itself receives as input two images, called marker and mask, and generates as output a morphologically reconstructed image based on the content of the two inputs. This work implements the algorithm through a strategy of image partition, executing a sequential reconstruction version (SR) in hardware together to a reconstruction using a queue of pixels on the boundary of the sub-images in software. The purpose of this work is to use inherent hardware implementation properties, like parallelization and high performance, together with the flexibility and quickness of a software design to achieve a final solution that at the same time has a good performance, in view of the limitations of an embedded system, and offers a flexible final solution, allowing the user to choose the image size that would be processed through software settings. The hardware architecture runs at 150 Mhz, uses Avalon protocol to communicate, has a DDR3 memory of 1GB to store temporary images and is connected to an ARM Cortex-A9 through an AMBA® AXI3 bus. The software part runs on this processor at 925 Mhz and has another DDR3 memory of 1GB. The program first configures the internal hardware registers according to the parameters chosen (image size, memory address, and others), commands the hardware to perform the morphological reconstruction algorithm several times in smaller images and then runs the propagation algorithm between the borders of the sub-images processed by the hardware. As a functional verification method for this solution, the resulted images were compared to the ones produced by MATLAB®. The best solution proposed by this work achieved a speedup of around 2x compared to the best theoretical solution possible of the sequential morphological reconstruction algorithm implemented in a hardware that runs at 150Mhz, a speedup of up to 12x in relation to the fast hybrid reconstruction algorithm proposed by Vincent [1] being executed in an ARM® Cortex™A9 processor, and even a speedup of up to 2x in comparison with the same algorithm running in an Intel® Core™ i5 CPU. The final tests to validate this solution show correct results for 8 bits grayscale images of up to 8192x8192 pixels (8k images). To the best knowledge of the author, this is the first hardware/software implementation of the morphological image reconstruction algorithm described in the literature.

Table of Contents

LIST OF FIGURES iv

LIST OF TABLES vi

1 Introduction 1
1.1 Background 1
1.2 Thesis Statement 3
1.3 Objectives 4
1.4 Thesis outline 4

2 Theoretical Background 5
2.1 Morphological Image Reconstruction Algorithm 5
2.2 Implementation Strategies 7
2.2.1 Standard Reconstruction 7
2.2.2 Sequential Reconstruction 8
2.2.3 Reconstruction Using a Queue 9
2.2.4 Fast Hybrid Reconstruction 11
2.2.5 Downhill Filter Reconstruction 11
2.2.6 Directional Propagation Reconstruction 14
2.2.7 Implementation Strategy Used in This Work 14
2.3 SoC FPGAs 16
2.4 Altera Cyclone V System-on-Chip (SoC) FPGA 18
2.4.1 ARM Cortex A-9 19
2.4.2 HPS to FPGA Bridge and memory map 20
2.5 SoCKit development board 22
2.6 Related Works 24

3 Morphological Reconstruction Codesign 26
3.1 Proposed Solution 26
3.2 System Architecture 31
3.3 Hardware Development 33
3.3.1 Controller Submodule 34
3.3.2 Controller Programmable Internal Registers 37
3.3.3 Raster&Anti-Raster Module Interface 39

3.3.4 Hardware Operation 39
3.3.5 Hardware Features and Specification 42
3.4 Software Development 42
3.4.1 Reading and Writing Procedures 44
3.4.2 Time Measurement using Private Internal Timers 45

4 System Verification, Testing and Analysis 48
4.1 Introduction..... 48
4.2 Verification Methodology..... 49
4.3 Tests 50
4.4 Analysis 50
4.4.1 Hardware/Software co-design using 1 Hardware Module 56
4.4.2 Performance Impact of Borders Size 57
4.4.3 Performance Impact with Sub-Image Format 59
4.4.4 Hardware/Software Co-Design using 2 Hardware Modules 60
4.5 Best Hardware/Software Co-Design Performance Comparison 62
4.6 Resources Utilization for two Hardware Modules Implementation 65

5 Conclusion 66

REFERENCES 68

APPENDIX 72

I ARM Cortex-A9 Memory Map 73

LIST OF FIGURES

2.1 Structuring element. 6
2.2 Gray scale morphological reconstruction in 1-dimension. Red is the marker image, blue is the mask, and green is the result of morphological reconstruction. Adapted from [2]. 6
2.3 Raster and Anti-Raster scans. 8
2.4 Structuring element, pixel p is being processed and is placed in the center of the figure. The neighborhood of pixel p is represented by pixels from u_0 to u_7. Adapted from [3]. 9
2.5 Comparison between SoC FPGA and separated FPGA plus CPU solution. 17
2.6 Architecture difference between an standard SoC and a SoC FPGA. 17
2.7 Altera SoC FPGA Device Block Diagram extracted from [4]. 18
2.8 HPS Block Diagram and System Interconnect extracted from [4]. 20
2.9 Memory Map for MPU Subsystem (ARM Cortex-A9) adapted from [4]. 21
2.10 SoCKit development Board block diagram. Extracted from [5]. 22
2.11 SoCKit development Board (Top view). Extracted from [5]. 23
2.12 SoCKit development Board (Bottom view). Extracted from [5]. 23

3.1 Image partitioned in 16 sub-images with border propagation highlighted in red, the sub-images are not necessarily square. Adapted from [6]. 27
3.2 Hardware partitioning strategy using one and two modules in its first and second iterations. 30
3.3 System architecture with HW/SW partitioning. 31
3.4 Hardware Architecture. 34
3.5 Hardware Controller Architecture. 35
3.6 General Finite State Machine. 36
3.7 Main Finite State Machine. 37
3.8 Hardware configuration example running in a 1024x1024 pixels image, sub-blocks of 256x256 pixels, and overlapping border of 16. 40
3.9 Hardware configuration example running in a 1152x1152 pixels image, sub-blocks of 256x256 pixels, and overlapping border of 16. 41
3.10 HPS to FPGA Bridge Block diagram, adapted from [7]. 43
3.11 Private Timer Control Registers bit assignment, adapted from [8]. 46

4.1 4K marker images used for testing. 51

4.2 4K mask images used for testing. 52
4.3 4K output image results after morphological reconstruction tests. 53
4.4 8K images used for testing. 54
4.5 Two Hardwares running in a image divided in 16 sub-images. 60
4.6 Best Hardware/Software Co-Design solution graph comparison in seconds. 63

LIST OF TABLES

2.1 Memory Map for HPS-to-FPGA bridge of MPU Subsystem. ... 21
2.2 5CSXFC6D6F31 resources. ... 24
2.3 Related Works. ... 25

3.1 Register map for controller hardware. ... 38
3.2 Memory map for raster&anti-raster writing interface. ... 40
3.3 Memory map for raster&anti-raster reading interface. ... 40
3.4 Register configuration example. ... 41
3.5 Memory Map reduced for HPS-to-FPGA bridge of MPU Subsystem. ... 43
3.6 Private Timer registers, adapted from [8]. ... 45
3.7 Private Timer Control Registers, adapted from [8]. ... 46

4.1 Theoretical time estimation for SR algorithm implemented in hardware at 150 MHz. 55
4.2 Pure software time processing in the ARM Cortex-A9 and a CPU in seconds. ... 55
4.3 Hardware/Software time processing in seconds. ... 57
4.4 Hardware/Software borders size influence in time processing in seconds. ... 58
4.5 Hardware/Software sub-image format influence in time processing in seconds. ... 59
4.6 Best 1 hardware solution compared to 2 modules in seconds. ... 61
4.7 Best Hardware/Software Co-Design solution compared to only hardware and only software implementation. ... 62
4.8 Best Hardware/Software Co-Design solution compared to only hardware and only software implementation. ... 64
4.9 Hardware resources utilization summary for two hardwares architecture. ... 65

I.1 Memory Map for Peripheral Region of MPU Subsystem. ... 73
I.2 Memory Map for MPU Register Space, located between 0xFFFEC000 and 0xFF-FECFFF. ... 75

ACRONYMS

ADC	Analog-to-Digital Converter
AMBA®	ARM Advanced Microcontroller Bus Architecture
ARM®	Acorn RISC Machines
ASIC	Application-Specific Integrated Circuit
ASSP	Application-Specific Standard Product
AXI	Advanced Extensible Interface
CPU	Central Processing Unit
DAC	Digital-to-Analog Converter
DDR	Double Data Rate
DSP	Digital Signal Processor
FH	Fast Hybrid
FIFO	First-In First-Out
FPGA	Field-Programmable Gate Array
FSM	Finite State Machine
GB	Giga Byte
GPU	Graphics Processing Unit
HDL	Hardware Description Language
HPS	Hard Processor System
HSSI	High-Speed Serial Interface
IC	Integrated Circuits
KB	Kilo Byte
LUT	Look-Up Table
MB	Mega Byte
MCU	Microprocessor Control Unit
MPU	Microprocessor Unit
OS	Operating System
PC	Personal Computer
PCI-e	Peripheral Component Interconnect Express
PLL	Phased-Locked Loop
RAM	Random Access Memory
ROM	Read Only Memory
VHDL	VHSIC Hardware Description Language

VHSIC	Very-High Speed Integrated Circuit
SDRAM	Synchronous Dynamic Random Access Memory
SoC	System-on-Chip
SR	Sequential Reconstruction
UART	Universal Asynchronous Receiver-Transmitter
USB	Universal Serial Bus

Chapter 1

Introduction

1.1 Background

The invention of photography enabled experimental results to be documented objectively instead of being recorded by verbal description and manual drawings. Generally, images were used only for documentation, qualitative description and illustration of the phenomena observed [9]. Nowadays, images can also be used for quantitative evaluation and processing since the digital images have been popularized by the technology revolution, taking advantage of the rapid progress in video and computer technology. The technology is now available to almost any person and digital image processing has become more and more required in any electronic device.

Generally speaking, image processing is a domain of knowledge that performs operations on an image to get or to extract some useful information from it. The field of digital image processing means any method of processing an image by a digital device (a computer, camera, smartphone, tablet and so on). Many authors have different classifications for the same algorithms. There are no agreements where image processing stops and others related areas, like computer vision and image analysis start. The focus of this thesis is on a specific algorithm that can be classified as part of image analysis, which in turn can be classified between image processing and computer vision, either one or another [10].

Mathematical morphology is a theory for analysis and processing spatial structures in images and may be used to design tools for extracting image components that are useful in representation and description of region shapes, such as boundaries, skeleton, and the convex hull [9, 10]. It is called morphology because it aims at analyzing the shape and form of objects. It is mathematical because the analysis is based on set theory, integral geometry, and lattice algebra [11]. It is the foundation of the mathematical set of tools underlying the development of techniques that extract "meaning" from an image [10, 12].

In mathematical morphology, there is an essential set of operations constituted by geodesic transformations. They are particularly useful in image processing, where transformations may have to be performed conditionally to a restriction of the spatial domain. The approach taken

with geodesic transformations is to consider two input images, a marker and a mask, instead of one input image with specific structuring elements like the most of morphological transformations [11].

As a part of the set of geodesic image operators, we have the reconstruction [1]. A morphological reconstruction is an advanced approach to image analysis. It is a useful method for extracting meaningful information about shapes in an image. The forms could be almost anything: letters in a scanned text document, fluorescently stained cell nuclei, or galaxies in a far-infrared telescope image [13]. We can find many examples of morphological reconstruction applications in literature such as filtering, segmentation, and feature extraction [1], image and video compression [14], remote sensing [15], high-resolution satellite image analysis [16, 17], biomedical image analysis [18], and many others medical applications [19, 3].

Essentially, the morphological reconstruction is a generalization of flood-filling. It processes one image, called the marker, based on the characteristics of another image, called the mask. The high points (peaks) in the marker image, specify where processing begins. The peaks spread out, or dilate, while being forced to fit within the mask image. The spreading processing continues until the image values stop changing, reaching the stability [10, 13]. However, its computation can takes too much time for some input data [20].

The process of choosing the right embedded system platform to implement an algorithm like the image morphological reconstruction depends on several factors. What should be taken into consideration is time to market, performance, cost, development ease, power and feature flexibility [21]. The design engineers make significant efforts to reduce cost and power consumption while improving performance and flexibility. However, they find a wide variety of available technologies, all of them claiming to be the best choice for a certain application. There are the microcontrollers MCU (Microprocessor Control Unit), ASSPs (Application-Specific Standard Product), GPPs/RISCs (General Purpose Processor/Reduced Instruction Set Computer), FPGAs (Field Programmable Gate Array), ASICs (Application Specific Integrated Circuit), DSPs (Digital Signal Processor) and reconfigurable processors as options to development an embedded solution [21]. So, how to know which one will solve the demands of the designer meeting all the requirements specified?

In order to implement the morphological reconstruction algorithm in embedded systems, three possible solutions were taken into consideration: pure software implementation, pure hardware implementation, or a hybrid solution with a hardware/software co-design implementation. Normally, the first solution is the fastest to implement and the most flexible one, but since the morphological reconstruction algorithm is highly data dependent, it requires high-speed processors that are hard to find in embedded solutions. The pure hardware design is optimal for the best algorithm performance, but it leads to a time-consuming development and a non-expandable solution. A hardware/software co-design solution can accelerate the calculation of intensive tasks while preserving flexibility to evolve the implementation eventually to better versions while maintaining the already developed main core [22].

The hardware platform chosen was FPGA due to having a good performance x flexibility trade-

off, and the software part could be executed in a RISC processor with a frequency operation that would not compromise the hardware/software approach. At the end, we opted to use the SoCKit Development Kit from ARROW/Terasic with a Cyclone V FPGA embedded with a 925Mhz ARM Cortex-A9 in the same chip, providing an AMBA AXI3 interface acting as a high performance bridge communicator between the processor and the FPGA side, making the co-design solution not be bottlenecked by any parts.

In the scope of morphological reconstruction algorithms, there has been found many target implementations in literature, most of them focused only on software perspective, running at GPUs [20, 23, 2, 24], high-speed processors like Intel Xeon Phi [3, 25, 26] or CPUs [27] and Hybrid GPU-CPU [6, 28]. Just a few ones were implemented purely in hardware using FPGA [29, 30, 31]. To the best knowledge of the author, this is the first hardware/software implementation of the morphological reconstruction described in the literature.

1.2 Thesis Statement

A morphological image reconstruction algorithm has already been developed in VHDL by the Mechatronic Systems Ph.D. student Oscar Eduardo Anacona Mosquera from the Department of Mechanical Engineering of the University of Brasilia at the moment that this research started [30]. The architecture proposed by [30] performs a sequential reconstruction algorithm (SR) using only hardware approach and works perfectly for images of up to 288×288 pixels at the beginning of this work, and up to 512×512 nowadays. The solution was implemented and validated in a Cyclone IV FPGA.

The aim of this work is to improve this solution offering new approaches to the problem, being able to process images bigger than 288×288 pixels in a flexible solution that at the same time works for bigger pictures and smaller ones. This can be done by partitioning an image into small pieces, in a way that the size of each part could be less than the maximum of 288×288 supported by hardware. The final solution is mainly targeted in performance, maintaining the flexibility feature where is viable.

The hardware architecture performs a sequential morphological reconstruction algorithm. A software final processing to solve the border pixel propagation between these small images is executed using a reconstruction with a queue of pixels. Therefore, the proposed solution is an adapted version of the hybrid morphological image reconstruction algorithm, merging a sequential reconstruction in hardware with reconstruction using a queue of pixels in software. The flexibility can be achieved by inserting configurable registers that can be written via software and rules the way that the whole architecture works.

Making the hardware configurable through software brings many possibilities to the work. It is expected that the research could implement and compare different approaches to the problem by analysis of the tests performed after validation with different parameters.

1.3 Objectives

The purpose of this work is to create new features in the already implemented architecture. The general and specific objectives of this works are:

General objective: To create a state of the art solution capable of performing a morphological reconstruction algorithm in images bigger than the capacity of the original solution, 288x288.

Specific objectives:

- To implement an image partitioning strategy that combines a sequential reconstruction in hardware for small images, through raster and anti-raster scans, together with the reconstruction using a queue in software for the boundary pixels between those images.
- To design a hardware architecture in FPGA that controls the original hardware acting as a master ordering it to do the sequential reconstruction algorithm in smaller fractions of bigger images.
- To develop a software that performs the final part of the morphological reconstruction algorithm using a queue of pixels, a FIFO, in the ARM processor to propagate the boundary pixels between the smaller images processed by hardware.
- To develop a hardware solution that is configurable via software. The parameters that should be selectable are the size of the image to be processed, the size of the smaller blocks that will do the sequential morphological reconstruction, the memory addresses for marker and mask images, and the processing order of the blocks.

1.4 Thesis outline

The chapter 2 presents a theoretical background regarding the morphological reconstruction algorithm developed in this thesis and some information regarding reconfigurable hardware devices, FPGA, and SoC FPGA solutions. This chapter also explains, with minor details, all necessary information regarding the Cyclone V SoC Altera's FPGA used to implement the solution proposed by this thesis. Chapter 3 provides all the system features and steps for hardware and software development. The system verification, tests that were performed and analysis of the results are discussed in chapter 4, followed by conclusion in chapter 5. The appendix contains all the complementary material.

Chapter 2

Theoretical Background

This chapter intends to clarify the theory behind morphological reconstruction algorithms, providing to the reader all the theoretical foundation necessary for a deep understanding of the particularities of the problem. Moreover, this chapter provides meaningful information regarding reconfigurable hardware, FPGAs and SoC FPGAs solution.

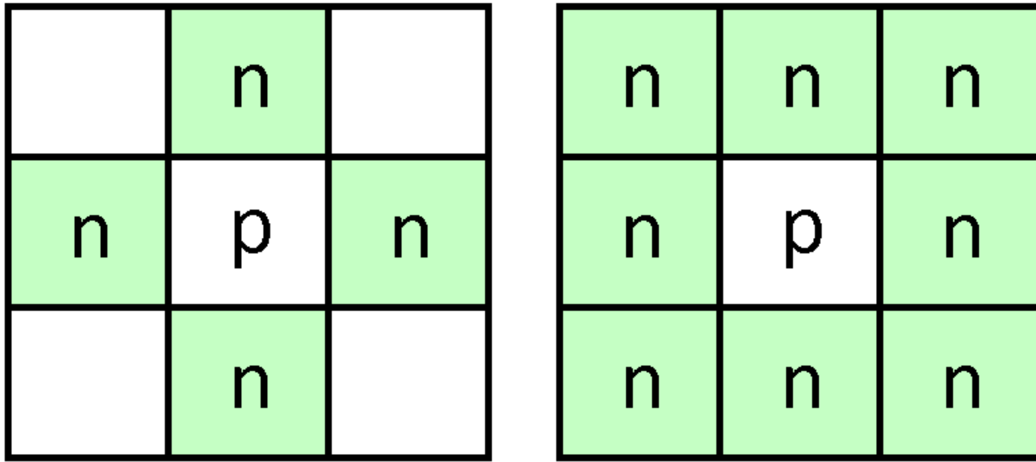
2.1 Morphological Image Reconstruction Algorithm

Morphological reconstruction algorithms are part of a mathematical theory for image processing called mathematical morphology [11, 12]. Mathematical morphology is a nonlinear approach for analyzing and processing geometrical structures. It is mainly based in extracting from an unknown image its geometry through transformations using another image completely defined. In other words, it extracts information associated with the geometry of objects and the topology of an unknown set through transformations using another well-defined set, called structuring element.

A structuring element is a matrix that defines the neighborhood used in the processing of each pixel and identifies the exact pixel in the image being treated. The center pixel of the structuring element identifies the pixel in the image being processed [32]. The two most common structuring elements are the 4-connected and 8-connected sets, N_4 and N_8 [33]. Figure 2.1 shows the basic structuring elements for 4-connected and 8-connected sets, the pixels n are neighbor to p because they are part of the same connectivity.

Lets define some notation used in this work. A grayscale image I is a mapping from a finite rectangular subset D_I of the discrete plane $\mathbb{Z}^2$ into a discrete set (0, 1, 2, . . , *N* - 1) of gray levels. *N* is the grayscale tonality, for an 8-bit representation of the grayscale level can assume values ranging from 0 to 255. The discrete grid $G \subset \mathbb{Z}^2 \times \mathbb{Z}^2$ provides the neighborhood relationships between pixels: n is a neighbor of p if and only if $(p, n) \in G$. N_G is the set of neighbor n of a pixel $p \in \mathbb{Z}^2$ for a grid G, as shown in Equation 2.1.

$$N_G(p) = \{n \in \mathbb{Z}^2 | (p, n) \in G\} \tag{2.1}$$



(a) 4-connected structuring element. (b) 8-connected structuring element.

Figure 2.1: Structuring element.

In morphological reconstruction, given two binary or grayscale images J and I, such that $J \leq I$ (i.e., for each pixel $p \in D_I$, $J(p) \leq I(p)$), the morphological reconstruction $R_I(J)$ of I in J is obtained by successive dilations of J using I as threshold until stability is reached, as shown in Equation 2.2 and Figure 2.2. The image I is the so called mask image and J is the marker. The mathematical symbol $\wedge$ stands for the pointwise minimum and $\delta_G(J)$ is the dilation of J [12, 1].

$$J \leftarrow \delta_G(J) \wedge I \tag{2.2}$$

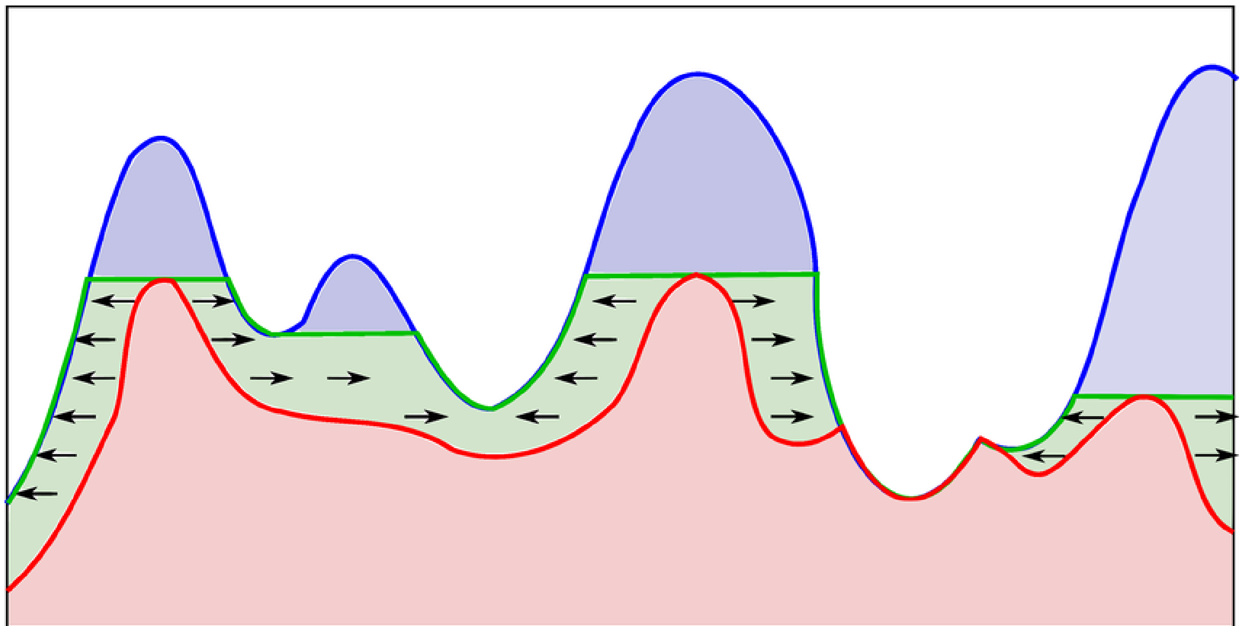


Figure 2.2: Gray scale morphological reconstruction in 1-dimension. Red is the marker image, blue is the mask, and green is the result of morphological reconstruction. Adapted from [2].

Based on Equation 2.2, it is possible to obtain Algorithm 1, that performs the morphological

reconstruction. This algorithm shows the basic implementation for reconstruction by successive propagations being conducted in each iteration. It verifies the neighborhood of each pixel p from the marker image and compares with each equivalent pixel p in mask image (lines 4, 5 and 6). If there is a valid attribution in line 5, it means that one propagation has happened, we can call it as a dilatation step. This procedure continues until the reconstructed image J does not have any pixel value modification, which means that the stability was reached (lines 1 and 7). This algorithm is simple in its construction but is very inefficient due to this irregularity of its updated values [3].

