**ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ**

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

# FPGA Implementation of Computer Vision Algorithms: Application on Linear Time Selection Algorithm

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΓΕΩΡΓΙΟΣ ΤΖΙΜΠΡΑΓΟΣ

**Επιβλέπων:** Δημήτριος Σούντρης

Επ.Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2012

**ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ**

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

# FPGA Implementation of Computer Vision Algorithms: Application on Linear Time Selection Algorithm

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΓΕΩΡΓΙΟΣ ΤΖΙΜΠΡΑΓΟΣ

**Επιβλέπων:** Δημήτριος Σούντρης
Επ.Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την …. Ιουλίου 2012.

| | | |
|---|---|---|
| ..... | ..... | ..... |
| Δημήτριος Σούντρης | Κιαμάλ Ζ. Πεκμεστζή | Γεώργιος Οικονομάκος |
| Επ. Καθηγητής Ε.Μ.Π. | Καθηγητής Ε.Μ.Π. | Επ. Καθηγητής Ε.Μ.Π. |

.

Αθήνα, Ιούλιος 2012

.....

ΓΕΩΡΓΙΟΣ ΤΖΙΜΠΡΑΓΟΣ

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

© 2012 – All rights reserved

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα.

Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν τον συγγραφέα και δεν πρέπει να ερμηνευθεί ότι αντιπροσωπεύουν τις επίσημες θέσεις του Εθνικού Μετσόβιου Πολυτεχνείου.

# Contents

**Preface** ..... **07**

**Abstract** ..... **11**

**Chapter 1 : Introduction** ..... **15**

**1.1 Introduction to FPGAs** ..... **15**

**1.2 Introduction to Computer Vision Algorithms** ..... **22**

**Chapter 2 : Behind the Project** ..... **25**

**2.1 Applications** ..... **25**

**2.2 Related Work** ..... **30**

**Chapter 3 : Hardware and Software** ..... **33**

**3.1 The Platform** ..... **33**

**3.2 Software Design Flow** ..... **45**

**Chapter 4 : Algorithm Analysis** ..... **51**

**Chapter 5 : Implementation** ..... **57**

**5.1 Version 1 : Implementation with minimum memory requirements** ..... **57**

**5.1.1 Median_top component** ..... **58**

**5.1.2 M_finder component** ..... **61**

**5.1.3 Mergesort component** ..... **64**

**5.1.4 Compexch component** ..... **67**

**5.1.5 Init_RAM component** ..... **69**

**5.1.6 SRAM component** ..... **70**

**5.1.7 S_RAMEM component** ..... **71**

**5.2 Version 2 : High-Performance** ..... **73**

**5.3 Comparison between Version 1 vs. Version 2** ..... **75**

**Chapter 6 : Communication** ..... **79**
**6.1 Integration with Ethernet** ..... **79**
**6.2 Arbiter** ..... **85**

**Chapter 7 : Implementation Results** ..... **87**
**7.1 Performance Results** ..... **88**
**7.2 Power and Leakage Results** ..... **95**
**7.3 Implementation Reports** ..... **97**
**7.4 Conclusion** ..... **99**

**References** ..... **103**

# Preface

The treatment of huge data sets or the application of computer vision algorithms are natural candidates for high performance computing systems, because of their complexity.

Our goal is to achieve performance improvements and cut down the runtime. The idea is to run collaboratively computer and FPGA device.

In this thesis, we focus on the FPGA implementation of a Linear Time Selection Algorithm, which have applications on both huge lists of data and computer vision algorithms, and its integration with *ethmac* for runtime communication.

The thesis is organized as follows :

In *Chapter 1* we introduce the reader to the FPGAs' technology and Computer Vision Algorithms.

*Chapter 2* describes the applications of Linear Time Selection Algorithm and the motivation behind this project. It also gives a description of previous related work.

Hardware description languages (HDLs), such as Very High Speed Integrated Circuit HDL (VHDL) and Verilog, are high level programming languages commonly used to describe the functionality of the circuits. After describing the circuit using HDL, the designs are simulated and synthesized. A more detailed explanation of the design steps is provided in *Chapter 3*. VHDL, Xilinx ISE 13.4 Design Suite and Modeltech Modelsim have been used in this thesis.

In *Chapter 4*, a brief description of the Linear Time Selection Algorithm and its mathematical background is provided to the reader.

In *Chapter 5* we analyze two alternative instantiations of target architecture, which were developed at this thesis. The first one is rather memory efficient, while the second one is more time efficient.

*Chapter 6* is devoted to the runtime communication of the FPGA device and thelocal host. Firstly, there is given a brief description about the integration of our implemented core with the ethmac/driver. This if followed by a presentation of a controller that we have developed. The previously mentioned component is responsible for the arbitration when more than one cores are attached to the ethmac.

In *Chapter 7* we present the implementation results and we compare them with the results of other alternatives, such as the C/C++ implementation of the same algorithm or the use of other algorithms instead.

# Acknowledgments

For the preparation of this thesis I would like to thank primarily Professor Dimitrios Soudris not only for the opportunity he gave me to work on a subject that really interests me but also for his overall help.

I would also like to express my appreciation on my co-workers on the microlab for their advices and especially Dionysis Diamantopoulos and Kostas Siozios for their invaluable guidance and motivation throughout this work.

Last but not least, a special thanks to my family and friends for their love and support.

# Abstract

Data intensive applications (e.g. computer vision and data management algorithms) impose considerable performance overheads that rarely are sufficiently implemented onto general-purpose computers. Instead of this, more advanced implementation medium are absolutely required in order to support sufficient performance. An example affects the usage of customized hardware accelerators, where the most computational intensive kernels are executed.