Algorithm 1 Morphological Reconstruction [1]

Input: mask I, binary or grayscale image
Input: marker J, binary or grayscale image
$\forall p \in D_I, J \leq I$
Output: J, reconstructed binary or grayscale image

1: **repeat**
2: $J_{temp} \leftarrow J$
3: Dilatation step:
4: **for all** pixel $p \in J_{temp}$ **do**
5: $J(p) \leftarrow min\{max\{J_{temp}(p), N_G(p)\}, I(p)\}$
6: **end for**
7: **until** stability is reached (i.e., no more pixel value modifications)

2.2 Implementation Strategies

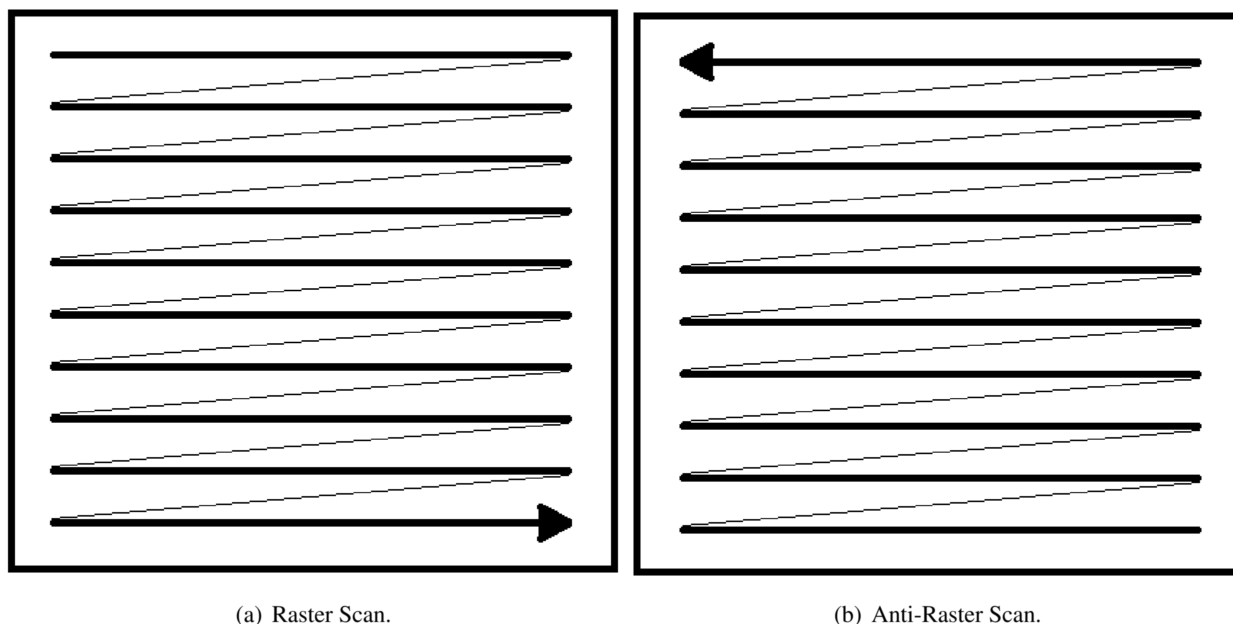
A number of morphological reconstruction algorithms have already been developed and classified as standard reconstruction, sequential reconstruction, reconstruction using a queue of pixels (propagation method), hybrid reconstruction, downhill filters and reconstruction using directional propagation [1, 27, 34]. This section explains those algorithms found in the literature.

2.2.1 Standard Reconstruction

Standard reconstruction [1] is computed directly from the definition in Equation 2.2 and can be implemented by the Algorithm 1. The image pixels can be scanned in an arbitrary order so that the implementation of this algorithm on a parallel machine is straightforward and efficient. However, it requires many complete iterations by image scanning, sometimes several hundred [1]. It is therefore not suited to conventional computers, where its execution time is often of several minutes. This implementation purely in hardware depends on the amount of memory available on the project, and as explained later, usually is not so much in embedded systems.

2.2.2 Sequential Reconstruction

Sequential reconstruction [1] was suggested to reduce the number of iterations in Algorithm 1. A marker image is scanned in two predefined directions, and the result from the scanning process is stored in the same image being processed, the marker image. In other words, the marker image is scanned in a raster direction and an anti-raster direction. At the same time, the marker image is dilated under a mask image. These two scans are repeated for the marker image until convergence, which occurs when there are no more pixel value modifications.



(a) Raster Scan. (b) Anti-Raster Scan.

Figure 2.3: Raster and Anti-Raster scans.

Figure 2.3 shows the directions for raster and anti-raster scans. Each scan is going to use a subset from the set of neighborhood n of a pixel $p \in \mathbb{Z}^2$ to performs the algorithm computation. This subset depends on the structuring element chosen to implement the algorithm (Figure 2.1).

Using an 8-grid structuring element, like the one in Figure 2.4, we can implement the sequential reconstruction with the Algorithm 2. The pixel propagation is performed in two different scans, the raster scan (lines from 3 to 6), and anti-raster scan (lines from 7 to 10). A complete iteration in this algorithm requires at least two scans to be executed, a raster and anti-raster scan.

The main difference between both scan orders is the neighbor pixels that are propagated. Pixels from $u0$ to $u3$ are propagated in raster order, and pixels from $u4$ to $u7$ in anti-raster order. When no more pixel is modified during both raster and anti-raster scans, the algorithm stops the processing, and the output is delivered in the J image (lines 1 and 11).

The main disadvantage of this approach is that after the first two scannings, only a few pixels are modified [1]. Also, when the image size is considerably big, any additional iteration has a high-performance cost, since for each iteration the whole image will be scanned at least twice.

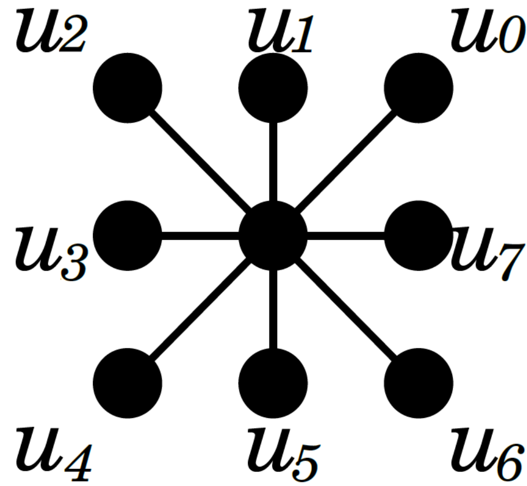


Figure 2.4: Structuring element, pixel p is being processed and is placed in the center of the figure. The neighborhood of pixel p is represented by pixels from u_0 to u_7. Adapted from [3].

2.2.3 Reconstruction Using a Queue

Reconstruction using a queue [1] relies on the use of a First In First Out (FIFO) queue. The FIFO is initialized with all boundary pixels of the local maxima from a marker image in advance. All the neighbors of each pixel in the queue are examined and their values are changed using a dilation process. Next, the first pixels put into the queue are removed and all the dilated boundary pixels are then added to the end of the queue to be processed later [1, 34]. This process continues until the queue is empty.

This algorithm is extremely efficient and simple since, after the initialization of the queue, only the relevant pixels are considered [1]. The main disadvantage of this approach is exactly the initialization of the queue when it needs to determinate all the boundary pixels of the regional maxima from the marker image.

A local maximum in grayscale images can be defined by the following equation [1]:

$$R(I)(p) = \begin{cases} I(p), \text{ if } p \text{ belongs to a maximum,} \\ 0, \text{ otherwise} \end{cases} \tag{2.3}$$

Using the notion of grayscale regional maxima from the Equation 2.3, a reconstruction using queue can be implemented with the Algorithm 3 [1]. The regional maxima computing is done in line 1, with this result the queue can be initialized with all the regional maxima form the image (lines from 2 to 6). After the queue initialization, the propagation can be executed (the dilatation step, lines from 7 to 17). The if condition in line 12 analyses if a propagation is going to happen. If so, a propagation is executed in line 13, and the propagated pixel q is inserted again in the queue to be lastly processed (line 14). The while loop (lines 8 to 17) finishes only when the FIFO queue is empty.

Algorithm 2 Sequential Reconstruction [1]

Input: mask I, binary or grayscale image
Input: marker J, binary or grayscale image
$\forall p \in D_I, J \leq I$
Output: J, reconstructed binary or grayscale image

1: **repeat**
2: $J_{temp} \leftarrow J$
3: Scan J_{temp} in raster order:
4: **for all** pixel $p \in J_{temp}$ being processed in raster order **do**
5: $J(p) \leftarrow min\{max\{J_{temp}(p), J_{temp}(u0), J_{temp}(u1), J_{temp}(u2), J_{temp}(u3)\}, I(p)\};$
6: **end for**
7: Scan J_{temp} in anti-raster order:
8: **for all** pixel $p \in J_{temp}$ being processed in anti-raster order **do**
9: $J(p) \leftarrow min\{max\{J_{temp}(p), J_{temp}(u4), J_{temp}(u5), J_{temp}(u6), J_{temp}(u7)\}, I(p)\};$
10: **end for**
11: **until** stability is reached (i.e., no more pixel value modifications)

Algorithm 3 Reconstruction using a queue [1]

Input: mask I, binary or grayscale image
Input: marker J, binary or grayscale image
$\forall p \in D_I, J \leq I$
Output: J, reconstructed binary or grayscale image

1: $J \leftarrow R(J)$; {Compute regional maxima of J}
2: **for all** pixel $p \in J$ **do**
3: **if** $J(p) \neq 0$ and $\exists q \in N_G(p), J(q) = 0$ **then**
4: $fifo_write(p)$; {Initialization of the queue with boundaries of maxima}
5: **end if**
6: **end for**
7: Propagation:
8: **while** $fifo_empty() = false$ **do**
9: $p \leftarrow fifo_read()$;
10: **for all** pixel $q \in N_G$ **do**
11: Look if q is lower than p and if it is necessary to propagate it:
12: **if** $J(q) < J(p)$ and $I(q) \neq J(q)$ **then**
13: $J(q) \leftarrow min\{J(p), I(q)\}$; {Propagate pixel q}
14: $fifo_write(q)$; {Insert the propagated pixel q into the end of the queue}
15: **end if**
16: **end for**
17: **end while**

2.2.4 Fast Hybrid Reconstruction

The fast hybrid reconstruction algorithm is designed to use a mixture between the already explained sequential reconstruction and reconstruction using a queue, sections 2.2.2 and 2.2.3. It combines the more efficient techniques from each approach in a way that the performance could be better than the utilization of any method individually. As explained in section 2.2.2, while sequential reconstruction requires several additional scannings after the first two in which only a few pixels are modified, reconstruction using a queue does not have this problem. However, reconstruction using a queue is slowed down by the initial determination of the regional maxima of the marker image [1], whereas sequential reconstruction does not have this problem.

Therefore, these two algorithms have complementary drawbacks and advantages. Using sequential scan to initialize the queue and reconstruction using a queue in the late process gives an optimal solution for the reconstruction algorithm [1]. This is a very well known solution and was implemented many times in literature [1, 13, 26, 6]. The Matlab® function *imreconstruct* uses the fast hybrid reconstruction as well [13, 10], and the results given by this function can be compared to the ones found by this thesis as a system verification and validation process.

The algorithm idea is to start with the first two scannings of the sequential reconstruction. During the second one (anti-raster), every pixel p such that its current value could still be propagated during the next raster scanning is put into the queue. The last step of the algorithm is the same as the propagation step of the FIFO algorithm explained in section 2.2.3. However, the number of pixels to be considered during this step is considerably smaller than previously [1].

A fast hybrid reconstruction implementation is described in Algorithm 4. The raster scan (lines from 2 to 5) works the same way as Algorithm 2. The anti-raster scan (lines from 6 to 12) is also very similar, but with an addition that now this part needs to store the pixels that could still be propagated into the queue (lines 9 and 10). In this part, $N_G^-(p)$ is the negative neighborhood of the pixel p that is being processed, i.e., pixels from $u4$ to $u7$ showed in Figure 2.4. The last part is performed by the propagation step, a while loop (lines from 13 to 22) that removes one pixel from the FIFO and process it (line 15), if these pixels would be propagated (line 18) then the propagation is executed (line 19) and the same pixel is inserted at the end of the queue (line 20).

2.2.5 Downhill Filter Reconstruction

Downhill filter reconstruction [27] improves reconstruction using a queue (section 2.2.3), in which some pixels are scanned more than once during the propagation step, by utilizing a multi-queue with regional maxima from a marker image in the order of their intensities. Instead of a FIFO queue, the proposed method implements a random access queue to allow the process of dilation could be executed in an optimal order and guarantees that every pixel is only processed once.

First, a multi-queue is initialized using all the boundary pixels from the regional maxima, which are also sorted in the order of their intensity. Next, all the neighbors of each pixel in the multi-

Algorithm 4 Fast hybrid reconstruction [1]

Input: mask I, binary or grayscale image

Input: marker J, binary or grayscale image

$\forall p \in D_I, J \leq I$

Output: J, reconstructed binary or grayscale image

1: $J_{temp} \leftarrow J$
2: Scan J_{temp} in raster order:
3: **for all** pixel $p \in J_{temp}$ being processed in raster order **do**
4: $J(p) \leftarrow min\{max\{J_{temp}(p), J_{temp}(u0), J_{temp}(u1), J_{temp}(u2), J_{temp}(u3)\}, I(p)\};$
5: **end for**
6: Scan J_{temp} in anti-raster order:
7: **for all** pixel $p \in J_{temp}$ being processed in anti-raster order **do**
8: $J(p) \leftarrow min\{max\{J_{temp}(p), J_{temp}(u4), J_{temp}(u5), J_{temp}(u6), J_{temp}(u7)\}, I(p)\};$
9: **if** There exists $q \in N_G^-(p)$ such that:$J(q) < J(p)$ and $J(q) < I(q)$ **then**
10: $fifo_write(p);$
11: **end if**
12: **end for**
13: Propagation:
14: **while** $fifo_empty() = false$ **do**
15: $p \leftarrow fifo_read();$
16: **for all** pixel $q \in N_G$ **do**
17: Look if q is lower than p and if it is necessary to propagate it:
18: **if** $J(q) < J(p)$ and $I(q) \neq J(q)$ **then**
19: $J(q) \leftarrow min\{J(p), I(q)\};$
20: $fifo_write(q);$
21: **end if**
22: **end for**
23: **end while**

queue are examined and their intensities are changed through a dilation process. Finally, the pixels in the multi-queue are removed in the order of their intensity, while all dilated boundary pixels are added to the multi-queue in the order of their intensity. This process continues until the multi-queue is empty [27, 34].

This approach is based on a precondition assumption, reflecting particular requirements of the author's application. The precondition is that each pixel in the marker image is either equal to the corresponding pixel in the mask or it is equal to zero. Also, the images that were tested are very small, only 256×256 pixels, and very restricted to some particular applications.

The proposed method has performed even worse than two of the previous algorithms, sections 2.2.2 and 2.2.4, when a constant mid-grey image is processed with a single pixel at the center of the image used as a marker [27]. As explained by the author, this happens due to the higher initialization overheads involved in downhill filter reconstruction algorithm.

For the reason that this approach uses a very specific set of the morphological algorithm reconstruction application, restricting the marker image content, the algorithm execution time is hard to be compared in a general situation. More images with bigger resolution and different data content should have to be processed in order to get a better conclusion regarding algorithm efficiency.

The downhill filter reconstruction implementation is explained in Algorithm 5. It first begins finding the maximum pixel value inside the marker image (line 2). Then, m queues are created and each non-zero pixel inside the marker image is inserted in its respective queue (lines from 4 to 8). In this part, it is important to note that the attribution showed in line 6 ($L[J(p)]^\frown p$) means that the pixel being processed was concatenated in the correspondent queue related to its own value. At last, the lines from 10 to 21 performs the propagation step, processing the multi-queue from the highest to the lowest values. Each pixel inserted in the queue has an attribute to inform if it was processed or not. This can be represented as a data structure attribute. This is showed in line 15 where the pixel q from the marker image is analyzed to verify if the current pixel has already been processed or not.

Algorithm 5 Downhill filter reconstruction[27]

Input: mask I, binary or grayscale image
Input: marker J, binary or grayscale image
$\forall p \in D_I, J \leq I$
Output: J, reconstructed binary or grayscale image

```
1: Find m, the maximum pixel value in J:
2: m : ℕ | m ∈ J ∧ ∀p : (m ⩾ J(p));
3: Create m queues and place each non-zero pixel in J in its respective queue:
4: for all pixel p ∈ J do
5:     if J(p) ≠ 0 then
6:         L[J(p)] ← L[J(p)]⌢p;
7:     end if
8: end for
9: Process the m queues from high to low:
10: for n = m to n = 1 do
11:     while L[n] ≠ 0 do
12:         p ← pop(L[n]);
13:         for all q ∈ N_G(p) do
14:             if I(q) > 0 and !J(q).finalised then
15:                 J(q) ← min{n, I(q)};
16:                 L[J(q)] ← L[J(q)]⌢q;
17:             end if
18:         end for
19:     end while
20: end for
```

2.2.6 Directional Propagation Reconstruction

The directional morphological reconstruction [34] improves the geodesic dilation in the downhill filter by introducing propagation directions. All of the previous state-of-the-art methods check every neighbor around each pixel in a memory queue for geodesic dilation. The checking frequency for geodesic dilation can be considerably decreased by saving and utilizing the information on the propagation direction that is from the previously reconstructed pixel toward its neighbor. This method exploits the propagation direction of the current pixel obtained from a queue memory, which denotes the relative direction from the previously reconstructed pixel to itself [34].

The directional reconstruct propagation implementation is presented in Algorithm 6. The direction propagation information from a pixel p is stored in the variable d (line 14) using a $Dir(p)$ command. The Dir attribute means a direction map is recording propagation directions. Now the neighborhood of the pixel p is analyzed only using the information from the direction of the propagated pixel (line 16). Writing back the information regarding propagation direction is performed in the 20th line. It should be noted that the other operations in this method are exactly the same as in the downhill filter explained in section 2.2.5.

This approach is based on the same precondition assumption from section 2.2.5. Each pixel in the marker image is either equal to the corresponding pixel in the mask or it is equal to zero. So, we can conclude the same as before, it works well for a very specific set of morphological reconstruction problems but is hard to generalize to any image content.

2.2.7 Implementation Strategy Used in This Work

The implementation technique used in this work is an adapted version of the fast hybrid algorithm. It is an algorithm that has been implemented many times in the literature since 1993 by [1]. The downhill filter reconstruction and directional propagation reconstruction algorithms perform well for a very specific set of problems. This thesis intends to implement a morphological reconstruction algorithm efficiently regardless of the image content.

This MSc dissertation proposes an image partitioning strategy that combines the sequential reconstruction in hardware for small images, together with the reconstruction using a queue in software for the boundary pixels between those images. As stated in section 2.2.2, when the image size is considerably big any additional iteration using sequential reconstruction has a high-performance cost, since for each iteration the whole image will be scanned at least twice, one raster and one anti-raster scan.

The bigger is the image, the higher is the algorithm performance impact using sequential reconstruction. According to [20], the number of iterations using the sequential reconstruction algorithm is strongly related to the size and contents of the image. For small images the performance cost is not so high, an additional iteration during the sequential operation does not heavily impact the whole processing as it happens for big images.

As stated in the section 2.2.3, the reconstruction using a queue is very efficient, since after

Algorithm 6 Directional propagation reconstruction[34]

Input: mask I, binary or grayscale image
Input: marker J, binary or grayscale image
$\forall p \in D_I, J \leq I$
Output: J, reconstructed binary or grayscale image

1: Find m, the maximum pixel value in J:
2: $m : \mathbb{N} \mid m \in J \wedge \forall p : (m \geqslant J(p))$;
3: Create m queues and place each non-zero pixel in J in its respective queue:
4: **for all** pixel $p \in J$ **do**
5:     **if** $J(p) \neq 0$ **then**
6:         $L[J(p)] \leftarrow L[J(p)]^\frown p$;
7:     **end if**
8: **end for**
9: Process the m queues from high to low:
10: **for** $n = m$ to $n = 1$ **do**
11:     **while** $L[n] \neq 0$ **do**
12:         $p \leftarrow pop(L[n])$;
13:         **if** $J(p) = n$ **then**
14:             $d \leftarrow Dir(p)$;
15:         **end if**
16:         **for all** $q \in N_G^d(p)$ **do**
17:             **if** $I(q) > 0$ and $!J(q)$.finalised **then**
18:                 $J(q) \leftarrow min\{n, I(q)\}$;
19:                 $L[J(q)] \leftarrow L[J(q)]^\frown q$;
20:                 $Dir(q) \leftarrow d'$;
21:             **end if**
22:         **end for**
23:     **end while**
24: **end for**

the initialization of the queue, only the relevant pixels are considered [1]. The main disadvantage of using a reconstruction with queue resides in the initialization of the queue with the regional maxima from the marker image. This is not a problem for our approach because the queue initialization uses fixed positioned pixels. Only the ones placed in the border of the partitioned images are inserted in the FIFO queue, it is a static insertion with already known pixels that are always located in the same position.

Therefore, partitioning a big image into small pieces and performing a sequential reconstruction algorithm in each one, together with the reconstruction using a queue in software for the border of those images, seems to be a reasonable strategy for implementing the morphological image reconstruction algorithm.

2.3 SoC FPGAs

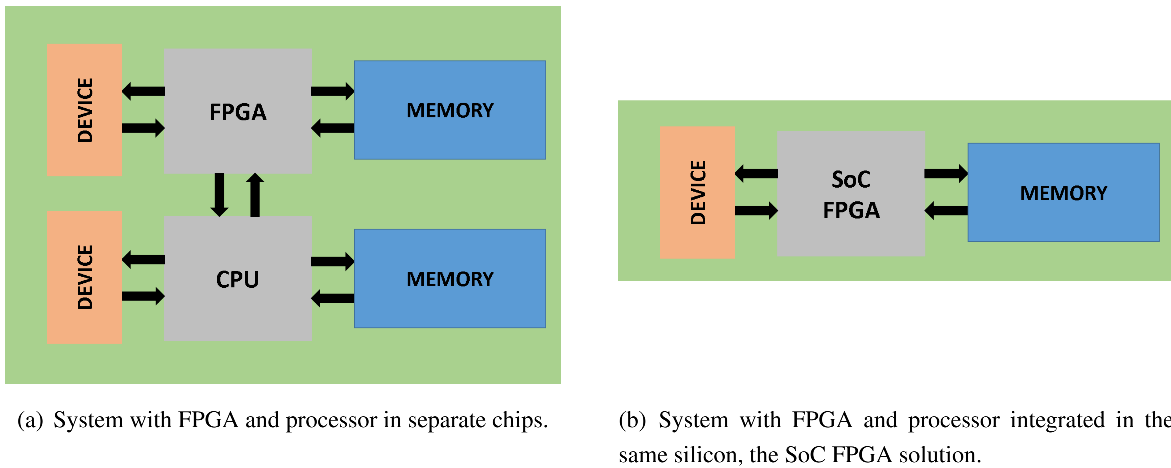
A field-programmable gate array (FPGA) is an integrated circuit designed to be configured by a customer or a designer after manufactured. It is composed by logical elements that can be programmed in order to execute logical functions, store data and to route the signals to other logical cells [35]. The FPGA configuration is generally specified using a hardware description language as Verilog HDL and VHDL. The same language used for an application-specific integrated circuit, or simply ASIC. Circuit diagrams can be used as well to configure the FPGA, as they were used for ASICs before, but this is increasingly rare since the designs are becoming bigger and even more complex.

A system-on-a-chip or system-on-chip (SoC or SOC) is an integrated circuit that integrates all components of a computer or other electronic systems. The implication is that a single silicon chip can be used to implement the functionality of a whole system, rather than several different physical chips being required in a big printed circuit board (PCB) [36]. A SoC may contain analog and digital components together with mixed-signal blocks as analog-to-digital and digital-to-analog converters (ADCs and DACs). Looking at the digital part only, a SoC can combine all aspects of a digital system: processor, high-speed logic, interfacing, memory, and so on. The SoC solution is lower cost, enables faster and more secure data transfer between the various system elements, has higher overall system speed, lower power consumption, smaller physical size, and better reliability [36].

Historically, the differences between a SoC and an FPGA were fairly obvious. Usually, they competed against each other in some applications, but normally the two technologies followed their paths. Early in their evolution, FPGAs were perceived by design engineers as merely configurable logic gates which could often be applied in repetitive operations in low-volume systems that could not justify the higher expense of an ASIC. Recently, the integration of ARM® Cortex®-A cores into FPGAs and compute-intense cores could lead one to believe that the paths of true multicore SoCs and these so-called FPGA SoCs had converged, and suggest that both solutions are on a collision course and are interchangeable [37].

SoC FPGA devices integrate both FPGA architectures and processor into a single device. Consequently, they provide higher integration, smaller board size, lower power, and higher bandwidth communication between the FPGA and processor. Also, they include a rich set of peripherals, high-speed transceivers, on-chip memory, and an FPGA-style logic array [38].

There are many advantages to use SoC FPGAs instead of designs that already use an FPGA and a separate microprocessor or DSP. It is likely to offer comparable, even superior performance and functionality, but at a lower board space, lower power, and lower system cost as much as 50% less as shown in Figure 2.5. As the signals between the FPGA and the CPU now reside on the same silicon, the communication between the two consumes substantially much less power compared to using separate chips. The integration of thousands of internal connections and wires between the FPGA and the processor leads to substantially lower latency and higher bandwidth compared to a two-chip solution which in turn leads to much better performance [38].



(a) System with FPGA and processor in separate chips.

(b) System with FPGA and processor integrated in the same silicon, the SoC FPGA solution.

Figure 2.5: Comparison between SoC FPGA and separated FPGA plus CPU solution.

Figure 2.6 shows the architecture difference between a standard SoC built in an ASIC solution and the SoC FPGA solution [38]. Comparing both, choosing a SoC FPGA brings many advantages, especially in a research project like this Master Thesis. Some of those advantages are:

- No charges or minimum purchase requirements, for a single, SoC FPGA, or millions of devices, cost-effectively;
- Faster development time;
- Lower risk. The SoC FPGA can be reconfigured at any time;
- Adaptable to changing research requirements and standards, supporting for in-field updates and upgrade;
- No additional licensing or royalty payments required for the embedded processor, high-speed transceivers, or other advanced system technology.

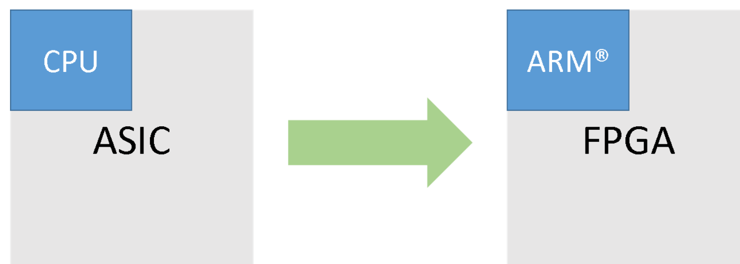


Figure 2.6: Architecture difference between an standard SoC and a SoC FPGA.

Nowadays, there are many vendors in the market offering FPGA solutions including Achronix Semiconductor, QuickLogic, Actel, Intel Altera, AMI Semiconductor, Atmel, Cypress Semiconductor, Lattice Semiconductor, and Xilinx. Each manufacturer offers devices with several sizes in terms of logical elements, embedded memories, and DSP processing elements. The most

popular SoC FPGAs vendors are Xilinx with their Zynq family and Intel Altera with Cyclone V and Arria V SoC FPGAs.

2.4 Altera Cyclone V System-on-Chip (SoC) FPGA

The Cyclone® V system-on-a-chip (SoC) is composed of two distinct portions: a dual-core ARM® Cortex™-A9 hard processor system (HPS) and an FPGA [4]. Figure 2.7 shows a high-level block diagram of the Altera® Cyclone V SoC device.

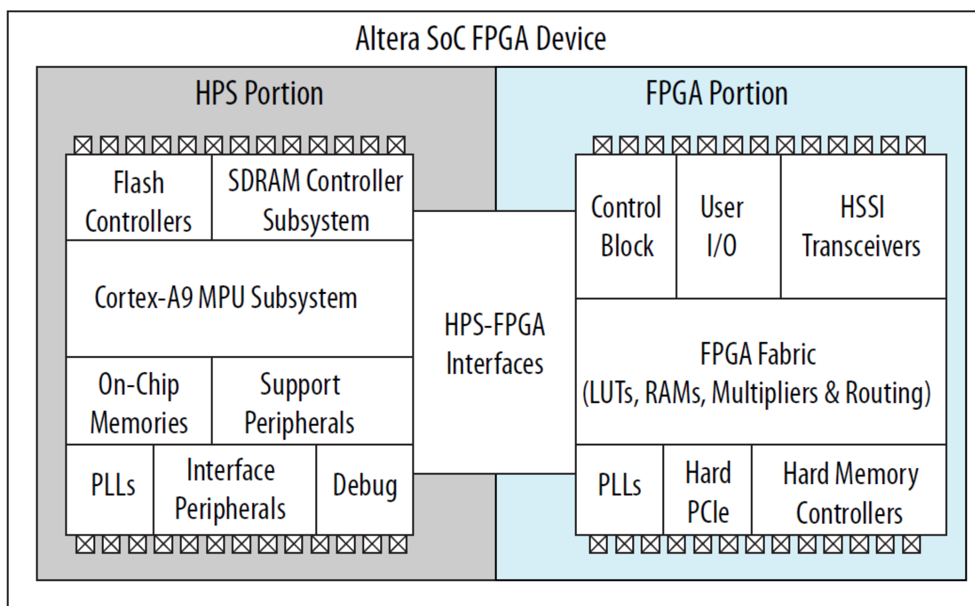


Figure 2.7: Altera SoC FPGA Device Block Diagram extracted from [4].

The HPS is the part that contains the Microprocessor unit (MPU) and other modules as well. As a whole this system consists of the following modules:

- Microprocessor unit (MPU) subsystem with single or dual ARM Cortex-A9 MPCore® processors;
- Flash memory controllers;
- SDRAM controller subsystem;
- System interconnect;
- On-chip memories;
- Support peripherals;
- Interface peripherals;
- Debug capabilities;

- Phase-locked loops (PLL)

The FPGA portion of the device contains:

- FPGA fabric;
- Control block (CB);
- Phase-locked loops (PLL);
- High-speed serial interface (HSSI) transceivers, depending on the device variant;
- Hard PCI Express® (PCI-e) controllers;
- Hard memory controllers

2.4.1 ARM Cortex A-9

The Cortex family of processors provided by ARM offers a range of solutions optimized for specific markets and applications. The ARM Cortex family comprises three series, which all adhere to the ARMv7 architecture and implement the Thumb®-2 instruction set to deliver the highest performance in cost-sensitive embedded markets [39]:

- **ARM Cortex-A Series:** applications processors supporting complex Operating Systems (OS) and multiple user applications;
- **ARM Cortex-R Series:** embedded processors for deeply embedded real-time systems;
- **ARM Cortex-M Series:** embedded processors optimized for very cost sensitive microcontrollers and FPGA

Nowadays ARM processors are the standard in embedded systems. Altera SoC FPGAs is equipped with one of ARM's latest, high-performance Cortex-A9 processor architectures. The Cortex-A9 architecture provides the latest ARM features and capabilities, industry-leading performance, and is widely deployed in products ranging from wireless handsets to tablet computers [40].

The ARM Cortex-A9 MPCore processor in Altera SoC FPGAs is dual-core and is designed for maximum performance and power efficiency, implementing the widely-supported ARMv7 instruction set architecture to address a broad range of industrial, wireless, and automotive applications [40].

For more information regarding all the features of ARM Cortex-A9 processor, refer to the "The ARM Cortex-A9 Processors" white paper [39].

2.4.2 HPS to FPGA Bridge and memory map

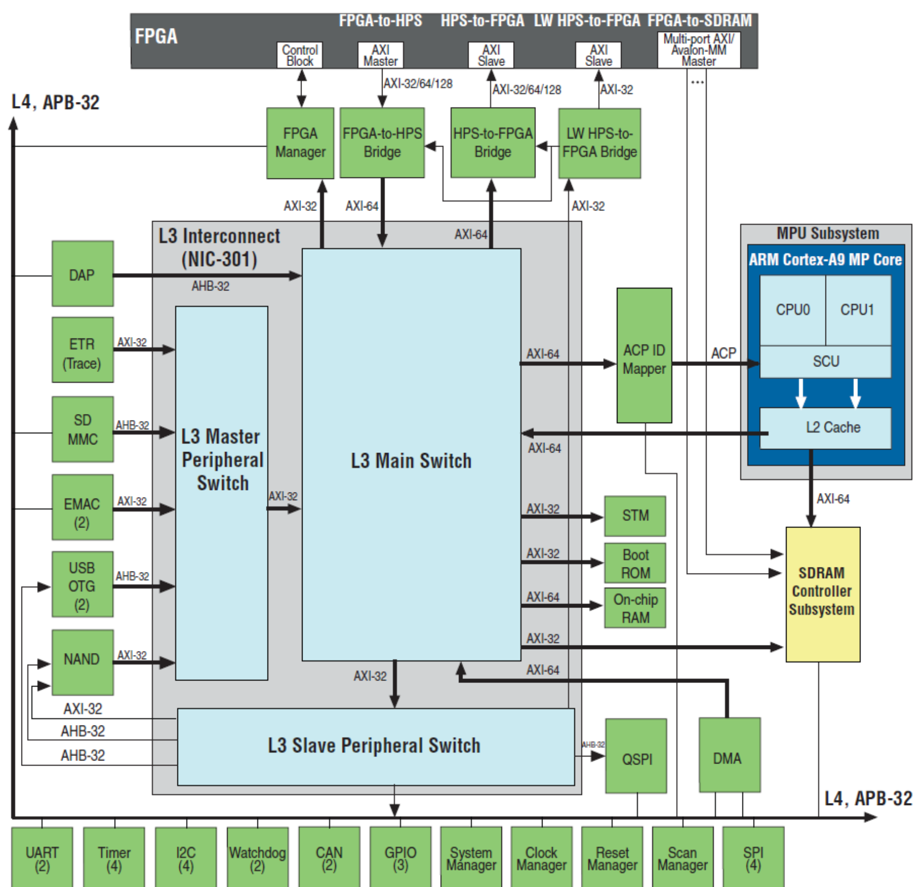


Figure 2.8: HPS Block Diagram and System Interconnect extracted from [4].

Figure 2.8 shows all the internal block diagram and interconnections for the ARM Cortex-A9 processor. It is a complete block diagram, but for this work, we will highlight only the ones that are very important for the developments of this thesis, starting from the bridges between FPGA and ARM processor.

In Altera SoC FPGAs, the HPS logic and FPGA fabric are connected through AXI (Advanced eXtensible Interface) bridge. For HPS processor communicate with FPGA, Altera system integration tool Qsys should be used for the system design to add Altera HPS component. From the AXI master port of the HPS component, the HPS can access those components instantiated in Qsys whose memory-mapped slave ports are connected to the master port.

As observed in Figure 2.8, The bridge between the HPS and FPGA is composed by three interfaces:

- FPGA-to-HPS Bridge - 32, 64 or 128 bits;
- HPS-to-FPGA Bridge - 32, 64 or 128 bits;
- Lightweight (LW) HPS-to-FPGA Bridge - 32 bits;

Figure 2.9 shows the memory map for the MPU Subsystem (ARM Cortex-A9 MP Core) of Figure 2.8. We can see that it has a space reserved for the FPGA slaves using the HPS-to-FPGA bridge. This space address will be used later for the communication between the HPS and FPGA during the morphological reconstruction algorithm implemented on this work.

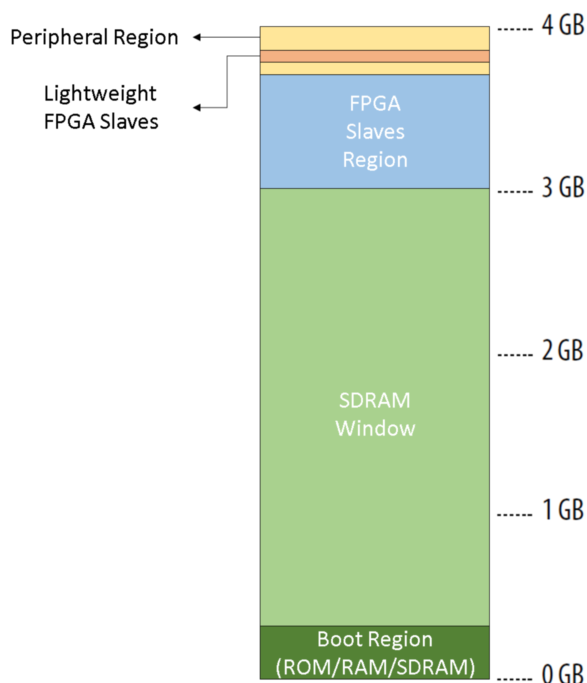


Figure 2.9: Memory Map for MPU Subsystem (ARM Cortex-A9) adapted from [4].

Table 2.1 shows the main addresses inside the peripheral's region used in this work. Two addresses for communication between the ARM and FPGA through the HPS-to-FPGA and Lightweight HPS-to-FPGA bridges and another address for configuration of internal registers inside the ARM, one of these registers is used for controlling an internal timer inside the HPS that act as a monitor for the time consumption of the algorithm. For a complete memory map of all the peripherals in the MPU peripheral's region, refers to the table I.1 in Appendix I.

Table 2.1: Memory Map for HPS-to-FPGA bridge of MPU Subsystem.

Interface	**Name**	**Start Address**	**End Address**	**Size**
HPS2FPGASLAVES	FPGA Slaves Accessed Via HPS-to-FPGA AXI Bridge	0xC0000000	0xFBFFFFFF	960 MB
LWFPGASLAVES	FPGA Slaves Accessed with Lightweight HPS-to-FPGA Bridge	0xFF200000	0xFF3FFFFF	2 MB
MPU	MPU registers	0xFFFEC000	0xFFFEEFFF	8 KB

2.5 SoCKit development board

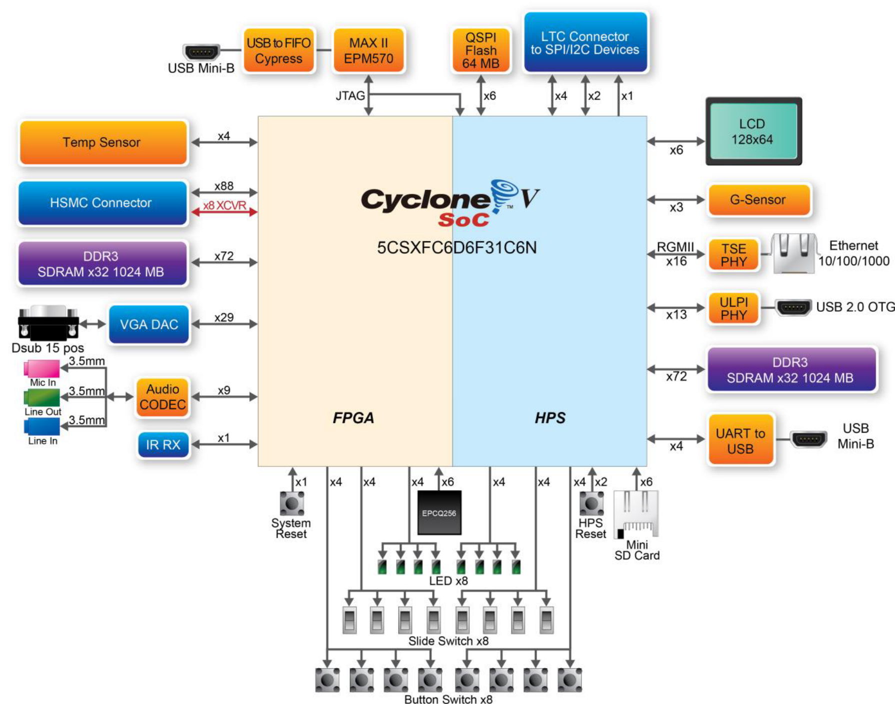


Figure 2.10: SoCKit development Board block diagram. Extracted from [5].

The North-American company Terasic provides an easy way to learn and develop solutions in a SoC: the SoCkit development board. It was designed with the purpose of helping engineers to quickly evaluate the flexibility and operation of Cyclone V SoC system for their projects. Figure 2.11 and Figure 2.12 presents the board top and bottom view and its resources. The Figure 2.10 shows the block diagram and the connections between the board peripherals and Cyclone V SoC chip. Some of them will be briefly described in this section since they are important for this thesis.

- **Cyclone V SoC 5CSXFC6D6F31 Device:** A system that combines an HPS processor, and an FPGA in a single chip;
- **Micro-SD connector:** This connector contains the Micro-SD card with a Linux distribution image file. The Linux Operating System (OS) boots from this device. This OS was provided by Terasic company in its website [1];
- **USB-Blaster II Port:** Used to programming the FPGA fabric with the right bitstream;
- **USB-UART connection:** Used for serial communication between the board and an external computer;
- **Ethernet Port:** Used for Ethernet connection between the board and an external computer;

[1]http://www.terasic.com/downloads/cd-rom/sockit/linux_BSP/SoCKit_RevD_SD.zip

- **FPGA DDR3 1GB:** the External memory connected to FPGA fabric. Used for storing the images being processed during the reconstruction algorithm;
- **HPS DDR3 1GB:** the External memory connected to HPS portion. Used as a memory RAM for the OS running at the HPS;

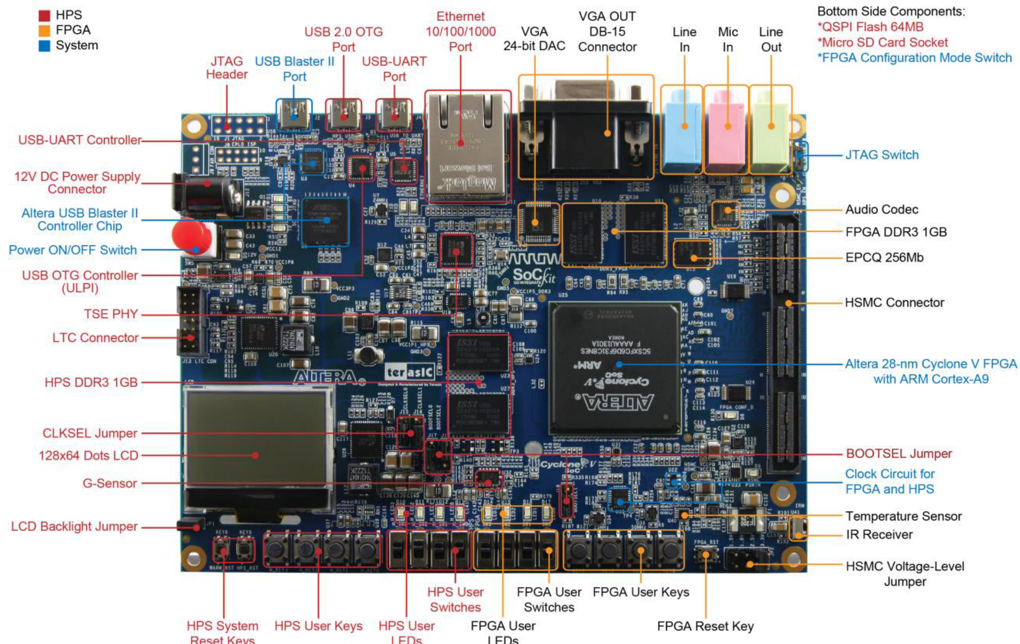


Figure 2.11: SoCKit development Board (Top view). Extracted from [5].

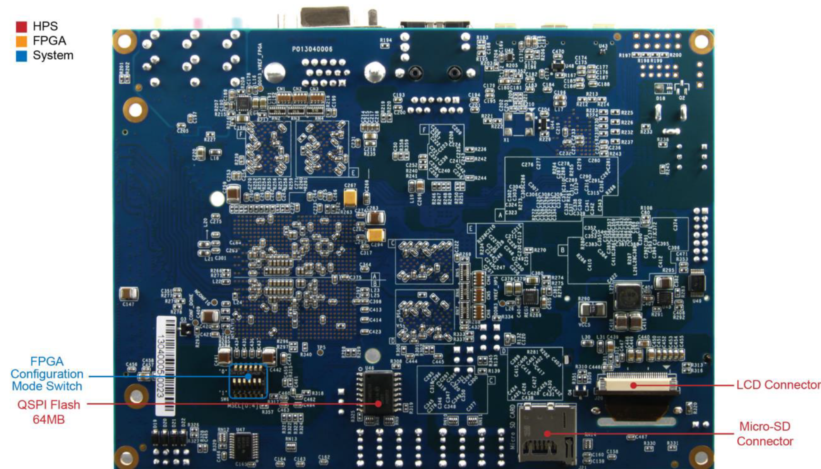


Figure 2.12: SoCKit development Board (Bottom view). Extracted from [5].

The Cyclone V SoC provided in this board has an ARM Cortex-A9 Dual-core of 32-bit architecture embedded in the same silicon as the FPGA. Its resources are very enough for the

implementation purposes of this project, even though it has an explicit limitation on memory size. The most important amount of Cyclone V resources are resumed in Table 2.2.

Table 2.2: 5CSXFC6D6F31 resources.

Cyclone V 5CSXFC6D6F31	
Processor cores (ARM Cortex-A9 MPCore)	Dual
CPU clock rate	925 Mhz
Logical Elements - LEs (K)	110
Adaptive Logic Module - ALM	41,910
M10K memory (Kb)	5,570
DSP blocks	112
HPS PLLs	3
FPGA fractional PLLs	6
HPS hard memory controllers	1
FPGA hard memory controllers	1

2.6 Related Works

In the scope of morphological reconstruction algorithms, there has been found many target implementation in literature. Most of them are focused only on software perspective, running at GPUs [20, 23, 2, 24], high-speed CPUs [3, 25, 26, 27, 34] and Hybrid GPU-CPU [6, 28]. Only a few ones were implemented purely in hardware using FPGA [29, 30, 31]. To the best knowledge of the author, this is the first hardware/software implementation of the morphological reconstruction described in the literature. Table 2.3 shows a summary of the related works found in the literature. This table describes some of the implementation features of those related works. The last line of this table presents the results provided by this MSc thesis.

Considering the related works about morphological reconstruction algorithm implementation, it can be observed that:

- There is no report in the literature about a hardware/software codesign implementation;
- The only three hardware implementations process images of up to 512×512 pixels size;
- Most of the morphological reconstruction algorithm implementation found in literature uses the fast hybrid reconstruction algorithm;
- Most of the previous reports about reconstruction techniques have been addressed to GPUs and/or CPUs based platforms;
- It is an open research area. Most of the related works have been published in the last ten years

Table 2.3: Related Works.

Work	Platform	System Architecture	Algorithm	Maximum Image Size	Year
[2]	GPU Software	Intel Core i7 2.66 GHz CPU and two NVIDIA GPUs (C2070 and GTX580)	Fast Hybrid Reconstruction algorithm	$50K \times 50K$	2012
[20]	GPU Software	Intel Core2 Quad Q6600 2.4 GHz CPU and NVIDIA GeForce GTX 470 GPU	Modified version of Sequential Reconstruction algorithm	3D images of $1030 \times 1030 \times 80$	2011
[24]	GPU Software	Intel Core i7 2.80 GHz CPU and NVIDIA GTX TITAN GPU	GPU Block Asynchronous Reconstruction	560×600	2015
[26]	CPU Software	2 Intel Xeon 8-Core E5-processor 2.66 GHz CPU and one coprocessor Intel Xeon Phi SE10P 1.1GHz	Fast Hybrid Reconstruction algorithm	$16K \times 16K$	2015
[27]	CPU Software	Intel Pentium 4 1.8GHz CPU	Downhill Filter algorithm	3D images of $298 \times 298 \times 60$	2004
[34]	CPU Software	Intel Core i7-2600 3.40GHz CPU	Directional Morphological Reconstruction algorithm	1300×864	2015
[6, 28]	CPU-GPU Hybrid Software	120 nodes in a cluster, each node has an Intel Xeon X5660 2.8 GHz CPU and 3 NVIDIA Tesla M2090 (Fermi) GPUs	Fast Hybrid Reconstruction algorithm	$96K \times 96K$	2013
[29]	FPGA Hardware	Xilinx-Virtex XCV1000E and Altera-Cyclone EP2C35 FPGAs	Standard Reconstruction algorithm	512×512 of 8-bit	2008
[30]	FPGA Hardware	Altera Cyclone IV (EPC4C115F29C7) FPGA	Sequential Reconstruction algorithm	288×288 of 8-bit	2017
[31]	FPGA Hardware	Xilinx Virtex II XC2V2000-BG575-6 FPGA	Standard Reconstruction algorithm	160×120 of 8-bit	2005
This work	SoC FPGA Hardware-Software	Altera Cyclone V SoC (5CSXFC6D6F31) FPGA	Adapted version of Fast Hybrid Reconstruction algorithm	8192×8192 of 8-bit	2018

Chapter 3

Morphological Reconstruction Codesign

This chapter presents the morphological reconstruction codesign solution in detail. The general overview of the proposed solution with the hardware and software development methodology, the whole system architecture, its features, how the algorithm is divided into hardware and software parts and how they are synchronized throughout the solution. Also, this chapter explains with minor details all the internal hardware registers, its function, state machines and programmability, the software features and capability that were developed in this work.

3.1 Proposed Solution

In order to process images bigger than the maximum hardware capacity, 288x288 pixels, the actual solution divides the whole image into smaller parts that can be processed by hardware. This work creates an image partitioning strategy that combines a sequential reconstruction in hardware for small images, together with the reconstruction using a queue in software for the boundary pixels between those images.

Figure 3.1 presents the strategy of image partition proposed by this thesis. In this example, an image is divided into 16 sub-images, each one is processed by hardware using the sequential morphological reconstruction algorithm. The main problem highlighted in this figure is the border propagation issue. As the morphological reconstruction algorithm needs to be processed in the whole image, performing the same algorithm in smaller pieces does not solve the propagation in the entire figure. A final pixel propagation between those sub-images needs to be done to finish the algorithm. This final pixel propagation between the borders can be implemented using the reconstruction with a queue in software.

The algorithm division between hardware and software intends to extract the best from each approach. The implementation proposed by this work in pseudo-code is presented in Algorithms 7 (hardware) and 8 (software). The hardware algorithm's output is the input for the software algorithm, an image almost reconstructed (still missing the border propagation). The output for the software algorithm is the final reconstructed image. Both are running at the same time.

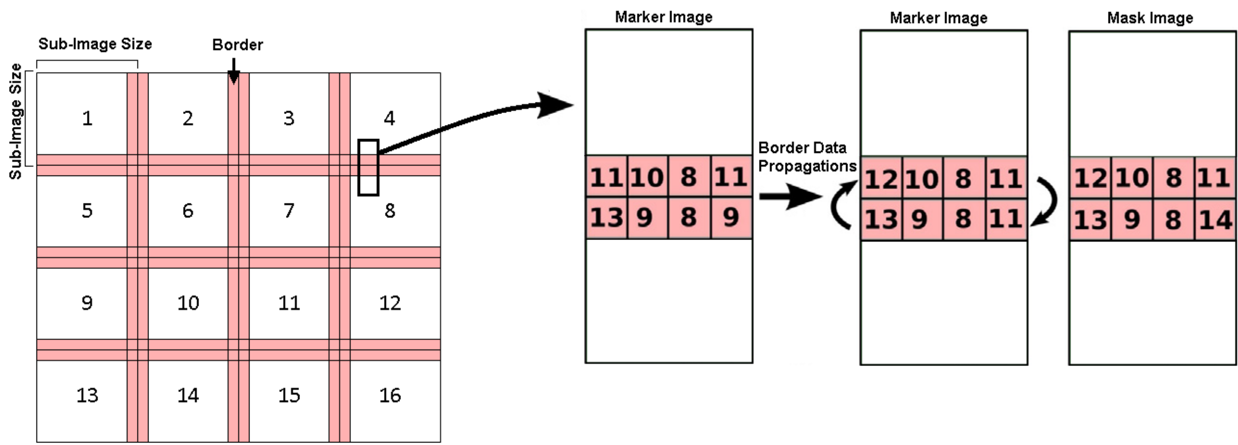


Figure 3.1: Image partitioned in 16 sub-images with border propagation highlighted in red, the sub-images are not necessarily square. Adapted from [6].

The Algorithm 7 is basically the sequential reconstruction algorithm, shown in section 2.2.2. The only difference is the image partition in line 1, where the marker image is partitioned into M sub-images. The sequential reconstruction is performed locally in every sub-image (lines from 4 to 7) as if it was the final image being processes. This makes the output for the hardware part be a partially reconstructed image. Every sub-image was processed using the sequential reconstruction algorithm, but the whole image has not been reconstructed yet, the border propagation is still missing. This last part is done by software.

The Algorithm 8 starts waiting until the hardware has processed at least two sub-images. The main loop (lines from 5 to 21) synchronizes with the hardware algorithm in the if condition (line 6), this allows the software not to overpass the sub-image that is being processed by hardware. In practice, this does not happen, since hardware execution time is faster than software. The lines from 11 to 19 are the same reconstruction using queue (section 2.2.3). The difference is situated in line 15, where the propagation happens only in the region that has already been processed by hardware. This guarantee that the software does not propagate any pixel that has not been processed by hardware. Usually, there are only a few pixels in the borders to be propagated, this can be considered the lightest part of the algorithm. If software tries to propagate the pixel in a region that has not been processed by hardware, the execution time will be very slow, since the software will try to execute the propagation in a region not pre-processed, making the toughest effort of the algorithm (that should be performed by hardware).

The most expensive algorithm cost is due to the sequential morphological reconstruction in smaller images. It is where the majority part of the pixels are propagated. On the other hand, as it is executed in raster and anti-raster scans until stability, it can be said that is a very uniform sequential operation. The algorithm implemented in hardware can take advantage of this uniformity in the memory access.

Since the pixels are adjacent to each other, the memory access can be performed in sequential reading and writing requests, from the beginning of the memory until the end (raster scan), and from the end of the memory until the beginning (anti-raster scan). This part can be executed

Algorithm 7 Sequential reconstruction partitioned in small images (Hardware algorithm)

Input: mask I, grayscale image
Input: marker J, grayscale image
$\forall p \in D_I, J \leq I$
Output: J, partially reconstructed grayscale image (missing the borders)

1: Partition the marker image J into M sub-images; $\{M \subset J\}$
2: **Hardware:**
3: **for** $N = 1$ to $N = M$ **do**
4: **repeat**
5: **Raster** scan in sub-image N;
6: **Anti-raster** scan in sub-image N;
7: **until** stability is reached (i.e., no more pixel value modifications)
8: **end for**

in hardware, taking advantage of burst requests. A single reading request in burst gives a pixel being read in sequence. The data from memory is given each clock after requisition in a sequentially manner, from the initial reading address until the size of the burst. In this situation, a hardware approach has a much better performance than software, especially due to this uniform memory access (pixels are adjacent to each other in the reading and writing operations). In hardware we can reach the processing throughput of 1 pixel per clock and, even with a low frequency, the performance is usually much better than a software approach.

The final part (reconstruction using a queue in software for the borders) is very non-uniform, and the memory access is not easily predictable. Not all the pixels are propagated between the images, and in some cases, there is no propagation between the tiles. As there is no raster neither anti-raster scan, the pixels can be propagated non-uniformly. We cannot take advantage of the burst reading and writings in memory access and sequentially execute this algorithm, the propagation is random (can happen or not in any direction). In theory, a hardware approach in this part of the algorithm is still better than a software design (needs to be confirmed), but the first one is extremely inflexible and takes too much time to develop.

A software approach to solving the borders is rapid to be developed, and also very flexible in a way that the solution can be performed in different ways, changing the sub-images division, its size, evaluating the border size influence and many others configuration that can provide valuable scientific contributions.

That being said, the hardware module reads a part of the image and starts to do the sequential reconstruction in each one until complete the whole image. Figure 3.2 shows a particular example solution of an image being partitioned into 16 sub-images. Figure 3.2.a shows how the image is processed using one hardware module. It first starts to do the morphological operation in a single tile, after it has finished this sub-image it begins to process the next sub-image (Figure 3.2.b). Here we can see that the order of the sub-images being processed is the sequential one, from 1 to 16.

Algorithm 8 Reconstruction using queue in the border of the sub-images (Software algorithm)

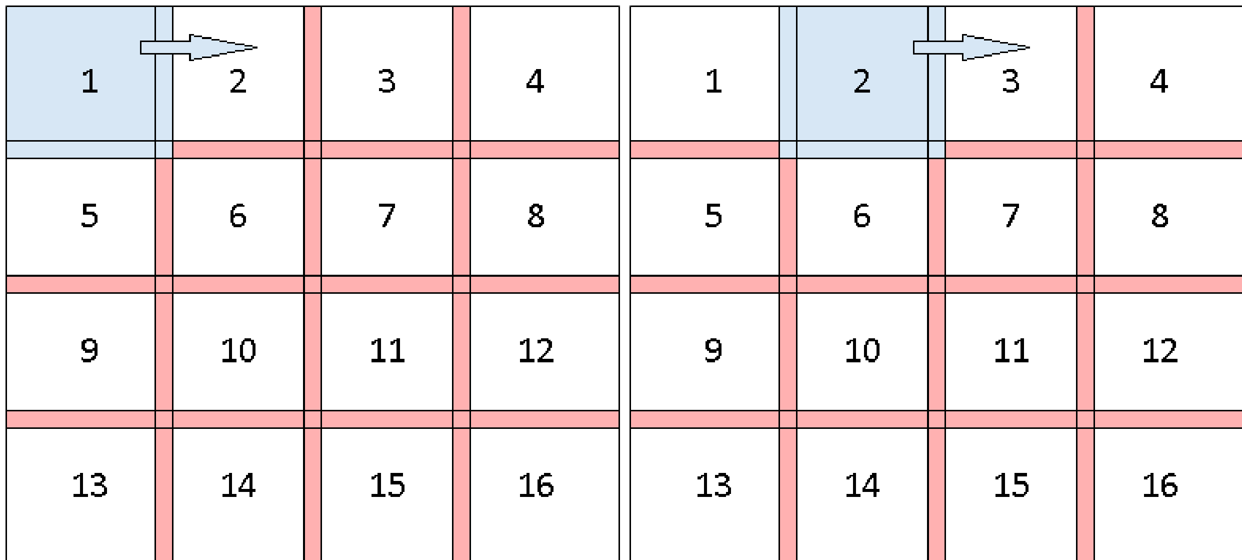
Input: mask I, grayscale image
Input: J, partially reconstructed grayscale image (missing the borders)
$\forall p \in D_I, J \leq I$
Output: J, reconstructed grayscale image

1: **Software:**
2: Partition the marker image J into M sub-images; $\{M \subset J\}$
3: **Wait** for the hardware has processed two sub-images;
4: $N \leftarrow 1$
5: **repeat**
6: **if** Sub-image processed by software has been processed by hardware **then**
7: $N \leftarrow N + 1$
8: **for all** pixel $p \in$ border between sub-image N and the previous sub-images **do**
9: $fifo_write(p)$;
10: **end for**
11: **while** $fifo_empty() = false$ **do**
12: $p \leftarrow fifo_read()$;
13: **for all** pixel $q \in N_G$ **do**
14: **if** $J(q) < J(p)$ and $I(q) \neq J(q)$ **then**
15: $J(q) \leftarrow min\{J(p), I(q)\}$; {Propagate pixel q into the region processed by hardware}
16: $fifo_write(q)$; {Insert the propagated pixel q into the end of the queue}
17: **end if**
18: **end for**
19: **end while**
20: **end if**
21: **until** all the sub-images has been processed (i.e., $N = M$)

The Figures 3.2.c and 3.2.d show the same operation as before, but in this case, it is used two hardware modules to process the image. The first module performs its operation from tiles 1 to 8, while the second hardware process at the same time the second half of the image, the tiles from 9 to 16. This solution reduces the hardware processing time by half because each module is responsible for processing only one half of the entire image.

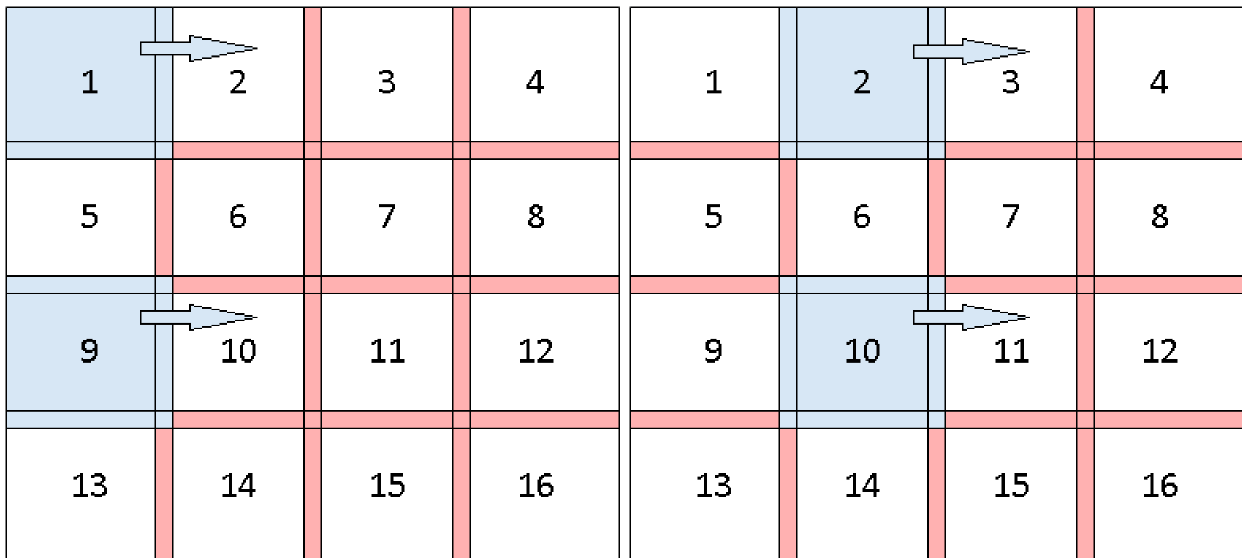
Figure 3.2 shows a particular solution when there is an overlapping between the borders of the sub-images. An overlapping happens when the next iteration of the hardware module processes pixels that have already been processed by the previous iteration (i.e., sub-image 2 has shared pixels with sub-image 1). This approach allows a pixels propagation from the previous tile to the next one in hardware, reducing the algorithm cost for the software part.

The opposite propagation, from the actual tile to the previous one, cannot be performed in this way, but a considerable amount of data are propagated using this approach, leaving only a few pixels between the borders. The best border overlapping size and its influence in time processing are posteriorly investigated in chapter 4.



(a) One instantiated hardware module processing the first sub-image.

(b) One instantiated hardware module processing the second sub-image.



(c) Two instantiated hardware modules processing an image, each one is processing its first sub-image.

(d) Two instantiated hardware modules processing an image, each one is processing its respective second sub-image.

Figure 3.2: Hardware partitioning strategy using one and two modules in its first and second iterations.

The hardware architecture proposed in this work is the same for one or two hardware modules, they use two identical configurations, just instantiating the same hardware twice and changing their address in relation to the ARM processor.

Throughout the development of this work, three solutions were designed for the software part when running with only one hardware. The first two are preliminary versions, and they are presented here to show the research evolution during this work. The last one is the main solution using only one hardware, and it can be seen in the form of a pseudo-code in Algorithm 8. All three solutions can be summarized as follow:

- **FIFO in the borders:** This solution waits until the hardware has finished all the sub-images processing. Then, it starts to make the pixels propagation between the sub-images boundaries using FIFO queue in software to do the last part of the reconstruction.

- **FIFO in the borders using two threads:** The same as above with the difference that now the software runs using two threads taking advantage of the fact that the ARM processor has 2 cores inside it. The use of two threads is only possible due to an operating system being executed in the ARM processor. In our implementation, it was used a Linux provided by Terasic (See section 2.5).

- **FIFO in the borders using two threads running simultaneously:** This solution does the same of the others, but it does not wait until the hardware has finished its processing. In this case, the software starts to run together with the hardware, taking advantage of the fact that when the hardware starts to deliver the sub-images results, the software can solve the borders between them at the same time.

It was used a different version of the solution "FIFO in the borders with two threads running simultaneously" when the software runs with two modules. This version is similar to the one presented in Algorithm 8. A better explanation is provided in chapter 4 when those different approaches will be tested and analyzed in more details.

3.2 System Architecture

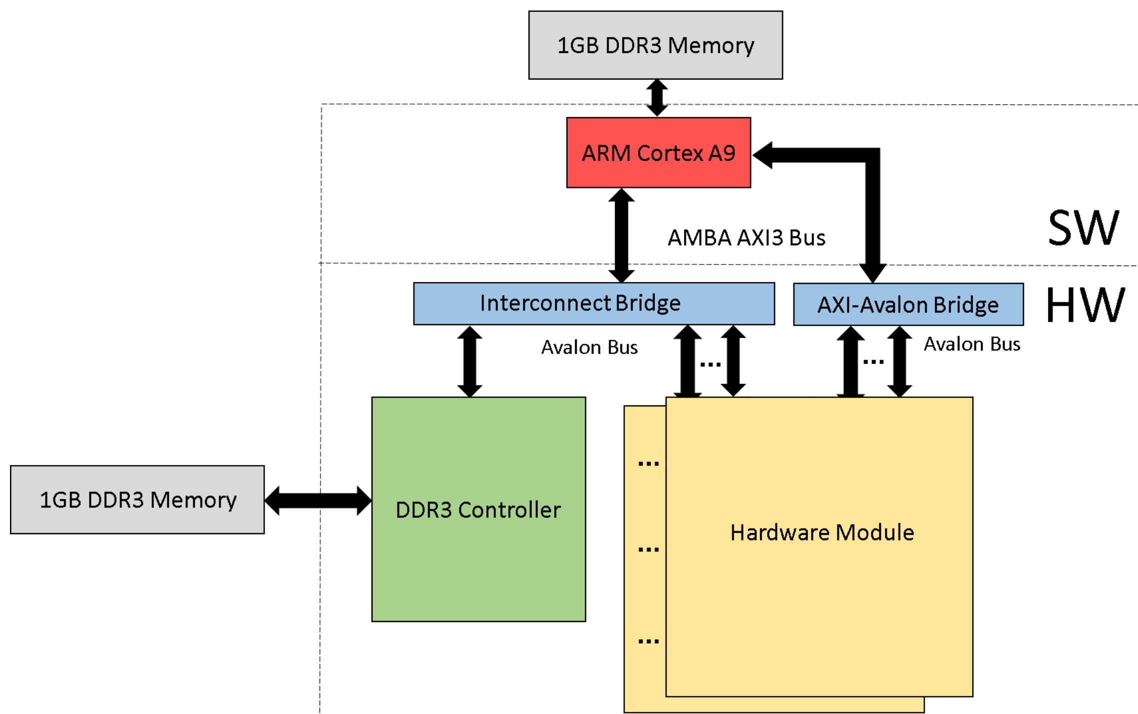


Figure 3.3: System architecture with HW/SW partitioning.

As showed in Figure 3.3, the system architecture is composed of 5 main elements as follow:

1. ARM Cortex-A9 Processor (in red);
2. Interconnect Bridges (in blue);
3. DDR3 Controller (in green);
4. External 1 GB DDR3 memories (in gray);
5. The hardware module that performs the morphological reconstruction algorithm (in yellow);

As explained in section 2.4.2, the HPS module has two bridges to communicate with FPGA, HPS-to-FPGA Bridge and Lightweight (LW) HPS-to-FPGA Bridge. The data width of the first bridge is configurable via Qsys software in the Quartus tool and on this work it was selected to the maximum allowed that is 128 bits to enhance the capacity throughput between ARM and FPGA side. This bridge was mainly used as an interface communication between the ARM processor and the 1GB DDR3 controller from the FPGA side to store the images that are going to be processed by the hardware module. The data width of the second bridge is fixed in 32 bits and was used to connect the ARM processor and the raster controller module developed in this thesis.

Qsys tool in Quartus environment automatically created the Interconnect Bridge and AXI-Avalon Bridge. Since both HPS-to-FPGA bridges use AXI protocol, when connecting them to some Avalon slave those interface bridges are automatically generated by Qsys to allow different protocols to communicate with each other. The Interconnect Bridge has two master and one slave. The masters are the HPS-to-FPGA Bridge and the raster controller, the slave is the DDR3 controller. The AXI-Avalon Bridge has only one master and one slave. It is just a bridge between AMBA AXI and Avalon protocol, the master is the LW HPS-to-FPGA Bridge, and the slave is the raster controller module.

The DDR3 SDRAM controller is an IP provided by Altera inside the Qsys. It communicates through a selectable Avalon interface, which in this work the maximum data width of 128 bits was chosen to reach the maximum possible performance. This module has also a burst option that was used to improve the speed communication between the raster controller and the DDR3 memory.

The memories are placed outside of Cyclone V SoC silicon. They are available in the SoCKit Development Board used on this work. As showed in Figure 2.10, one memory is connected directly to the HPS portion, and the other one is connected directly to the FPGA portion of the chip. The memory connected to the HPS is used as a normal RAM for the ARM, and its function during the morphological reconstruction algorithm is to store variables used on the software part of this solution. The memory connected to the FPGA side is used to store big images for reading and writing from the hardware part of this solution.

Finally, the hardware module that processes the morphological reconstruction algorithm is a block that can be replicated many times, in the way that system's design engineer demands. In

fact, on this work it was implemented a version with only one module instantiated and another version with two blocks working together. The maximum number of blocks used on this work was two because the FPGA Board chosen to develop this thesis has not enough memory to support more blocks. However, if more modules would be required, the system designer needs just to replicate the same hardware module more times, and be aware with the memory map of them.

The memory access conflicts are solved internally by the Avalon interface protocol. Since it has some waiting states, the interface guarantees that if two masters have requested a memory access at the same time, only one will have the request answered. The Avalon leaves one master at each time to have the memory access. As this interface is automatically created by Qsys software, the hardware development of this work does not need to worry about any type of internal priority interface scheduling. The hardware only needs to follow the protocol rules.

Worth to mention here is that the number of blocks used in a solution is directly correlated with the data traffic inside the interface. The proposed solution can become bottlenecked by the interface if many blocks are used. The designer should have to have a particular attention in the interface scheduling, allowing that each master could have the same time slot to access the memory. More tests should have to be performed to find the ideal number of blocks that keep this architecture as a good one to do the morphological reconstruction algorithm.

3.3 Hardware Development

The hardware part that performs the morphological reconstruction algorithm is shown in Figure 3.4. The hardware architecture can be divided into two main modules as follows:

1. **Controller:** the Main module that is the interface between the raster&anti-raster module, the ARM processor, and the external memory. It is mainly responsible for doing all the raster&anti-raster configuration, data managing and data transference to and from the 1GB DDR3 external memory. It acts as a Master for the raster module and memory controller and as a slave for ARM processor at the same time;
2. **Raster&Anti-Raster:** performs the sequential morphological reconstruction algorithm through raster and anti-raster scans, as described in Figure 2.3;

The main contribution of this work regarding hardware design is the controller module. The raster&anti-raster module has already been developed by [30], and its internal hardware characteristics are out of scope in this thesis, just the external interface and configuration are explained in section 3.3.3. It is worth to mention here is that the raster&anti-raster module performs the sequential reconstruction (SR) through raster and anti-raster scans until the image stabilization, see section 2.2. The hardware design is parametrizable, making possible to change the image size to be processed doing a compilation and synthetization with the corresponding values. The disadvantage of doing this is to fix the solution in hardware, letting very low flexibility to process images with size not equal to the hardware size.

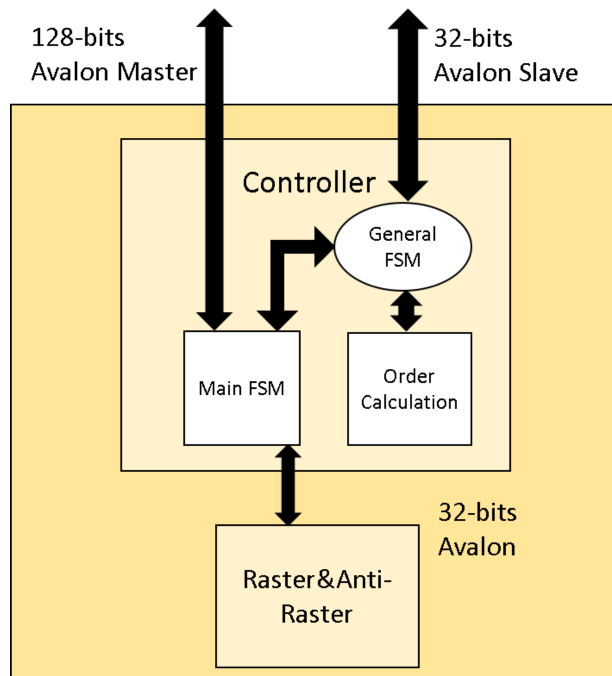


Figure 3.4: Hardware Architecture.

Therefore, the controller module adds new features to this already established solution, making possible to process images of up to $64K \times 64K$ pixels through software settings. This module contains few internal registers responsible for storing the parameters that will rule the right module functioning, they are summarized in table 3.1 and explained with more details in section 3.3.2.

3.3.1 Controller Submodule

The controller module has three Avalon interfaces, one is 128 bits width and is an Avalon Master to the Memory controller, the second one is a 32 bits width and is an Avalon Slave for ARM processor, the last one is a 32 bits width and is an Avalon Master to the raster&anti-raster module. As showed in Figure 3.4, the controller module is divided into three main sub-modules:

- **General Finite State Machine:** responsible for managing the entire module. It commands and order when the others two phases will start;
- **Main Finite State Machine:** responsible for doing all memory read and write operations, is the main state machine that deals with the raster&anti-raster communication in one hand and with the DDR3 memory controller in the other hand. This module configures the raster&anti-raster according to its requirements, transfers to it the marker and mask image, starts the raster module, reads the image result after the processing and saves it back to the DDR3 memory.
- **Order Calculation:** responsible for doing all the calculations for the next addresses to be read/write from/to memory. This is necessary since sub-images lines are not in a contiguous memory address region. Each sub-image line is spaced depending on the entire

image size. This sub-module is also supposed to calculate the order that the sub-images are going to be processed depending on the Order register value, but in this thesis, only the default order was implemented (Left to right, top to down);

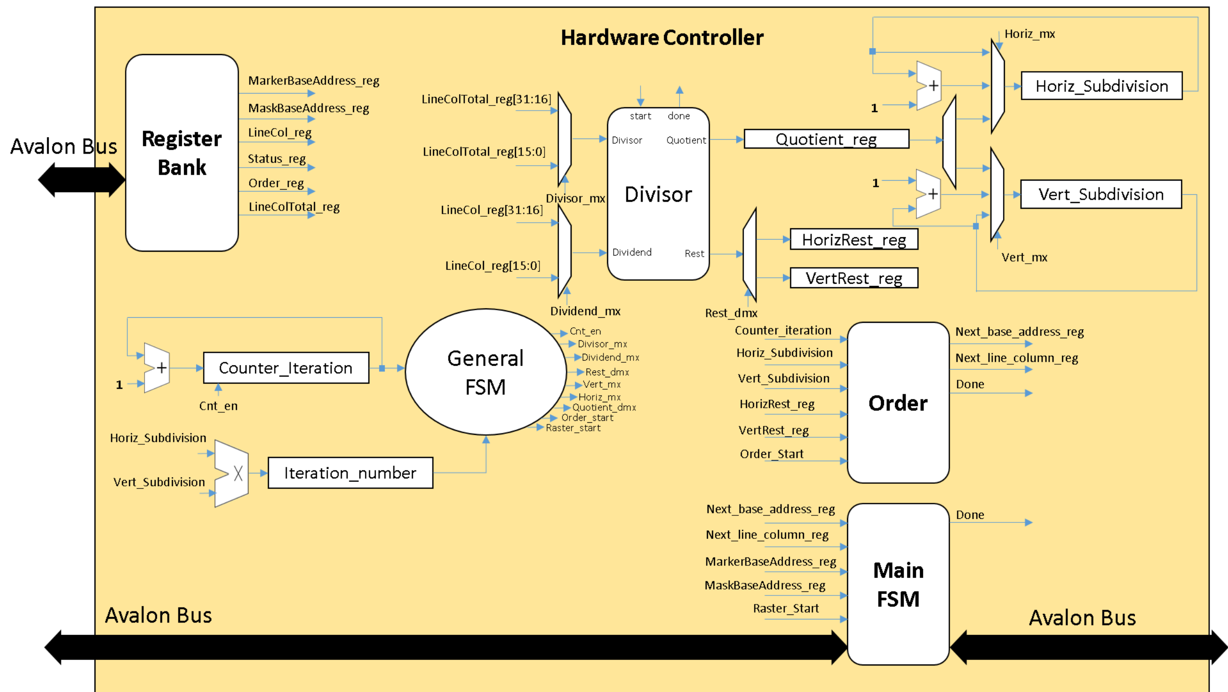


Figure 3.5: Hardware Controller Architecture.

Those sub-modules are three different state machines connected with each other. A complete diagram of General Finite State machine can be seen in Figure 3.6, and it shows that the General FSM rules the others two sub-modules. A better visualization of the hardware controller internal architecture in more details is shown in Figure 3.5. The correct functionality of the internal registers inside the Register Bank block is explained in section 3.3.2. Both Figures 3.6 and 3.5 together shows how the controller module works.

Firstly, it waits for the STATUS register to be written 1 to begin the processing. This can be done via ARM processor. Then, it starts to process the parameters that will rule the entire hardware, the number of sub-images partitions. This can be done through a division between the values in the registers LineColTotal_reg and LineCol_reg. The final value of the Horiz_Subdivision and Vert_Subdvision depends on the rest of the previous division. This guarantees that an image could be divided into a non-multiple sub-image size, this is better explained in the section 3.3.4.

After this initial calculation, the controller block activates the Order Calculation module that performs a control logic to obtain the right address values according to the sub-image that is going to be processed and waits until the Order Calculation module has finished through the signal flag ”ORDER_DONE”. After that, it goes to the next state that will send a signal called ”CONTROLLER_START” to the Main FSM to start its operations. Finally, the Main FSM does all the steps that it is programmed to do and then, the General FSM decides if it is going to repeat the operation for the next sub-image or if it is going to finish the processing. This decision is taken

based on the Counter_Iteration value. After each sub-image that the hardware has processed, this counter is incremented by 1. In the end, if the counter is equal to the number of iterations expected (Horizontal Subdivisions x Vertical Subdivisions), it stops the hardware processing, and the module comes back to the IDLE state.

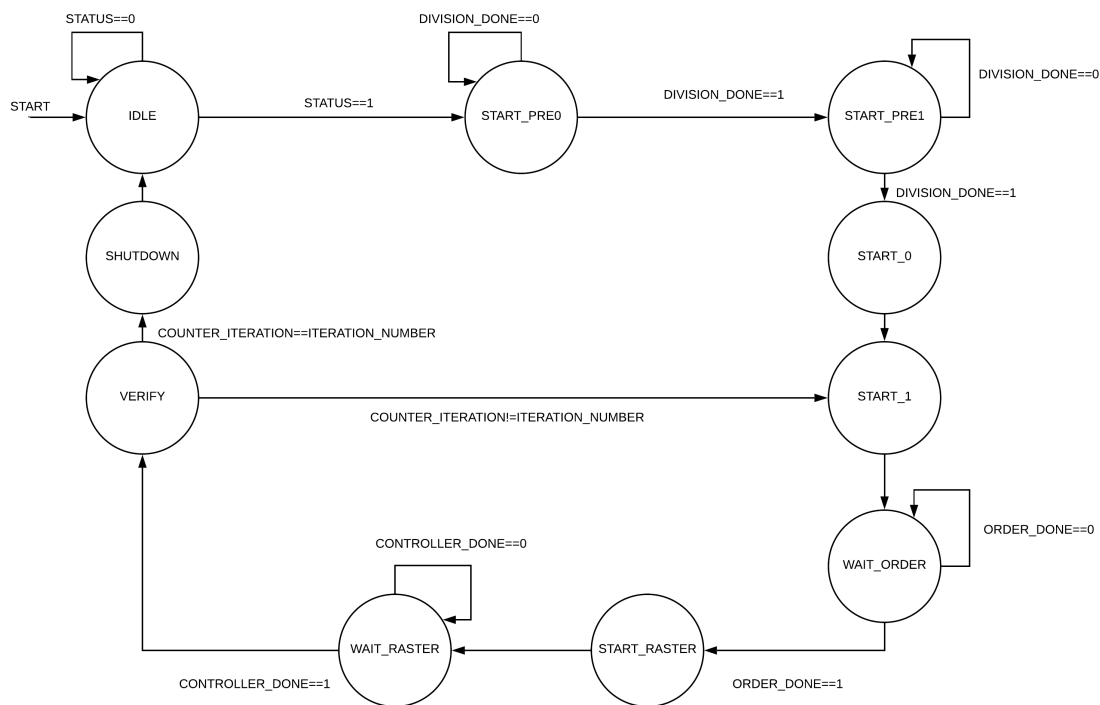


Figure 3.6: General Finite State Machine.

Figure 3.7 shows a complete diagram for the Main Finite State Machine module. It is responsible for doing all the read and write operations from and to the memory device. It is also responsible for raster module configuration, and for sending a signal to the General FSM called ”CONTROLLER_DONE” that synchronizes both states machines. The Main FSM only starts when the General FSM sends the ”CONTROLLER_START” signal. When this happens, the Main FSM starts to perform all the sequential operations automatically.

The Main FSM Block has five counter, each one is responsible for the synchronization of some particular states, that needs to run in a specific number of clocks. The Main FSM is basically constituted of 8 sequential steps: Read marker image from memory; write marker image to raster module; read mask image from memory; write mask image to raster module; play the raster module; wait the raster module has finished (raster and anti-raster scans until stability); read result from raster module, write result back to memory. The complete diagram of the Main Finite State Machine is shown in Figure 3.7, including the states that perform a zero padding and the intermediate steps that configure the raster&anti-raster module. The MAX_COL and MAX_ROW are the sizes of the raster&anti-raster module. They are hardwired and fixed in the moment of the hardware synthetization in FPGA.

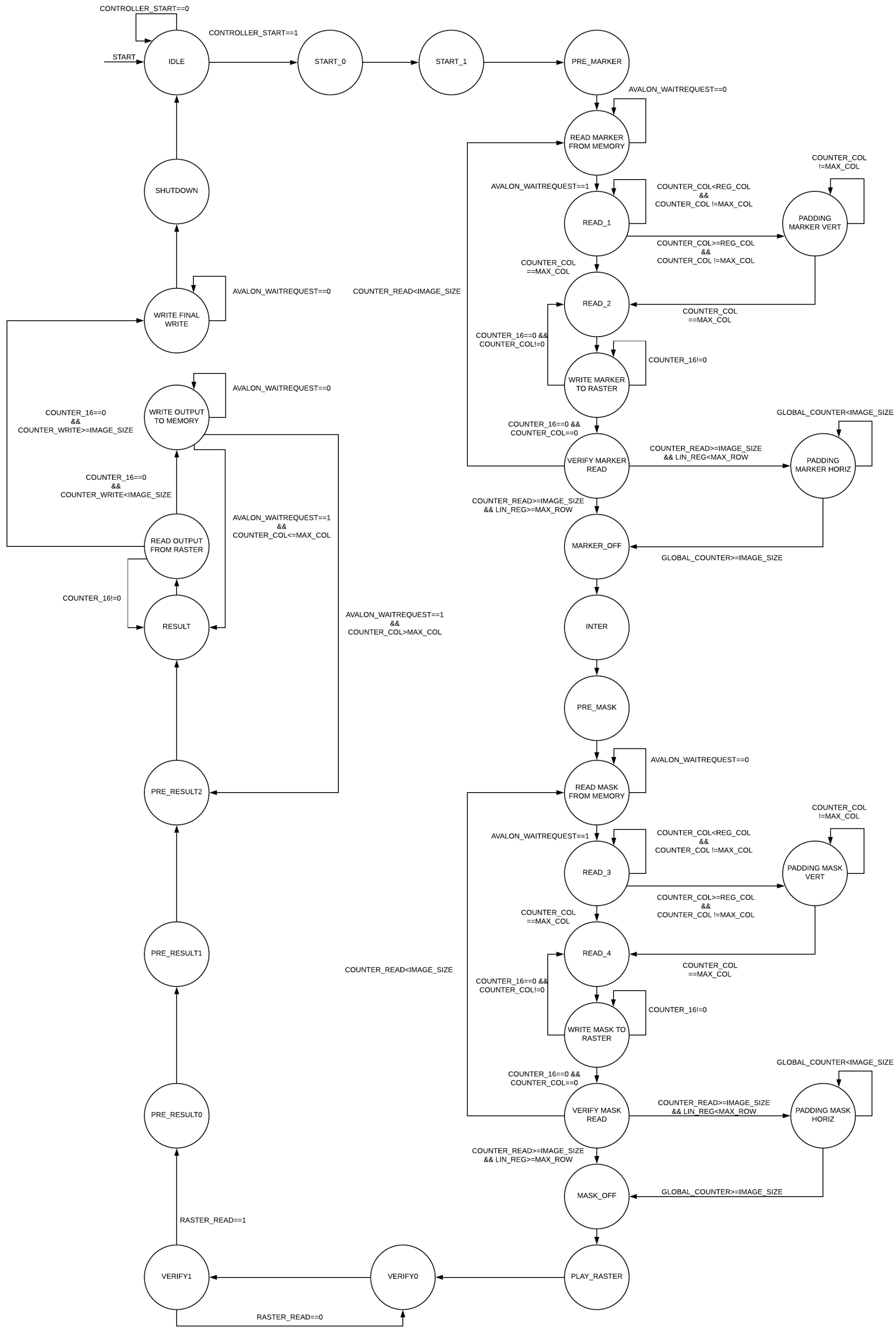


Figure 3.7: Main Finite State Machine.

3.3.2 Controller Programmable Internal Registers

The hardware controller module has six main registers responsible for storing meaningful information regarding the functionality and programmability of the hardware controller module.

Table 3.1 shows the registers and a brief description of their functionalities.

Table 3.1: Register map for controller hardware.

Offset	Register Name	R/W	Bits		Description
			[31:16]	[15:0]	
0x00	Marker Address	R/W	32 Bits		Contains the initial marker image memory address
0x01	Mask Address	R/W	32 Bits		Contains the initial mask image memory address
0x02	Line-Column	R/W	16 Bits	16 Bits	- Higher bits: Contains the sub-images vertical size (Line number). - Lower bits: Contains the sub-images horizontal size (Column number)
0x03	Status	R/W	32 Bits		Contains information regarding the hardware. - 0: Hardware inactive; - 1: Writing 1 makes the hardware start; - Others: Tells the sub-image number that hardware is processing at the moment.
0x04	Order	R/W	32 Bits		Register reserved to allow different sub-image processing order. - 0: Default processing order. Left to right, Top to down processing; - Others: Not implemented yet. Reserved for future improvements.
0x05	Line-Column-Total	R/W	16 Bits	16 Bits	- Higher bits: Contains the total image vertical size (Line number). - Lower bits: Contains the total image horizontal size (Column number)

For the correct morphological reconstruction operation, it is necessary to set the right registers according to the image parameters to be processed. The hardware design development took into consideration offering to software designer the possibility of choosing the parameters in a way that would be more convenient for them. The following steps are needed in order to make the hardware image morphological operation starts:

1. Write the initial address where the Marker image is stored;
2. Write the initial address where the Mask image is stored;
3. Write the total size of the image to be processed in the register Line-Column-Total;

4. Write the sub-image size that the entire image will be divided into the register Line-Column. The value written here must be less than the hardware size of the raster&anti-raster module (selected in hardware synthesis);
5. Write the order that the image will be processed. For now, it has only one option and this step can be skipped;
6. Write 1 in the register Status. This will start the entire module.

Table 3.1 shows that all the registers have read and write permission. But, after the hardware controller starts, all the registers must be held with the same value to avoid mistakes. During the hardware controller operation, the Status register shows the sub-image number that the raster&anti-raster module is processing at the moment. When the Status register becomes 0 again means that the hardware controller module has finished all sequential reconstruction in all sub-images of the entire image.

3.3.3 Raster&Anti-Raster Module Interface

As mentioned early, the raster&anti-raster module functioning details is out of this thesis scope. This section explains the internal configuration and proper interface communication to others modules. In our particular case, the interface communication with the controller module.

The raster&anti-raster block has an Avalon Interface of 32 bits wide. This module has been modified many times during the development of this research, nowadays it has a possibility of setting some parameters that were not exploited in this work as the number of raster and anti-raster scans to be performed by the module and store pixels with good propagation chance in a queue to future processing. The Tables 3.2 and 3.3 shows a summary of the writing and reading interface functionality. All addresses related to FIFO properties were not used in this work.

The steps performed by the Main Finite State Machine, Figure 3.7, can be done by writing into the correct addresses and reading from them. For instance, to write the marker Image into the raster&anti-raster module, the controller module should make a write operation using the address region from 0x00000000 to 0x00FFFFFF in the address line of Avalon interface. By doing this, the image will be written into the internal memory of the raster&anti-raster module. The same procedure can be applied for a reading operation and all others operations related to the communication between controller and raster&anti-raster modules.

3.3.4 Hardware Operation

Figure 3.8 illustrates an example of a hardware configuration in normal operation, the registers were written with the values shown in Table 3.4. In this case, the raster&anti-raster module needs to be configured at least in 288×288 size to support sub-images of 256×256 with overlapping of 16 in each direction ($256 + 16 + 16 = 288$), as shown in Figure 3.8.

Table 3.2: Memory map for raster&anti-raster writing interface.

Start Address	**End Address**	**R/W**	**Description**
0x00000000	0x00FFFFFF	W	Writes Marker Image
0x01000000	0x01FFFFFF	W	Writes Mask Image
0x02000000	0x02FFFFFF	W	Writes the number of scans to be done
0x03000000	0x03FFFFFF	W	Writes start command
0x04000000	0x04FFFFFF	W	Writes parameters for reading the Marker Image
0x05000000	0x05FFFFFF	W	Writes parameters for reading the Mask Image
0x06000000	0x06FFFFFF	W	Writes parameters for reading the FIFO
0x07000000	0xFFFFFFFF	W	Commands pause

Table 3.3: Memory map for raster&anti-raster reading interface.

Start Address	**End Address**	**R/W**	**Description**
0x00000000	0x00FFFFFF	R	Reads Marker Image
0x01000000	0x01FFFFFF	R	Reads Mask Image
0x02000000	0x02FFFFFF	R	Reads the stop FIFO flag
0x03000000	0x03FFFFFF	R	Reads the stop general flag
0x04000000	0x04FFFFFF	R	Reads FIFO values
0x05000000	0x05FFFFFF	R	Reads FIFO counter
0x06000000	0xFFFFFFFF	R	Reads the number of scans that has been done

(a) First iteration.

(b) Sixth iteration.

Figure 3.8: Hardware configuration example running in a 1024x1024 pixels image, sub-blocks of 256x256 pixels, and overlapping border of 16.

Also, it is good to note that the initial Mask address must be a value that would not be in the same region of the Marker image, otherwise we could have an overlapping between both images. In our example, once the initial Marker address is 0 and the total Marker image size is $1024 \times 1024 = 1.048.576$, is easy to conclude that the minimum initial Mask Address should be

Table 3.4: Register configuration example.

Register Name	Value	
	[31:16]	[15:0]
Marker Address	0	
Mask Address	1.048.576	
Line-Column	256	256
Order	0	
Line-Column-Total	1024	1024

exactly the size of the Marker image($1.048.576$).

The area of the image that the raster&anti-raster is processing during the first sub-image is represented in blue in Figure 3.8, the 256 line pixels and 256 column pixels plus an overlapping of 16 pixels of the next image. After the module performs the sequential reconstruction algorithm until the image stabilizes, the hardware module starts to process the next sub-image in the normal order chosen, the default one (left to right, top to down). In our example, 1024×1024 divided by 256×256 images is equal to 16 iterations.

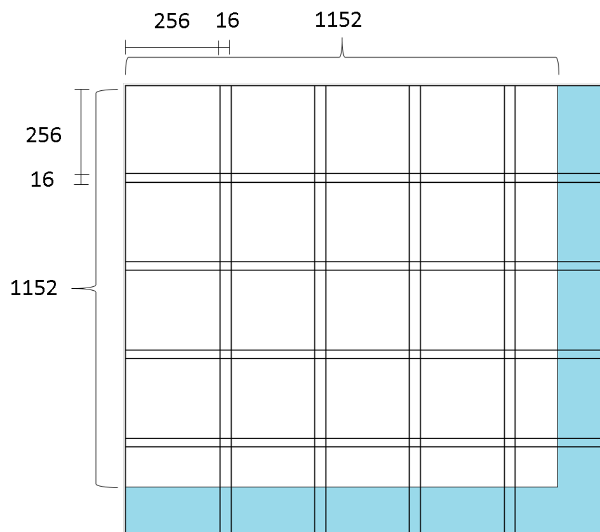


Figure 3.9: Hardware configuration example running in a 1152x1152 pixels image, sub-blocks of 256x256 pixels, and overlapping border of 16.

Another kind of configuration can be seen in Figure 3.9. As mentioned before, the hardware controller is flexible and offers the possibility of processing very different image sizes, including image size that does not match with the sub-image size chosen. Figure 3.9 shows a situation when the image size (1152×1152) is not multiple of 256×256.

In this case, the hardware controller detects it and decides to fill the rest of the image with zeros (padding), in blue, until the entire image matches the 256×256 division. The sub-module raster&anti-raster will perform the SR algorithm as normal, considering the pixels in zero as part of the image. This feature improves the solution flexibility to almost any situation regardless of

the image size chosen but is not performance optimized due to the fact that the raster&anti-raster module will process an image of 1280×1280 instead of 1152×1152. Like many situations where a performance x flexibility trade-off exists, this one is not different, and as covered in section 1.1, we prioritized flexibility over performance in this particular case.

3.3.5 Hardware Features and Specification

All the hardware specifications can be summarized as follows:

1. Image size must be multiple of 16;
2. Image maximum size of 64K x 64K;
3. Sub-image size must be multiple of 16;
4. Sub-image maximum size depends on the raster&anti-raster module size;

Features 1 and 3 are a restriction of the Avalon interface. Since the DDR3 controller is addressable in bytes, having 128 bits on its Avalon interface means that each address is written as a multiple of 16. As stated in [41], Avalon-MM interface does not support unaligned access to Avalon slaves. This situation could be solved internally in hardware, reading the aligned address and solving any unaligned pixel later, but it would require a longer development time that the author preferred to keep this as a feature solution instead of solving this. At the same time, this feature will not significantly impact the overall flexibility characteristic of the proposed solution.

Feature 2 is a direct result of the entire capacity of Total-Line-Column register. As it is 16 bits for Line configuration and 16 bits for Column, the maximum image size that this register can be configured is equal to $2^{16} \times 2^{16} = 64K \times 64K$.

The last feature is obtained from the raster&anti-raster maximum capacity. The sub-image size can be any value inferior to the raster&anti-raster size plus the overlapping between the sub-images. An example, if the raster&anti-raster size is 288×288 with overlapping of 16, the maximum sub-image size that can be set is 256×256 because $288 - 16 - 16 = 256$. If the software user decides to process an image smaller than raster&anti-raster size, the controller module will fill the rest with zeros until the image size have been equated to the raster&anti-raster size.

Any of those specifications that are not followed by the user of the hardware architecture will lead to a mistake in the final image result processing (i.e., process an image with size not multiple of 16 will generate a wrong image result).

3.4 Software Development

Before we approach the steps towards the software development that will be run in the ARM processor, we need to talk about the memory map of HPS to know better which addresses will be

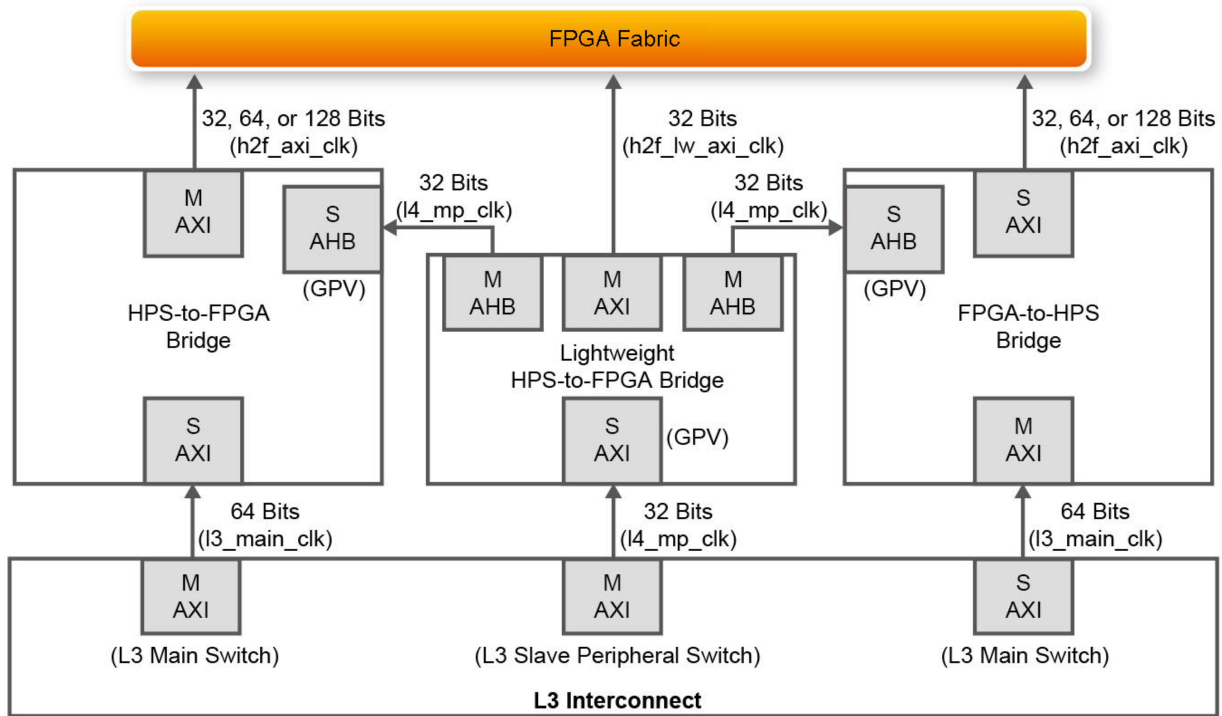


Figure 3.10: HPS to FPGA Bridge Block diagram, adapted from [7].

used to write and read information from and to the hardware architecture. Table 2.1 was copied to this section at Table 3.5 to better visualization, and it shows a summary of the main memory address that will be used in the software design. The complete memory map can be seen in Table I.1 in Appendix I. Figure 3.10 shows the AXI bridge diagram with the interfaces between HPS and FPGA, the ”FPGA-to-HPS Bridge” was not used in this thesis, as the hardware module act only as a slave in relation to the ARM Processor.

Table 3.5: Memory Map reduced for HPS-to-FPGA bridge of MPU Subsystem.

Interface	**Name**	**Start Address**	**End Address**	**Size**
HPS2FPGASLAVES	FPGA Slaves Accessed Via HPS-to-FPGA AXI Bridge	0xC0000000	0xFBFFFFFF	960 MB
LWFPGASLAVES	FPGA Slaves Accessed with Lightweight HPS-to-FPGA AXI Bridge	0xFF200000	0xFF3FFFFF	2 MB
MPU	MPU registers	0xFFFEC000	0xFFFEEFFF	8 KB

In this project, the HPS will configure the hardware module through ”Lightweight HPS-to-FPGA AXI Bridge”, the addresses that will be used in the C program are equivalent to the LWF-PGASLAVES interface (0xFF200000 to 0xFF3FFFFF). The HPS processor needs to write/read images from/to DDR3 external memory through ”HPS-to-FPGA AXI Bridge”, the addresses be-

tween 0xC0000000 and 0xFBFFFFFF.

The last addresses from the HPS memory map that were used in this thesis contain some internal timers and are responsible for knowing the exact algorithm processing time. They are located in the region between 0xFFFEC000 and 0xFFFEEFFF inside the MPU registers, the exact functionality of these timers are better explained in section 3.4.2.

The base addresses for DDR3 controller and the hardware modules in relation to the ARM processor are both 0x00000000. Therefore, developing a software program that writes at address 0xC0000000 means to write in the DDR3 memory at address 0x00. Likewise, when writing at 0xFF200000 means to write at the internal register 0x00 in the hardware module, which is equal to the Marker Address register according to the Table 3.1.

3.4.1 Reading and Writing Procedures

Before we start the basic commands for reading and writing, we need to define the base address that is going to be used as follow:

```
#define HW_REGS_BASE ( 0xC0000000 )
#define HW_REGS_SPAN ( 0x3FFFFFFF )
```

After this, the first thing inside the main function is to use the ”dev/mem” and ”mmap” system-calls to map the physical base address of the hardware into a virtual address in which can be accessed through C software. In the following piece of code, the ”dev/mem” is used to open the memory device driver, the virtual_base address is where the physical address is mapped into a virtual address, in this case is in relation to the address 0xC0000000 called HW_REGS_BASE. Finally, The raster_base_address is the virtual address plus the offset for the raster module (located at 0xFF200000).

```
void *virtual_base, *raster_base_address;
int fd= open("/dev/mem",(O_RDWR|O_SYNC));
virtual_base = mmap(NULL, HW_REGS_SPAN, (PROT_READ | PROT_WRITE), MAP_SHARED, fd,
    HW_REGS_BASE);
raster_base_address = virtual_base + 0xFF200000 - HW_REGS_BASE;
```

Now that everything was set, the software can communicate with the hardware writing parameters and reading the image pixels from FPGA fabric and also monitoring the hardware flags to synchronize the actions between software and hardware. This can be done through direct pointers attributions pointed to those memory address or simply using ”memcpy” function as showed in this piece of code extracted from the original software developed in this thesis:

```
//Copying the images from buffers to DDR3 memory using the virtual_base address
for (i = 1; i < rows + 1; i++)
{
```

```
  memcpy((virtual_base+(i-1)*cols), marker[i]+1, cols);
  memcpy((virtual_base+(cols*rows)+(i-1)*cols), mask[i]+1, cols);
}

//Configuring HARDWARE through raster_base_address
*(uint32_t *)(raster_base_address)    = 0x00000000;       // Marker address
*(uint32_t *)(raster_base_address+ 4)=(cols*rows);        // Mask address
*(uint32_t *)(raster_base_address+ 8)=(256 <<16 | 256);   // Line-Column
*(uint32_t *)(raster_base_address+ 16)=0x00000000;        // Order
*(uint32_t *)(raster_base_address+ 20)=(rows<<16 | cols); // Line-Column-Total

//Starting the module
*(uint32_t *)(raster_base_address+ 12)=0x01;              // START command
```

All the software codes developed in this thesis are available in github: `https://github.com/f1rgc/felipe-master-thesis`.

3.4.2 Time Measurement using Private Internal Timers

The MPU internal Timers are located inside the MPU registers space, and as shown in Table 3.5, remains in the addresses between 0xFFFEC000 and 0xFFFEEFFF. The timer that was used in this thesis for measuring the algorithm processing time is a Private Timer, and can be located in the space address between 0xFFFEC600 and 0xFFFEC6FF, for a complete memory map of the MPU registers refers to the Table I.2 in Appendix I.

Table 3.6: Private Timer registers, adapted from [8].

Offset	Type	Reset Value	Description
0x00	R/W	0x00000000	Private Timer Load Register.
0x04	R/W	0x00000000	Private Timer Counter Register.
0x08	R/W	0x00000000	Private Timer Control Register.
0x0C	R/W	0x00000000	Private Timer Interrupt Status.

The Private Timer has 4 registers responsible for store information regarding the normal functioning of the timer. Their offset address can be seen in Table 3.6. The Private Timer has the following features [8]:

- a 32-bit counter that generates an interrupt when it reaches zero
- an eight-bit prescaler value to qualify the clock period
- configurable single-shot or auto-reload modes
- configurable starting values for the counter
- the clock for these blocks is PERIPHCLK

Is important to note that the clock used for these timers is the PERIPHCLK running at $1/4$ the rate of microprocessor unit base clock [4]. Since the MPU base clock of the SoCKit Development Kit runs at 925Mhz [42], we can state that PERIPHCLK runs at 231.25Mhz.

Before entering in the timer details, it is worth to state that the period between two events generated by a private timer is calculated using the following equation in seconds [8]:

$$\frac{(\text{PRESCALER_value} + 1) \times (\text{Load_value} + 1)}{\text{PERIPHCLK}} \tag{3.1}$$

An event happens when the Counter Register reaches the value 0. The $PRESCALE_value$ can be configured in the Timer Control register, the $LOAD_value$ can be configured in the Timer Load Register, and the $PERIPHCLK$ is already known and is equal to 231,250,000 (231,25Mhz).

Figure 3.11 and Table 3.7 show the Private Timer Control Registers and its descriptions. These information are useful for setting the right parameters in order to measure the timing according to the requirements.

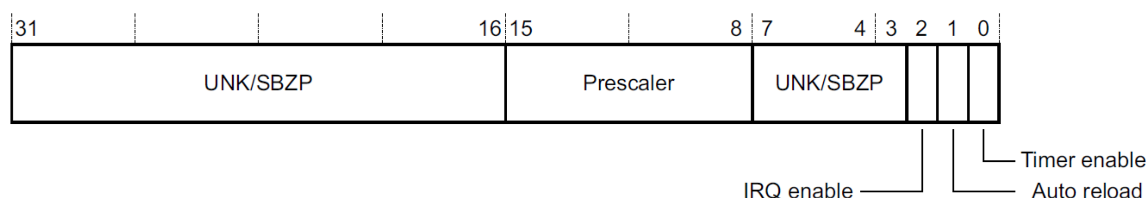


Figure 3.11: Private Timer Control Registers bit assignment, adapted from [8].

Table 3.7: Private Timer Control Registers, adapted from [8].

Bits	Name	Description
[31:16]	-	Unknown/Should-Be-Zero-or-Preserved
[15:8]	Prescaler	The prescaler modifies the clock period for the decrementing event for the Counter Register. See the equation 3.1.
[7:3]	-	Unknown/Should-Be-Zero-or-Preserved
[2]	IRQ enable	If set, the interrupt ID 29 is set as pending in the Interrupt Distributor when the event flag is set in the Timer Status Register.
[1]	Auto reload	**0:** Single shot mode.Counter decrements down to zero, sets the event flag and stops. **1:** Auto-reload mode.Each time the Counter Register reaches zero, it is reloaded with the value contained in the Timer Load Register.
[0]	Timer enable	**0:** Timer is disabled and the counter does not decrement. All registers can still be read and written. **1:** Timer is enabled and the counter decrements normally.

The Timer Counter Register is a decrementing counter. It decrements if the timer is enabled using the timer enable bit in the Timer Control Register. When the Timer Counter Register reaches zero, and the auto-reload mode is enabled, it reloads with the value in the Timer Load Register and then decrements from that value. If the auto-reload mode is not enabled, the Timer

Counter Register decrements down to zero and stops. The timer is decremented every prescaler value+1. For example, if the prescaler has a value of five then the private timer is decremented every six clock cycles, the PERIPHCLK is the reference clock.

Chapter 4

System Verification, Testing and Analysis

This chapter presents the verification methodology used in this thesis, all the tests that have been done, the analysis followed those tests and all discussions and comparison to measure the algorithm performance of the proposed solution.

4.1 Introduction

One of the best advantages of using an FPGA to prototype a solution is the capability to verify the whole system much faster than using standard verification procedures used in digital integrated circuits (IC) design. In IC design, exhaustive verification methods are used to find all the possible bugs that can lead to a mistake. The main intention is to verify the circuit functionality, exercising the system with random packets and aleatory inputs to see if some bad behavior happens, or if the output is not the one that was expected. Within several verification methodologies, we can highlight coverage-based verification (code coverage and functional coverage), assertion-based verification, detailing Statement, Toggle, Branch and Conditional coverage [43].

FPGA prototyping can be used as a proof of concept platform, the entire device with all functionality can be represented, specific blocks only or any combination in between. Also, for obvious reasons, the software integration can be shortened and become an easy task. The FPGA platform can provide the software designer with a bit exact, clock cycle accurate development system. Once a certain amount of software is written it becomes possible to evaluate the whole system architecture and its functionality. [43].

In the morphological reconstruction algorithm implementation proposed in this thesis, the functional verification is performed by comparing the output image given by the system with the image result provided by an established solution, i.e., a golden model. To do a functional validation, the golden model chosen was the function "*imreconstruct*" from Matlab® software, that does exactly the image morphological reconstruction. All the results given by the software solutions and hardware/software co-design were thoroughly compared with the Matlab results to guarantee the correctness of each step during the development of this work.

4.2 Verification Methodology

The most challenging part during verification is the hardware design. We need to make sure that it will work properly to reduce the time development of the project. Since the software part is usually faster to implement and verify than hardware, the hardware part is the bottleneck of the project regarding development time,. For the hardware part, the verification methodology was divided into two steps:

- **HDL Testbench:** A simple testbench was created in VHDL language to stimulate the hardware with basic inputs. The intention is to have a quick verification step that can validate the basic hardware functionality without the necessity of synthesizing in FPGA. With this testbench, basic errors can be found in early stages of the project and can be fixed much faster.
- **FPGA prototyping:** This part of verification is done at the FPGA system and can be divided into two as well:
  - **Small images processing:** Images whose sizes were smaller than hardware capability were processed and compared with the Matlab results. In this case, small images were used because of internal memory space limitation in FPGA. It is not the intention here to process bigger images because we need to isolate the verification focusing only on the logical functionality of one single hardware. Otherwise, we could miss where is the error in the system.
  - **Big images processing:** In this phase is required to process bigger images. The output of this step is not the morphological reconstruction algorithm yet, the border between the sub-images processed by hardware needs to be solved. So to validate this part, it was required to process each sub-image individually in Matlab, and their outputs should be merged in a bigger image that can be compared with the hardware output. The intention here is to validate the hardware processing when images bigger than its maximum size are processed, it is essential to monitor the steps between each sub-image, looking for bugs as overlapping images, wrong access address and so on.

For software part, the verification is much easier and faster. It is just a matter of comparing the output image of the software with Matlab result. In this case, the input images are the same that were generated by Matlab when individually running each sub-image merged in a bigger image. This step needs to be done because we need to make sure that the final part of the algorithm implemented in software, the pixels propagation using FIFO queue in sub-images borders, is working correctly.

Once the hardware module and software part alone are fully verified, the next step is to test the solution as a whole, in which both are working together. This eliminates the following errors to possible mistakes between hardware and software communication, instead of anyone individually.

4.3 Tests

The input images used in experimental evaluation have been collected in brain tumor research studies made by In Silico Brain Tumor Research Center (ISBTRC) [44]. The input images vary both in terms of size (from $4K \times 4K$ to $8K \times 8K$ pixels) and tissue coverage, which refers to the approximate percentage of the image area that is covered with tissue: 100% tissue coverage relates to images ”marker-100”, ”mask-100” and ”Out-100”. The same procedure can be applied for the other $4K \times 4K$ images shown in Figures 4.1, 4.2 and 4.3. The 8K images can be seen in Figure 4.4.

Each different configuration was executed five times for each one of the images. The hardware execution time remained constant for all the tests performed with the same image in the case that only one hardware block was instantiated. In the case that two modules were used, it was noticed a very tiny variation in time processing.

This can be related to the memory conflicts caused when two or more modules are trying to access the same memory at the same time, causing non-deterministic timing variations. The hardware processing time using two modules has changed randomly but in a very little value, showing that the memory conflicts using two modules have a very low impact in the final execution time. Tests connecting more than two hardware modules in the final solution should have to be performed in order to establish an accurate relationship between performance impact and memory access conflicts.

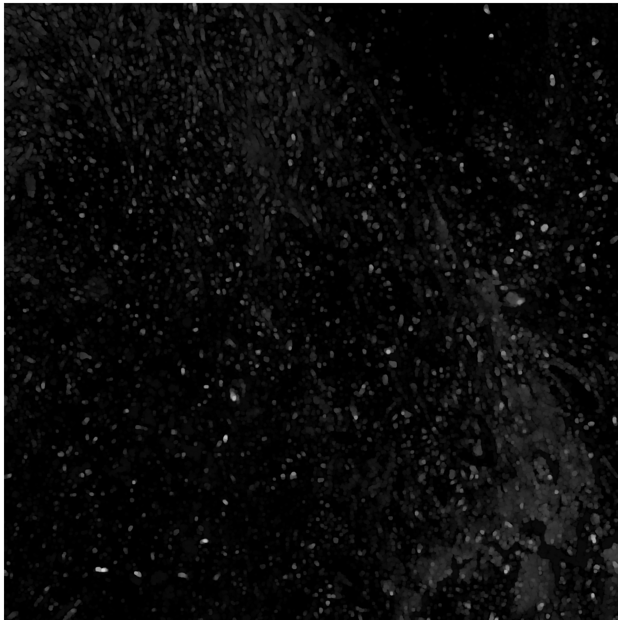
The software execution time showed some variations when executed more than once for the same image. The final time measurement took into consideration the ordinary arithmetic mean between the tests performed.

The hardware architecture used during all the tests was the same. The only changes that have been done were the internal parameters that rule the size of the hardware block and the overlapping border between sub-images. The solution with two hardware modules used two identical configurations, just instantiating the same hardware twice and changing their address in relation to the ARM processor.

4.4 Analysis

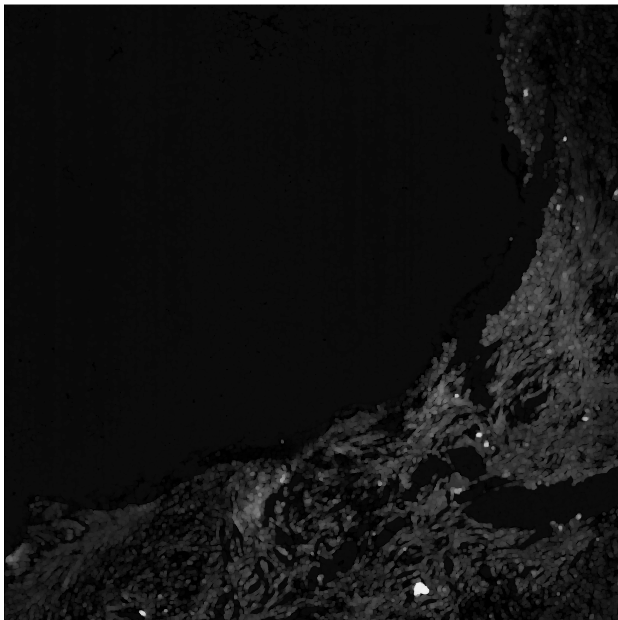
Before entering into details of the tests that have been made, it is necessary to evaluate some morphological reconstruction algorithms implemented on different platforms to use as a baseline reference. This brings a meaningful reference point to which we can compare with our solution.

Firstly, a hardware theoretical reference was created using the pure sequential reconstruction (SR) algorithm, the same that was used in this thesis for smaller images. The intention is to estimate the cost of the SR algorithm fully implemented in hardware, ignoring memory access delays, and in a situation that we have enough internal memory to process images bigger than 288×288 pixels. The best possible hardware solution, in this case, processes 1 pixel per clock.



(a) 4K marker-100.

(b) 4k marker-75.

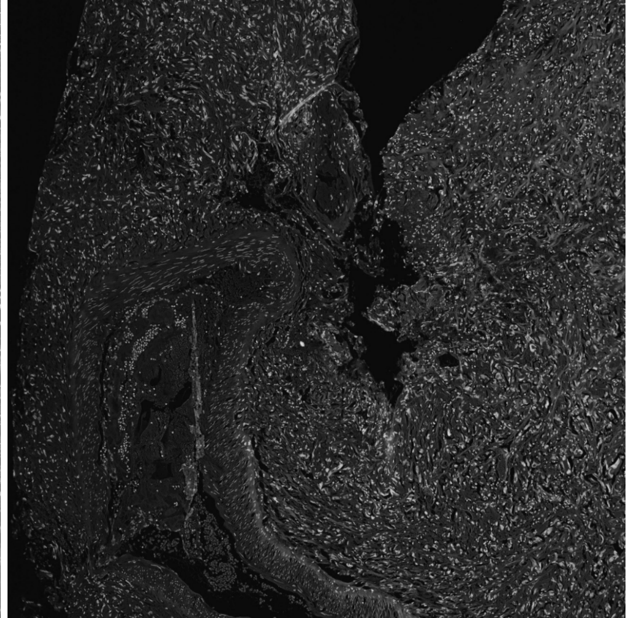


(c) 4K marker-50.

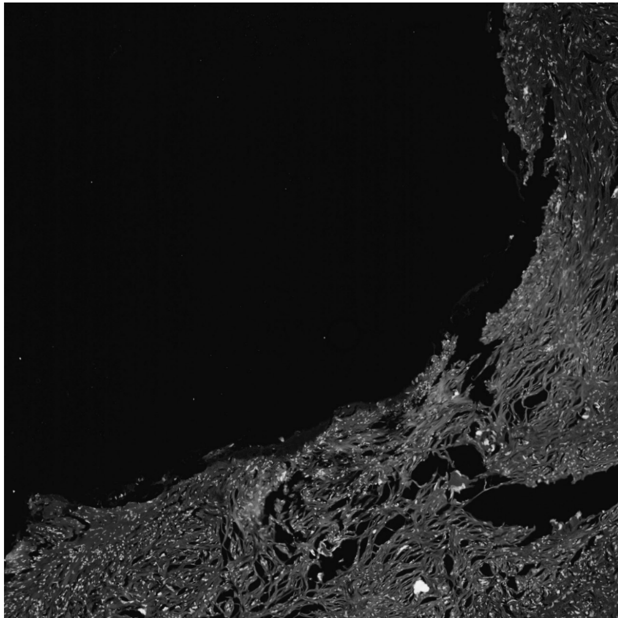
(d) 4k marker-25.

Figure 4.1: 4K marker images used for testing.

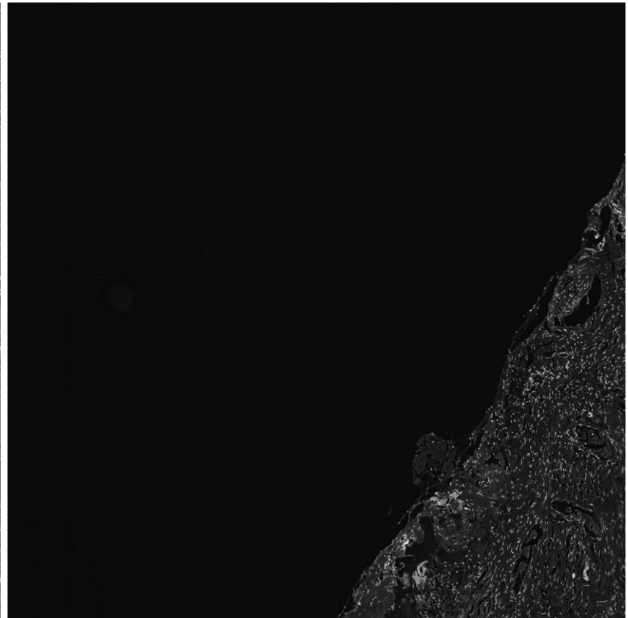
(a) 4K mask-100.



(b) 4k mask-75.

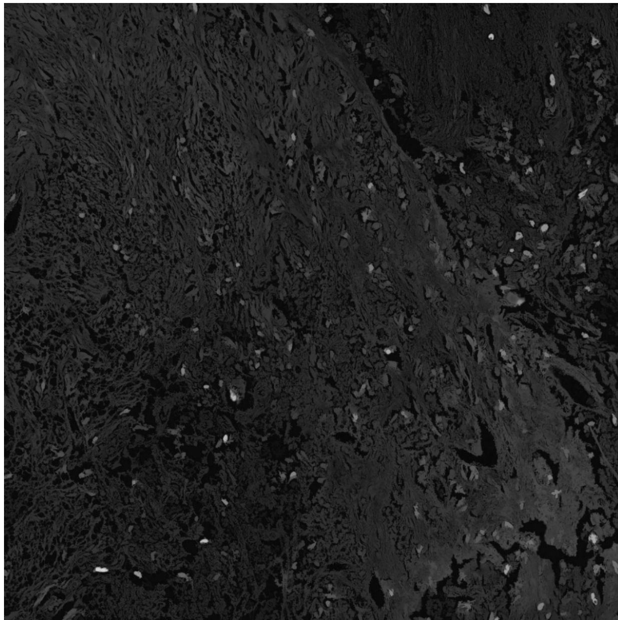


(c) 4K mask-50.

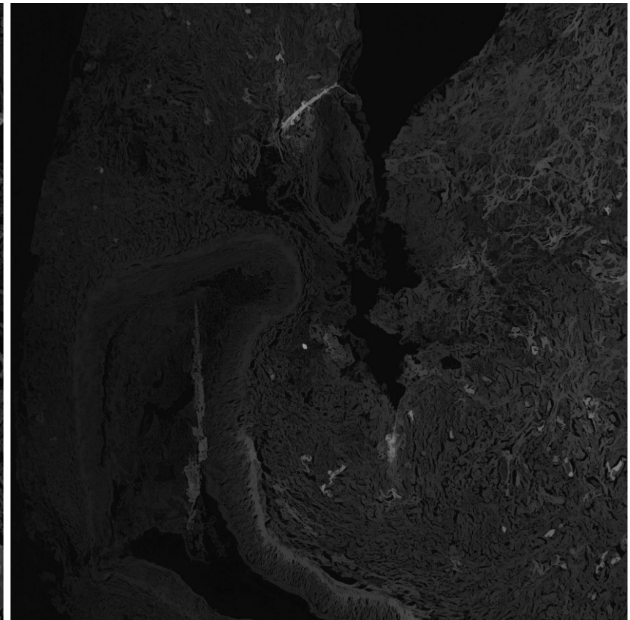


(d) 4k mask-25.

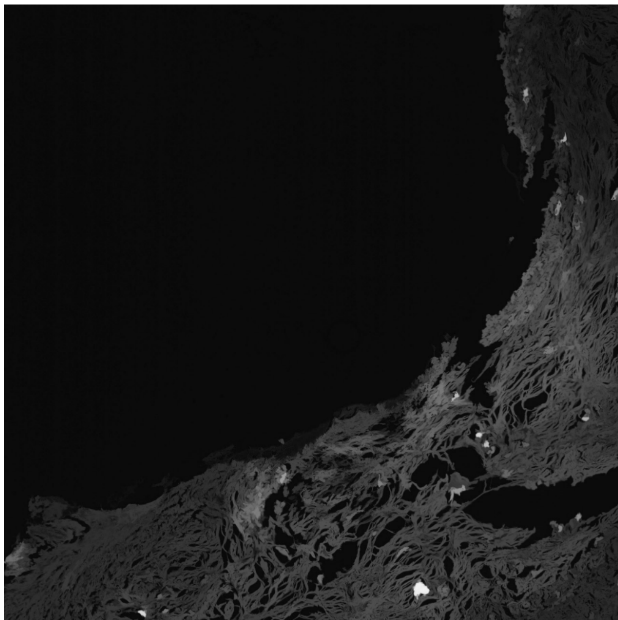
Figure 4.2: 4K mask images used for testing.



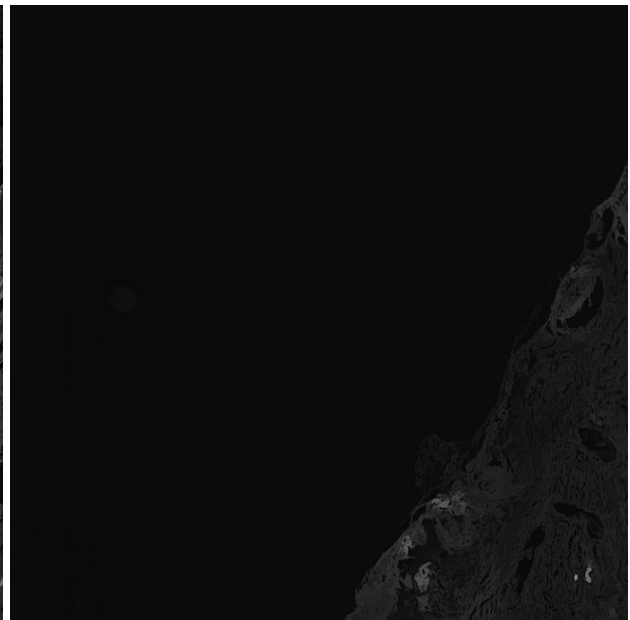
(a) 4K Out-100.



(b) 4k Out-75.

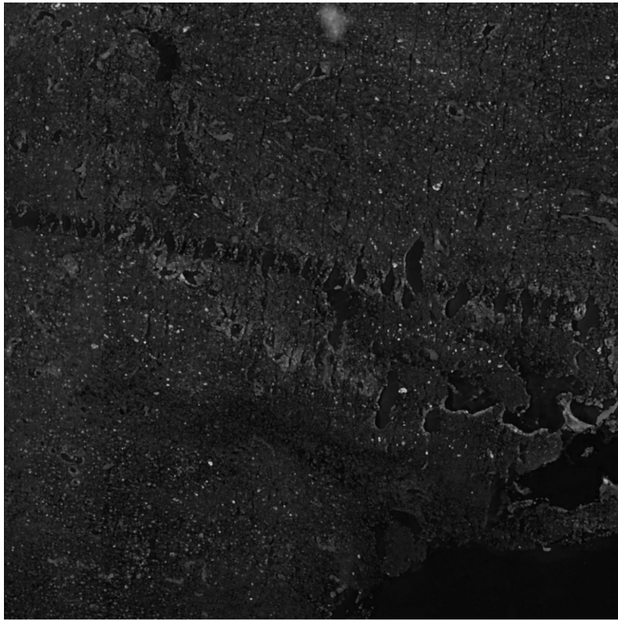


(c) 4K Out-50.

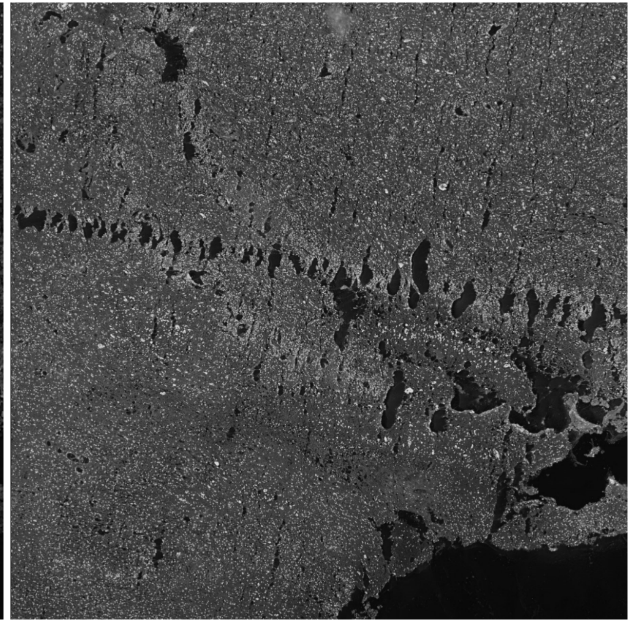


(d) 4k Out-25.

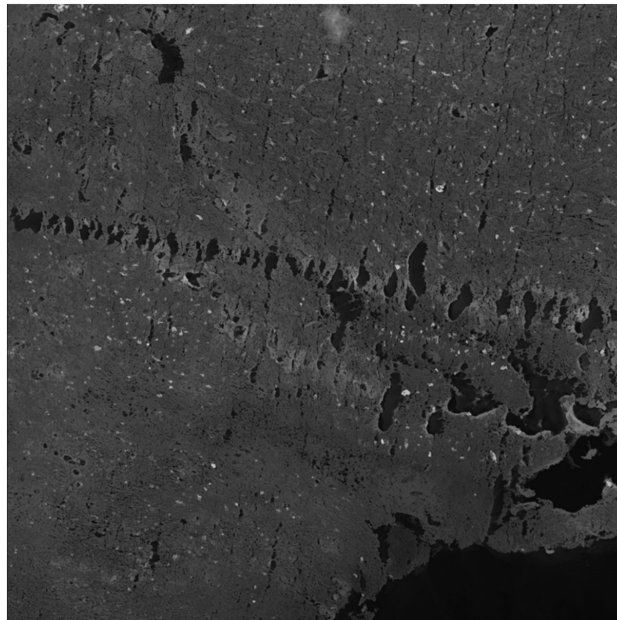
Figure 4.3: 4K output image results after morphological reconstruction tests.



(a) 8K marker.



(b) 8k mask.



(c) 8k Output.

Figure 4.4: 8K images used for testing.

With that information, we can then construct a table of reference for the hardware algorithm running in 150 Mhz that can be seen in Table 4.1. The number of iterations is the number of raster and anti-raster scans, the pixels processed is the image size multiplied by the number of iterations twice (one for raster other for anti-raster), the last column is the time result considering a 150 Mhz clock processing 1 pixel each clock.

Table 4.1: Theoretical time estimation for SR algorithm implemented in hardware at 150 MHz.

Images	**Iterations**	**Pixels processed**	**Time(150 MHz)**
100-Marker (4Kx4K)	18	603.979.776	4,026 s
75-Marker (4Kx4K)	27	905.969.664	6,040 s
50-Marker (4Kx4K)	15	503.316.480	3,355 s
25-Marker (4Kx4K)	18	603.979.776	4,026 s
8K x 8K	18	2.415.919.104	16,106 s

For the software baseline reference, we have the table 4.2 that shows the results for a pure software implementation in both, ARM[1] processor and a CPU[2]. Two solutions were tested, the first one was the sequential reconstruction (SR) algorithm, and the second one was the fast hybrid (FH) algorithm, both were developed using C language. The last column of this table shows the result after executing the function *imreconstruct* in Matlab software, according to [13], this function runs the fast hybrid algorithm proposed by [1]. The pure software implementation and pure hardware estimation are good to have a parameter when comparing with the actual hardware/software co-design solution proposed by this work.

Table 4.2: Pure software time processing in the ARM Cortex-A9 and a CPU in seconds.

Images	**Pure Software ARM (SR Solution)**	**Pure Software CPU (SR Solution)**	**Pure Software ARM (FH Solution)**	**Pure Software CPU (FH Solution)**	**Pure Software CPU (Matlab *imreconstruct* function)**
100-Marker (4Kx4K)	185,979 s	19,544 s	25,683 s	4,137 s	1,490 s
75-Marker (4Kx4K)	278,175 s	28,912 s	26,824 s	4,516 s	1,475 s
50-Marker (4Kx4K)	145,379 s	12,826 s	17,282 s	2,281 s	0,752 s
25-Marker (4Kx4K)	180,575 s	16,325 s	15,782 s	1,991 s	0,726 s
8K x 8K	738,877 s	75,172 s	97,966 s	15,890 s	5,676 s

Comparing the hardware and software references in Tables 4.1 and 4.2, it is noticeable the difference of performance between a hardware running the SR algorithm in 150Mhz and a software running in the ARM processor. Even the FH algorithm executed in the ARM (in theory much faster than SR algorithm [1]) cannot be compared with the SR hardware implementation, the difference can be in the order of 10 times.

[1]ARM Cortex-A9 MPCore, Dual Core, CPU(925 Mhz), 1GB de RAM, Linux OS.

[2]PC Intel® Core™ i5-4210U CPU (1.7GHz $\sim$ 2.7GHz), 8GB de RAM, Windows 10 OS Home Single Language 64 bits.

On the other hand, this hardware solution cannot be implemented in the SoCKit. Two $4K \times 4K$ images (one for the marker and other for the mask) would require $4096 \times 4096 \times 2 \times 8 = 268,435,456$ bits and the Cyclone V used on this work has only $5,570,000$ bits according to Table 2.2. Therefore, the smarter solution that trade-off performance and memory space is the hardware/software co-design.

4.4.1 Hardware/Software co-design using 1 Hardware Module

Table 4.3 shows the time processing results for hardware/software implementation with only one hardware. The best results are in the last column. Each column has a different implementation explained as follow:

- The first column represents the implementation with 1 hardware that processes the sub-images first. Then, after it has finished, the software part performs the border propagation between the sub-images to continue the algorithm;
- The second column is the same as above but with the software running the program in two threads taking advantage of the fact that the ARM processor has two cores inside it;
- The third column takes advantage of the time in which software is idle waiting for the hardware and starts to make the propagation as soon as the hardware has delivered the first results. It has also been used two ARM cores in this implementation.

This Table is a summary of the best implementations with only one hardware and shows the evolution during the research project. It can be seen the software processing time in each implementation decreasing when a better solution has been taken. The author believes that has achieved the best performance using one hardware solution.

The software time consumption in first two implementations was measured as it is. When the hardware has finished, the timer is turned on, and the measurement starts until the software has completed its tasks propagating the pixels throughout the image. In the last approach the time is measured in the same way, even though the software starts to process as soon as the hardware has delivered the first results, it can become confusing to measure the time at the beginning, so it was preferred to do like that in order to have a better comparison with the other two approaches. Indeed, the real-time measurement shown in Table 4.3 for the last column represents only the time in which still remains pixels to be processed by software after the hardware has finished its processing, in the real situation, the software has started to make the pixels propagation much earlier.

Is good to remember from Table 4.2 that the pure software implementation in the ARM processor has a time consumption in the order of dozens of seconds, which is much slower than the hardware. Therefore, even with the software starting to solve the border pixels earlier, it will still be slower than hardware processing.

Table 4.3: Hardware/Software time processing in seconds.

4K Images		1 Hardware (256x256 sub-blocks + overlapping border 16) 150Mhz + Software C ARM (FIFO in the borders)	1 Hardware (256x256 sub-blocks + overlapping border 16) 150Mhz + Software C using 2 threads ARM (FIFO in the borders)	1 Hardware (256x256 sub-blocks + overlapping border 16) 150Mhz + Simultaneously Software C using 2 threads ARM (FIFO in the borders)
100-Marker	Hardware	2,693	2,693	2,693
	Software	2,809	1,921	1,077
	Total	5,502	4,614	3,770
75-Marker	Hardware	2,900	2,900	2,900
	Software	3,427	2,085	1,040
	Total	6,327	4,985	3,940
50-Marker	Hardware	1,944	1,944	1,944
	Software	1,266	0,954	0,459
	Total	3,210	2,898	2,403
25-Marker	Hardware	1,567	1,567	1,567
	Software	1,536	1,036	0,595
	Total	3,103	2,603	2,162

The best result shown in the last column could also be useful for comparison with the pure hardware theoretical estimation time in Table 4.1 or even a pure software FH solution implemented in a typical CPU as shown in Table 4.2, but different aspects of the solution should have to be tested. Also, the hardware processing time could still be decreased using different approaches as shown in next sections.

4.4.2 Performance Impact of Borders Size

In this test, it was analyzed the performance impact when processing the images with different borders size. As shown in Figure 3.8, there is an overlapping border between the sub-images that improves the pixel propagation to next sub-images and reduces the effort in software phase of the algorithm.

As explained in section 3.3.5, the image sizes must be multiple of 16 because the DDR3 Controller has 128 bits of data width in its interface and is addressable in bytes in which each address is written as a multiple of 16. To facilitate the tests, it was tested different borders multiple

of 16. In this test, it was used the best software implementation shown in Table 4.3 and 256×256 pixels sub-images, the overlapping borders size was 0, 16, 32, and 48.

Table 4.4: Hardware/Software borders size influence in time processing in seconds.

4K Images		**1 Hardware (256x256 sub-blocks + overlapping border 0) 150Mhz + Simultaneously Software C using 2 threads ARM (FIFO in the borders)**	**1 Hardware (256x256 sub-blocks + overlapping border 16) 150Mhz + Simultaneously Software C using 2 threads ARM (FIFO in the borders)**	**1 Hardware (256x256 sub-blocks + overlapping border 32) 150Mhz + Simultaneously Software C using 2 threads ARM (FIFO in the borders)**	**1 Hardware (256x256 sub-blocks + overlapping border 48) 150Mhz + Simultaneously Software C using 2 threads ARM (FIFO in the borders)**
100-Marker	Hardware	2,084	2,693	3,358	4,423
	Software	4,704	1,077	0,701	0,531
	Total	6,788	3,770	4,059	4,954
75-Marker	Hardware	2,082	2,900	3,646	4,423
	Software	7,083	1,004	0,752	0,494
	Total	9,165	3,904	4,398	4,917
50-Marker	Hardware	1,524	1,944	2,426	2,967
	Software	1,318	0,459	0,348	0,314
	Total	2,842	2,403	2,774	3,281
25-Marker	Hardware	1,219	1,567	1,959	2,368
	Software	2,458	0,595	0,566	0,581
	Total	3,677	2,162	2,525	2,949

Table 4.4 shows that the bigger is the border size, the more is the hardware time processing. On the other hand, increasing the border size represents a decreasing in the software time to process the neighbor pixels. This pattern just does not happen when the border is 0. In this case, there are two borders for each adjacent sub-image, and also there were no pixels sharing between the sub-images, which leads to the software processing time takes much longer than when there is an overlapping between the images.

For this test, the solution that has better dealt with the trade-off between hardware and software time consumption was the solution with border size equals to 16. Besides its software time is bigger than the tests with border 32 and 48, it has the smallest total processing time amongst all the performed tests.

4.4.3 Performance Impact with Sub-Image Format

In this test, it was analyzed the performance impact with different sub-image division format. To develop this test, it was used 4K images divided in 128×512, 256×256, 512×128, 128×1024, 256×512, and 512×256 as showed in Table 4.5. Important to note that the right half of the table shows subdivisions that are twice the size of the left half, 256×512 is exactly the same as putting together two images of 256×256 pixels each.

The results presented in this Table show that increasing the size of the hardware that is processing the sub-images not necessarily reduces the hardware timing, actually the experiments state the opposite. The bigger is the Sub-Image Format, the worse is the hardware performance. On the other hand, the software time reduces when increasing the hardware size. As it is expected, the bigger is the hardware block size, the smaller is the number of sub-images that will propagate their pixels to the neighborhood, making the software task easier.

The formats that are more vertical like 512×128 and 512×256 showed significantly less processing time than the 128×512 and 128×1024 that are more horizontal. As the algorithm is very data dependent, is hard to find a reason for this. The images that have been used in those tests can be the reason, propagating more vertically than horizontally. For a better analysis, more tests with different images should have to be done.

Table 4.5: Hardware/Software sub-image format influence in time processing in seconds.

4K Images		**Sub-Image Format**					
		128x512	**256x256**	**512x128**	**128x1024**	**256x512**	**512x256**
100-Marker	Hardware	2,833	2,693	2,781	3,047	2,837	2,809
	Software	1,956	1,077	0,763	1,699	0,920	0,708
	Total	4,789	3,770	3,544	4,746	3,757	3,517
75-Marker	Hardware	3,048	2,900	2,872	3,392	3,128	3,050
	Software	1,735	1,004	1,180	2,587	1,029	0,897
	Total	4,783	3,904	4,052	5,979	4,157	3,947
50-Marker	Hardware	2,074	1,944	2,032	2,202	2,059	2,002
	Software	1,025	0,459	0,543	1,029	0,411	0,410
	Total	3,099	2,403	2,575	3,231	2,470	2,412
25-Marker	Hardware	1,656	1,567	1,681	1,730	1,573	1,566
	Software	1,119	0,595	0,769	1,287	0,872	0,688
	Total	2,775	2,162	2,450	3,017	2,445	2,254

In general, the 256×256 division showed the best results for different tissue coverage. Only one image, Marker 100, performed better when 512×128 and 512×256 was chosen, but this can be an isolated case that happens using a more vertical division in those specific images, we cannot generalize for all situations. We can say that the 256×256 division shows the best average performance for different image content amongst all the tests performed in this section.

4.4.4 Hardware/Software Co-Design using 2 Hardware Modules

Figure 4.5 shows the hardware operation when it is selected two hardware modules running at the same time. One hardware block is processing the first half of the image and the second one is processing the last half. Using this approach we can improve the hardware time processing by half of the time as shown in Table 4.6.

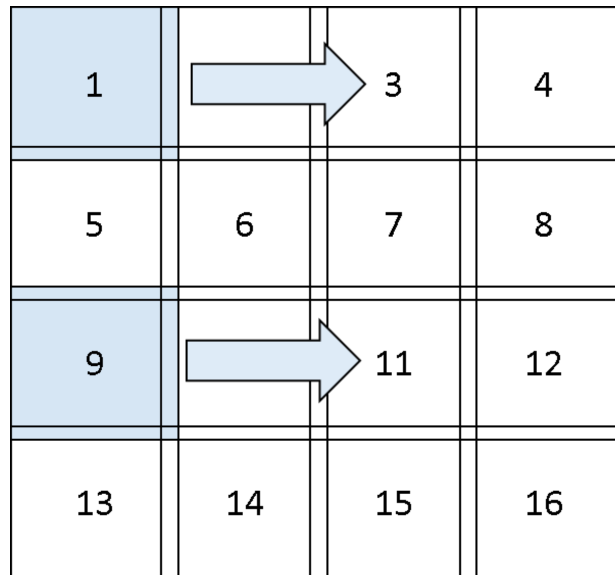


Figure 4.5: Two Hardwares running in a image divided in 16 sub-images.

When two hardware modules are running at the same time, we have a situation where both are competing with the same memory together with the ARM processor. In this case, we can have a performance impact caused by a bottleneck in the memory controller interface being accessed by many sources. As shown in Table 4.6, we can see that the impact was not so significant in our solution. Taking the image Marker-100 as an example, the hardware time consumption using only one block was $2,693$ seconds, while using two was $1,357 + 1,355 = 2,712$ seconds, a 0,7% of difference in time processing.

The hardware block only accesses the memory in three situations as shown in the Main Finite State Machine in Figure 3.7. The first two are reading the Mask and Marker image from DDR3 memory, the third one is writing the final result back to the memory. Most part of hardware processing is made inside the module when it does the Raster and Anti-Raster scans until getting a stable image. Due to this inherent characteristic of the module, the traffic inside the Avalon bus interface is not so intense, the probability of one block accessing the memory exactly at the same time than the other one is very low, leading to a lower impact when inserting two or more modules working together.

It can happen a situation where the processing time of two hardware modules is smaller than a single one as in image Marker-75 shown in Table 4.6. This can be explained in parts due to the non-linearity property of the algorithm, as it is too much data dependent, we can have a situation where the pixels non-propagated from the first half of image to the second one has helped the

algorithm in terms of processing time, reducing the number of iteration performed by the second hardware block.

Table 4.6: Best 1 hardware solution compared to 2 modules in seconds.

Images		1 Hardware (256x256 sub-blocks + overlapping border 16) 150Mhz + Simultaneously Software C using 2 threads ARM (FIFO in the borders)	2 Hardware (256x256 sub-blocks + overlapping border 16) 150Mhz + Simultaneously Software C using 2 threads ARM (FIFO in the borders)	%
100-Marker (4Kx4K)	Hardware1	2,693	1,357	-49,6%
	Hardware2	-	1,355	-
	Software	1,077	1,240	15,1%
	Total	3,770	2,597	-31,1%
75-Marker (4Kx4K)	Hardware1	2,900	1,441	-50,3%
	Hardware2	-	1,448	-
	Software	1,004	1,451	44,5%
	Total	3,904	2,892	-25,9%
50-Marker (4Kx4K)	Hardware1	1,944	0,800	-58,8%
	Hardware2	-	1,175	-
	Software	0,459	0,852	85,6%
	Total	2,403	1,652	-31,3%
25-Marker (4Kx4K)	Hardware1	1,567	0,691	-55,9%
	Hardware2	-	0,901	-
	Software	0,595	1,071	80,0%
	Total	2,162	1,762	-18,5%
8K x 8K	Hardware1	10,756	5,418	-49,6%
	Hardware2	-	5,451	-
	Software	1,947	2,301	18,2%
	Total	12,703	7,719	-39,2%

As explained earlier, the 4K images vary in terms of tissue coverage, from approximately 25%, 50%, 75% and 100%. As showed in Figures 4.1 and 4.2, we can see that the images with 50% and 25% of tissue coverage have more data in the second half of the image, while the images with 75%, 100%, and the 8K image are more uniform. This feature directly impacts the processing time difference between the second hardware module and the first one for those images as shown in Table 4.6.

The software implementation using two hardware blocks is a little bit different than the version with only one module. In this situation, the software needs to monitor both blocks and solve

the borders of the sub-images as soon as each module has delivered some results, alternating between solving the borders of the images processed by the first hardware and the second one. Since each hardware finishes its task approximately at half of the period, and the software part of the algorithm running at the ARM is very slow, the software processing time at Table 4.6 has significantly increased when two blocks are being used. Also, it is worth to note that each hardware processes half of the entire image individually, without pixels propagating between one half and the other, this last propagation is done by the software algorithm and can impact in more time consumption for the software task.

The software time measurement in this situation works the same as with only one hardware. The program starts to measure the time as soon as the first hardware finishes, whichever one. Actually, the software has begun to process almost at the same time that the hardware, but the results presented in Table 4.6 for software time consumption show only the fraction after the first hardware has entirely finished its processing. In other words, the time results shown in this table regarding software part represents only the time that still remains pixels to be processed by software after the hardware processing.

Even with a significant software performance impact, the total time consumption of the algorithm has reduced in 39,2% in the 8K image, and ranging from 18% to 31% to the other 4K images, compared to the best solution using only one hardware.

4.5 Best Hardware/Software Co-Design Performance Comparison

Table 4.7 shows the processing time of the best solution proposed by this work highlighted in gray compared with the hardware estimation processing time from Table 4.1 and the pure software implementation of the FH algorithm in ARM Cortex-A9 processor from Table 4.2. The best solution is the one with 2 hardware modules, 256×256 sub-image division with overlapping border of 16 pixels and a software in C language running simultaneously.

Table 4.7: Best Hardware/Software Co-Design solution compared to only hardware and only software implementation.

Images	Pure Hardware Estimation at 150 Mhz (SR solution)	Pure Software ARM (FH Solution)	Best solution proposed by this work
100-Marker (4Kx4K)	4,026 s	25,683 s	2,597 s
75-Marker (4Kx4K)	6,040 s	26,824 s	2,892 s
50-Marker (4Kx4K)	3,355 s	17,282 s	1,652 s
25-Marker (4Kx4K)	4,026 s	15,782 s	1,762 s
8K x 8K	16,106 s	97,966 s	7,719 s

The actual solution combines the best of both, using an SR implementation in hardware for smaller images, and solving the borders of that sub-images using the FH algorithm in software

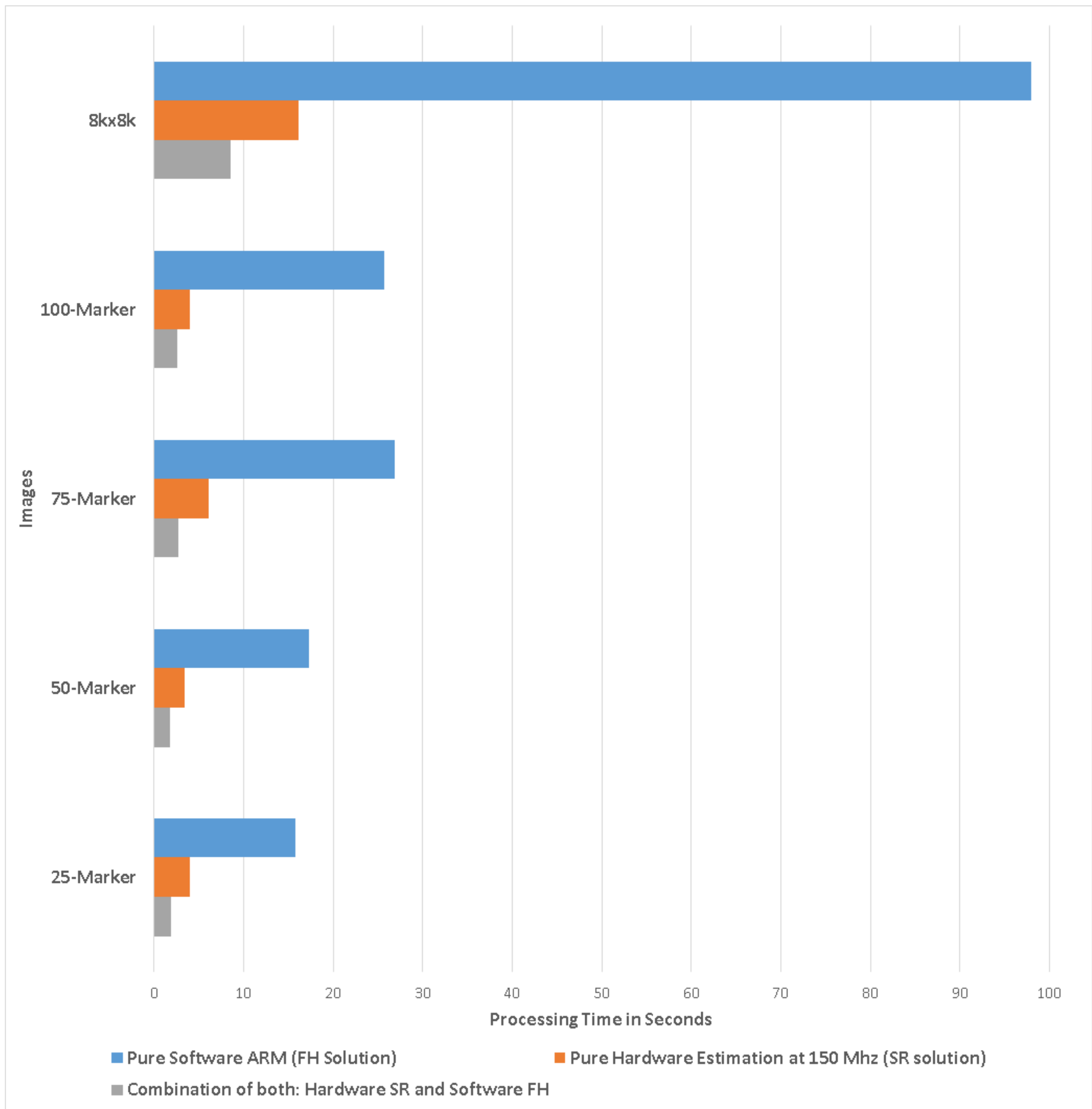


Figure 4.6: Best Hardware/Software Co-Design solution graph comparison in seconds.

running at the ARM processor. Figure 4.6 shows a graphical comparison between those solutions. The results displayed in this figure are very powerful since we are comparing different solutions running on the same platform. The pure software implementation was executed in the same ARM Cortex A9 used to validate this architecture, the same for the pure hardware estimation time that was estimated based on a hardware implementation that has enough memory to process the entire image using SR algorithm. This graph shows that combining both solutions we achieved the best performance than each one individually. Also, we merged the better performance property by designing part of the algorithm in hardware to the flexibility by developing the final part in software. All of this together in an embedded system with very few memory availabilities.

The best solution proposed by this work can also be compared to the morphological reconstruction algorithm being executed in a normal CPU. Table 4.8 shows a comparison between the best solution proposed by this thesis, the FH algorithm implemented in C language running in a currently CPU used in normal laptops[3] and a Matlab program using the *imreconstruct* function running on the same CPU.

Table 4.8: Best Hardware/Software Co-Design solution compared to only hardware and only software implementation.

Images	**Pure Software CPU (FH Solution)**	**Best solution proposed by this work**	**Pure Software CPU (Matlab *imreconstruct* function)**
100-Marker (4Kx4K)	4,137 s	2,597 s	1,490 s
75-Marker (4Kx4K)	4,516 s	2,892 s	1,475 s
50-Marker (4Kx4K)	2,281 s	1,652 s	0,752 s
25-Marker (4Kx4K)	1,991 s	1,762 s	0,726 s
8K x 8K	15,890 s	7,719 s	5,676 s

This Table shows that our solution has achieved a performance improvement of up to 2x the speed of a pure software implementation running in this CPU. The difference is bigger when compared to the $8K \times 8K$ image, showing that for bigger images this difference in time processing is even more noticeable. The Matlab software has a function called ”imreconstruct” that performs exactly the morphological reconstruction algorithm. This function receives as inputs the Marker and Mask images and outputs the image reconstructed as a result. According to the reference [13], this function runs the Fast Hybrid algorithm.

For a reason that cannot be explained by this author, since the real implementation of ”imreconstruct” function is a property of Matlab MathWorks company, this function runs 3x faster than the FH software implementation developed in C language used in this thesis. Even in this case, our results can still be comparable to Matlab function because the difference in time consumption is only 36% more (For $8K \times 8K$ image) using our approach. Also, the Table 4.8 shows that the bigger is the image, the smaller is the performance difference from Matlab and the solution proposed by this work. Tests with more images bigger than 8K should have to be done to verify if the Matlab solution could maintain the performance gain compared to our solution.

Is good to remember that the hardware part of our system is running at 150 Mhz frequency and the software is being executed in a 32-bit Dual core RISC processor running at 925 Mhz, while the CPU is a 64-bit Dual core CISC processor that can run up to 2,7 GHz. Also, according to [45], the CPU wastes 15W of energy while ours is less than 1W (Table 4.9), excluding any power comparison between a whole laptop and an FPGA board.

Even though a Matlab is a very well established solution, the purpose of this work is not to substitute it. The main goal is to offer a good implementation alternative. These results confirm

[3]PC Intel® Core™ i5-4210U CPU (1.7GHz $\sim$ 2.7GHz), 8GB de RAM, SO Windows 10 Home Single Language 64 bits.

that our solution is a very good one for embedded systems, energy efficient applications and low memory consumption (Table 4.9).

4.6 Resources Utilization for two Hardware Modules Implementation

The hardware resources utilization for our solution implemented in the SoCKit FPGA board can be seen in Table 4.9. This table shows that the main part of the hardware resources is concentrated in block memory bits, with 61% of utilization. On the other hand, a small part of logic has been used, 22% only. A solution with 3 hardware blocks could also be implemented in SoCKit board in terms of memory utilization, but when connecting three hardware modules using Qsys tool, the Altera Quartus software creates a huge amount of memory blocks and registers in the interconnection that exceeds the maximum allowed. A solution for this situation is to design our own interconnection module that will deal with three Avalon masters using only one Avalon slave. Unfortunately, the author did not have enough time to implement this version, leaving this alternative for future implementations.

Table 4.9: Hardware resources utilization summary for two hardwares architecture.

Module	**Logic utilization (41,910 max)**	**Block memory bits utilization (5,662,720 max)**	**Power consumption**
Raster Controller 1	1,345	2,048	15.59 mW
Raster&Anti-Raster 1	804	1,337,352	61.28 mW
Raster Controller 2	1,311	2,048	15.82 mW
Raster&Anti-Raster 2	794	1,337,352	60.94 mW
Interconnect Bridge	1,350	524,288	35.41 mW
AXI-Avalon Bridge	225	0	1.87 mW
DDR3 Memory Controller	3,390	225,040	57.25 mW
Others (JTAG, I/O etc.)	208	0	396,10 mW
Total	9,427 (22%)	3,428,128 (61%)	644.26 mW

Chapter 5

Conclusion

This Master's thesis presents a novel implementation of the morphological image reconstruction algorithm using a hardware/software approach in an FPGA based embedded system. The work was based in an already established solution made by the Mechatronic Systems Ph.D. student Oscar Eduardo Anacona Mosquera from the Department of Mechanical Engineering of the University of Brasilia [30].

This work improves the actual solution providing new features that allow the user to process images much bigger than the 288×288 maximum size from the last architecture, and also obtain performance gains comparable even with regular CPU solutions.

This thesis combines a Sequential Reconstruction algorithm implementation in hardware for smaller images, solving the borders of that sub-images in software running in a Dual Core ARM® Cortex™ A9 processor. This approach is proved to be an excellent solution for an embedded system, with performance comparable even to powerful CPUs, consuming very few memory resources and still being able to process up to $64k \times 64k$ pixels image.

The hardware architecture runs at 150 Mhz, uses Avalon protocol to communicate, has a DDR3 memory of 1GB to store temporary images and is connected to an ARM Cortex-A9 through an AMBA® AXI3 bus. The software part was developed in C language, it runs in this processor at 925 Mhz and has another DDR3 memory of 1GB.

The final solution was thoroughly verified by comparing exhaustive tests to the MATLAB® results. The combination of hardware and software for a morphological reconstruction algorithm was proved to be an optimal solution in this specific problem, combining both solutions we achieved a performance that is better than each one individually.

The best solution proposed by this work has achieved a performance improvement of around 2x compared to the best theoretical solution possible of the sequential morphological reconstruction algorithm implemented in a hardware that runs at 150Mhz, a speedup of up to 12x in relation to the fast hybrid reconstruction algorithm proposed by Vincent [1] being executed in an ARM® Cortex™ A9 processor, and even a speedup of up to 2x in comparison with the same algorithm running in an Intel® Core™ i5 CPU.

The final tests to validate the solution show correct results for 8 bits grayscale images of up to 8192x8192 pixels (8k images). To the best knowledge of the author, this is the first hardware/-software implementation of the morphological image reconstruction algorithm described in the literature.

There are some specific aspects in this research that were not covered or needs to have a a more profound investigation for better conclusions. For future implementations we can highlight the following:

- Implement three hardware modules in the SoCKit FPGA and analyze the performance with the results provided by this thesis. This would require to develop an own Avalon interface connection with three masters and one slave to reduce the area usage of the SoCKit FPGA;
- The solution proposed by this thesis has a very good justification for using dynamic partial reconfiguration in FPGA;
- Do tests with the maximum theoretical size proposed by this thesis, $64K \times 64K$ pixels image, and compare the result obtained with others platforms. This would require an FPGA board with more external memory available.
- Investigate with more tests the best sub-image division for the hardware processing that combined with the software gives the best solution. This work has found the best sub-image division is 256×256 but many others combinations were not tested;
- Using an FPGA with more internal memory capacity, test and investigate the performance influence of implementation with different numbers of hardware modules, with different size. For instance, an implementation with 8 hardware modules of 128×128 pixels size each requires approximately the same memory resources than 2 hardware modules of 256×256 and could be implemented in the same SoCKit FPGA used in this thesis;
- As the Fast Hybrid algorithm is proved to have a better performance compared to the sequential algorithm, implement a hardware version that performs a Fast Hybrid algorithm instead of the sequential one;
- Implement the last part of the algorithm, the propagation between the borders, in hardware and analyze if there is any performance improvement;

REFERENCES

[1] VINCENT, L. Morphological grayscale reconstruction in image analysis: Applications and efficient algorithms. *IEEE Transactions on Image Processing*, v. 2, n. 2, p. 176–201, 1993.

[2] TEODORO, G. et al. A fast parallel implementation of queue-based morphological reconstruction using gpus. *Emory University, Center for Comprehensive Informatics Technical Report*, 2012.

[3] GOMES, J. M. *Execução Eficiente do Padrão de Propagação de Ondas Irregulares na Arquitetura Many Integrated Core*. Dissertação (Mestrado) — Universidade de Brasília, Programa de Pós-graduação em Informática, Departamento de Ciência da Computação, 2016.

[4] ALTERA. Cyclone v hard processor system technical reference manual. In: *Altera's documentation*. [S.l.]: Altera® Corporation, 2015.

[5] TERASIC. Sockit user manual. In: *Terasic's documentation*. [S.l.]: Terasic, 2015.

[6] TEODORO, G. et al. Efficient irregular wavefront propagation algorithms on hybrid cpu–gpu machines. *Parallel computing*, Elsevier, v. 39, n. 4-5, p. 189–211, 2013.

[7] TERASIC. De0-nano-soc, my first hps-fpga manual. In: *Terasic's documentation*. [S.l.]: Terasic, 2015.

[8] ARM. Cortex™-a9 mpcore®-technical reference manual. In: *ARM's documentation*. [S.l.]: ARM®, 2012.

[9] JÄHNE, B. *Digital Image Processing*. [S.l.]: Springer, 2005.

[10] GONZALEZ, R. C. *Digital Image Processing Using MATLAB*. second. [S.l.]: Gatesmark Publishing, 2009.

[11] SOILLE, P. *Morphological Image Analysis: Principles and Applications*. [S.l.]: Springer-Verlag, 2003.

[12] SERRA, J. *Image Analysis and Mathematical Morphology*. [S.l.]: London: Academic Press, 1983.

[13] GONZALEZ, R. C.; WOODS, R. E.; EDDINS, S. L. Morphological reconstruction. In: *Matlab® documentation*. [S.l.]: MathWorks, 2010.

[14] SALEMBIER, P. et al. Morphological operators for image and video compression. *IEEE Transactions on Image Processing*, v. 5, p. 881–898, 1996.

[15] SOILLE P.; PESARESI, M. Advances in mathematical morphology applied to geoscience and remote sensing. *IEEE Transactions on Geoscience and Remote Sensing*, IEEE, v. 40, 2002.

[16] PESARESI, M.; BENEDIKTSSON, J. A. A new approach for the morphological segmentation of high-resolution satellite imagery. *IEEE transactions on Geoscience and Remote Sensing*, IEEE, v. 39, n. 2, p. 309–320, 2001.

[17] AREFI, H.; HAHN, M. A morphological reconstruction algorithm for separating off-terrain points from terrain points in laser scanning data. *International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences*, v. 36, n. 3/W19, p. 120–125, 2005.

[18] RUUSUVUORI, P. et al. Evaluation of methods for detection of fluorescence labeled subcellular objects in microscope images. *BMC Bioinformatics*, BioMed Central, v. 11, 12 2010.

[19] MENDONCA A.M.; CAMPILHO, A. Segmentation of retinal blood vessels by combining the detection of centerlines and morphological reconstruction. *IEEE Transactions on Medical Imaging*, IEEE, v. 25, 2006.

[20] KARAS, P. Efficient computation of morphological greyscale reconstruction. In: SCHLOSS DAGSTUHL-LEIBNIZ-ZENTRUM FUER INFORMATIK. *OASIcs-OpenAccess Series in Informatics*. [S.l.], 2011. v. 16.

[21] ADAMS, L. Choosing the right architecture for real-time signal processing designs. In: *Texas Instruments documentation*. [S.l.]: Texas Instruments, 2002.

[22] SAMPAIO, R. C.; BERGER, P. A.; P.JACOBI, R. Hardware and software co-design for the aac audio decoder. In: *Proceedings of the 25th Synposium on Circuits and Systems Design*. [S.l.: s.n.], 2012. p. 1–6.

[23] THURLEY M. J.; DANELL, V. Fast morphological image processing open-source extensions for gpu processing with cuda. *IEEE Journal of Selected Topics in Signal Processing*, v. 6, 11 2012.

[24] QUESADA-BARRIUSO, P. et al. Wavelet-based classification of hyperspectral images using extended morphological profiles on graphics processing units. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 2015.

[25] TEODORO, G. et al. Comparative performance analysis of intel (r) xeon phi (tm), gpu, and cpu: a case study from microscopy image analysis. In: IEEE. *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. [S.l.], 2014. p. 1063–1072.

[26] GOMES, J. M. et al. Efficient irregular wavefront propagation algorithms on intel(r) xeon phi(tm). In: *Symposium on Computer Architecture and High Performance Computing*. [S.l.: s.n.], 2015. v. 2015, p. 25–32.

[27] ROBINSON, K.; WHELAN, P. F. Efficient morphological reconstruction: a downhill filter. *Pattern Recognition Letters*, Elsevier, v. 25, n. 15, p. 1759–1767, 2004.

[28] TEODORO, G. et al. High-throughput analysis of large microscopy image datasets on cpu-gpu cluster platforms. In: 2013 IEEE 27TH INTERNATIONAL SYMPOSIUM ON PARALLEL AND DISTRIBUTED PROCESSING. [S.l.], 2013.

[29] JIVET, I.; BRINDUSESCU, A.; BOGDANOV, I. Fpga implementation of image morphological decomposition with reconstruction. p. 385–390, 01 2008.

[30] ANACONA, O. M. et al. Efficient hardware implementation of morphological reconstruction based on sequential reconstruction algorithm. In: ACM. *Proceedings of the 30th Symposium on Integrated Circuits and Systems Design: Chip on the Sands*. [S.l.], 2017. p. 162–167.

[31] BAUMANN, D.; TINEMBART, J. Designing mathematical morphology algorithms on fpgas: An application to image processing. In: GAGALOWICZ, A.; PHILIPS, W. (Ed.). *Computer Analysis of Images and Patterns*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005. p. 562–569. ISBN 978-3-540-32011-1.

[32] STRUCTURING Elements. `https://uk.mathworks.com/help/images/structuring-elements.html`. Accessed: 2018-03-01.

[33] MORPHOLOGY-BASED Operations. `http://www.mif.vu.lt/atpazinimas/dip/FIP/fip-Morpholo.html#Heading95`. Accessed: 2018-03-01.

[34] SUNG, J.-M. et al. Morphological image reconstruction using directional propagation. *Journal of Imaging Science and Technology*, Society for Imaging Science and Technology, v. 59, 09 2015.

[35] SAMPAIO, R. C. *Coprojeto de um Decodificador de Áudio AAC-LC em FPGA*. Dissertação (Mestrado) — Universidade de Brasília, Programa de Pós-graduação em Informática, Departamento de Ciência da Computação, 2013.

[36] CROCKETT, L. H. et al. *The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc*. [S.l.]: Strathclyde Academic Media, 2014.

[37] PRAKASH, P. et al. Multicore socs stay a step ahead of soc fpgas. In: *Texas Instruments documentation*. [S.l.]: Texas Instruments, 2016.

[38] ALTERA. What is an soc fpga? In: *Altera's documentation*. [S.l.]: Altera® Corporation, 2014.

[39] ARM. The arm cortex-a9 processors - white paper. In: *ARM's documentation*. [S.l.]: ARM®, 2009.

[40] ALTERA. Soc fpga arm cortex-a9 mpcore processor advance information brief. In: *Altera's documentation*. [S.l.]: Altera® Corporation, 2012.

[41] ALTERA. Avalon interface specifications. In: *Altera's documentation*. [S.l.]: Altera®, 2015.

[42] ALTERA. Cyclone v device datasheet. In: *Altera's documentation*. [S.l.]: Altera®, 2016.

[43] LANDRY, R. *FPGA Prototyping as a Verification Methodology*. https://www.design-reuse.com/articles/13212/fpga-prototyping-as-a-verification-methodology.html. Accessed: 2018-04-20.

[44] SALTZ, J. et al. Multi-scale, integrative study of brain tumor: In silico brain tumor research center. 01 2010.

[45] INTEL. *Intel® Core™ i5-4210U Processor specifications*. https://ark.intel.com/products/81016/Intel-Core-i5-4210U-Processor-3M-Cache-up-to-2_70-GHz. Accessed: 2018-06-30.

APPENDIX

I. ARM CORTEX-A9 MEMORY MAP

Table I.1: Memory Map for Peripheral Region of MPU Subsystem.

Interface	Name	Start Address	End Address	Size
HPS2FPGASLAVES	FPGA Slaves Accessed Via HPS-to-FPGA AXI Bridge	0xC0000000	0xFBFFFFFF	960 MB
STM	Space Trace Macrocell	0xFC000000	0xFEFFFFFF	48 MB
DAP	Debug Access Port	0xFF000000	0xFF1FFFFF	2 MB
LWFPGASLAVES	FPGA Slaves Accessed with Lightweight HPS-to-FPGA Bridge	0xFF200000	0xFF3FFFFF	2 MB
LWHPS2FPGAREGS	Lightweight HPS-to-FPGA bridge global programmer's view (GPV) registers	0xFF400000	0xFF47FFFF	1 MB
HPS2FPGAREGS	HPS-to-FPGA bridge GPV registers	0xFF500000	0xFF507FFF	1 MB
FPGA2HPSREGS	FPGA-to-HPS bridge GPV registers	0xFF600000	0xFF67FFFF	1 MB
EMAC0	Ethernet MAC 0	0xFF700000	0xFF701FFF	8 KB
EMAC1	Ethernet MAC 1	0xFF702000	0xFF703FFF	8 KB
SDMMC	SD/MMC	0xFF704000	0xFF7043FF	4 KB
QSPIREGS	Quad SPI flash controller registers	0xFF705000	0xFF7050FF	4 KB
FPGAMGRREGS	FPGA manager registers	0xFF706000	0xFF706FFF	4 KB
ACPIDMAP	ACP ID mapper registers	0xFF707000	0xFF707FFF	4 KB
GPIO0	GPIO 0	0xFF708000	0xFF70807F	4 KB
GPIO1	GPIO 1	0xFF709000	0xFF70907F	4 KB
GPIO2	GPIO 2	0xFF70A000	0xFF70A07F	4 KB
L3REGS	L3 interconnect GPV	0xFF800000	0xFF87FFFF	1 MB

NANDDATA	NAND flash controller data	0xFF900000	0xFF9FFFFF	64 KB
QSPIDATA	Quad SPI flash data	0xFFA00000	0xFFAFFFFF	1 MB
USB0	USB 2.0 OTG 0 controller registers	0xFFB00000	0xFFB3FFFF	256 KB
USB1	USB 2.0 OTG 1 controller registers	0xFFB40000	0xFFB7FFFF	256 KB
NANDREGS	NAND flash controller registers	0xFFB80000	0xFFB807FF	64 KB
FPGAMGRDATA	FPGA manager configuration data	0xFFB90000	0xFFB90003	4 KB
CAN0	CAN 0 controller registers	0xFFC00000	0xFFC001FF	4 KB
CAN1	CAN 1 controller registers	0xFFC01000	0xFFC011FF	4 KB
UART0	UART 0	0xFFC02000	0xFFC020FF	4 KB
UART1	UART 1	0xFFC03000	0xFFC030FF	4 KB
I2C0	I2C controller 0	0xFFC04000	0xFFC040FF	4 KB
I2C1	I2C controller 1	0xFFC05000	0xFFC050FF	4 KB
I2C2	I2C controller 2	0xFFC06000	0xFFC060FF	4 KB
I2C3	I2C controller 3	0xFFC07000	0xFFC070FF	4 KB
SPTIMER0	SP Timer 0	0xFFC08000	0xFFC080FF	4 KB
SPTIMER1	SP Timer 1	0xFFC09000	0xFFC090FF	4 KB
SDREGS	SDRAM controller subsystem registers	0xFFC20000	0xFFC3FFFF	128 KB
OSC1TIMER0	OSC1 Timer 0	0xFFD00000	0xFFD000FF	4 KB
OSC1TIMER1	OSC1 Timer 1	0xFFD01000	0xFFD010FF	4 KB
L4WD0	Watchdog Timer 0	0xFFD02000	0xFFD020FF	4 KB
L4WD1	Watchdog Timer 1	0xFFD03000	0xFFD030FF	4 KB
CLKMGR	Clock manager	0xFFD04000	0xFFD041FF	4 KB
RSTMGR	Reset manager	0xFFD05000	0xFFD050FF	4 KB
SYSMGR	System manager	0xFFD08000	0xFFD0BFFF	16 KB
DMANONSECURE	DMA nonsecure registers	0xFFE00000	0xFFE00FFF	4 KB
DMASECURE	DMA secure registers	0xFFE01000	0xFFE01FFF	4 KB
SPIS0	SPI slave 0	0xFFE02000	0xFFE0207F	4 KB
SPIS1	SPI slave 1	0xFFE03000	0xFFE0307F	4 KB
SPIM0	SPI master 0	0xFFF00000	0xFFF000FF	4 KB
SPIM1	SPI master 1	0xFFF01000	0xFFF010FF	4 KB

SCANMGR	Scan manager registers	0xFFF02000	0xFFF0201F	4 KB
ROM	Boot ROM	0xFFFD0000	0xFFFDFFFF	64 KB
MPU	MPU registers	0xFFFEC000	0xFFFEDFFF	8 KB
MPUL2	MPU L2 cache controller registers	0xFFFEF000	0xFFFEFFFF	4 KB
OCRAM	On-chip RAM	0xFFFFF000	0xFFFFFFFF	64 KB

Table I.2: Memory Map for MPU Register Space, located between 0xFFFEC000 and 0xFFFECFFF.

Module Instance	**Description**	**Start Address**	**End Address**
SCU	This address space is allocated for the Snoop Control Unit registers.	0xFFFEC000	0xFFFEC0FF
GIC	This address space is allocated for the General Interrupt Controller (GIC) registers.	0xFFFEC100	0xFFFEC1FF
Global Timer	This address space is allocated for the Global Timer registers.	0xFFFEC200	0xFFFEC2FF
Reserved	This address space is reserved.	0xFFFEC300	0xFFFEC5FF
Private Timers and Watchdog Timers	This address space is allocated for private timers and watchdog timers.	0xFFFEC600	0xFFFEC6FF
Reserved	This address space is reserved. Caution: Any access to this region causes a SLVERR abort exception.	0xFFFEC700	0xFFFEC7FF
Interrupt Distributor	This address space is allocated for the interrupt distributor.	0xFFFED000	0xFFFEDFFF
Reserved	This address space is reserved.	0xFFFEE000	0xFFFEEFFF