The goal of this thesis is to provide an efficient hardware/software co-design implementation of such a data intensive algorithm. For this purpose, all the timing critical kernels, as they already derived from profiling procedure, were implemented onto reconfigurable hardware. More specifically, the target reconfigurable medium is a state-of-the-art Xilinx Virtex-6 (xc6vlx240t), whereas regarding the rest kernels (non-timing critical) are actually mapped onto a general-purpose CPU.

Even though the introduced solution is applicable to various application intensive applications, at this thesis we are dealing with the implementation of Median algorithm targeting to two different application domains: (i) the implementation of Median filtering targeting to remove impulsive noise from data, and (ii) an algorithm for data querying.

Since the scope of this thesis affects the sufficient implementation of this algorithm, in terms both of performance and amount of utilized resources, two different versions of this algorithm were developed. More specifically, the first of them employees the minimum amount of memory blocks, whereas the second one is characterized by increased performance. Note that the second implementation is very important due to inherent limitation about memory blocks found in FPGAs. Additionally, we have to highlight that based on our exploration results we achieve significant increased performance compared to the software implementation (C++).



**Keywords:** FPGA, Median, Selection, Query, Order, runtime Communication, Integration, Arbiter, Virtex6, Computer Vision, co-Design.

# Περίληψη

Εφαρμογές με υψηλές υπολογιστικές απαιτήσεις, π.χ αλγόριθμοι για διαχείριση δεδομένων, αλγόριθμοι ρομποτική όρασης, απαιτούν πολύ καλές επιδόσεις που σπάνια συναντώνται σε προσωπικούς υπολογιστές. Έτσι δημιουργείται η ανάγκη της υλοποίησής τους σε πιο εξελιγμένα μέσα για την αποδοτική τους εκτέλεση. Μια τέτοια υλοποίηση επιτυγχάνεται με τη χρήση customized hardware επιταχυντών, στους οποίους αναθέτουμε την εκτέλεση των πιο βαριών υπολογιστικά kernels.

Ο στόχος αυτής της εργασίας είναι η παροχή ενός αποτελεσματικού συν-σχεδιασμού hardware/software για την καλύτερη υλοποίηση τέτοιων απαιτητικών αλγορίθμων. Για το σκοπό αυτό, ύστερα από την διαδικασία του profiling που έγινε στους κώδικές μας, επιλέξαμε να υλοποιήσουμε σε επαναπροσαρμόσιμο hardware, τα kernels με το μεγαλύτερο critical path. Η πλατφόρμα που χρησιμοποιήσαμε για την υλοποίηση αυτή είναι το Xilinx Virtex-6 (xc6vlx240t).

Σε σύγκριση με τις ήδη υπάρχουσες υλοποιήσεις, σε αυτή την εργασία ασχολούμαστε με την υλοποίηση του Median αλγόριθμου, στοχεύοντας σε δύο διαφορετικούς τομείς: (i) την υλοποίηση ενός φίλτρου για την αφαίρεση θορύβου από εικόνες (ii)και την υλοποίηση αλγορίθμου επιλογής με στόχο τη διερεύνηση σε λίστες δεδομένων.

Καθώς, ο στόχος μας επηρεάζει τον τρόπο υλοποίησης τόσο ως προς την επίδοση όσο και ως προς το σύνολο των απαιτούμενων πόρων, αναπτύξαμε δυο διαφορετικές εκδόσεις αυτού του αλγορίθμου. Η πρώτη υλοποίηση χρησιμοποιεί τον ελάχιστο αριθμό πόρων (μνήμη) και η δεύτερη χαρακτηρίζεται από πιο υψηλές επιδόσεις. Αξιοσημείωτη είναι η εφαρμογή ακόμα και της δεύτερης υλοποίησης σε μεγάλο πλήθος αριθμών, δεδομένου των περιορισμένων πόρων μνήμης των FPGA .

Να αναφέρουμε τέλος και την μεγάλη αύξηση της επίδοσης που επιτύχαμε σε σύγκριση με την εκτέλεση στην CPU .

**Λέξεις Κλειδιά:** FPGA, Median, Selection, Query, Order, runtime Communication, Integration, Arbiter, Virtex6, Computer Vision, co-Design.

# Chapter 1

# Introduction

In this chapter we give some information about what's a FPGA, its history and the future challenges. Also the reader will be informed about Computer Vision Algorithms and why FPGA implementation was chosen.

## 1.1 Introduction to FPGAs

### What is a FPGA?

The field-programmable gate array (FPGA) is a semiconductor device that can be programmed after manufacturing. Instead of being restricted to any predetermined hardware function, an FPGA allows you to program product features and functions, adapt to new standards, and reconfigure hardware for specific applications even after the product has been installed in the field—hence the name "field-programmable". You can use an FPGA to implement any logical function that an application-specific integrated circuit (ASIC) could perform, but the ability to update the functionality after shipping offers advantages for many applications.

Unlike previous generation FPGAs using I/Os with programmable logic and interconnects, today's FPGAs consist of various mixes of configurable embedded SRAM, high-speed transceivers, high-speed I/Os, logic blocks, and routing. Specifically, an FPGA contains programmable logic components called logic elements (LEs) and a hierarchy of reconfigurable interconnects that allow the LEs to be physically connected. You can configure LEs to perform complex combinational functions, or merely simple logic gates like AND and XOR. In most FPGAs, the logic blocks also include memory elements, which may be simple flipflops or more complete blocks of memory.

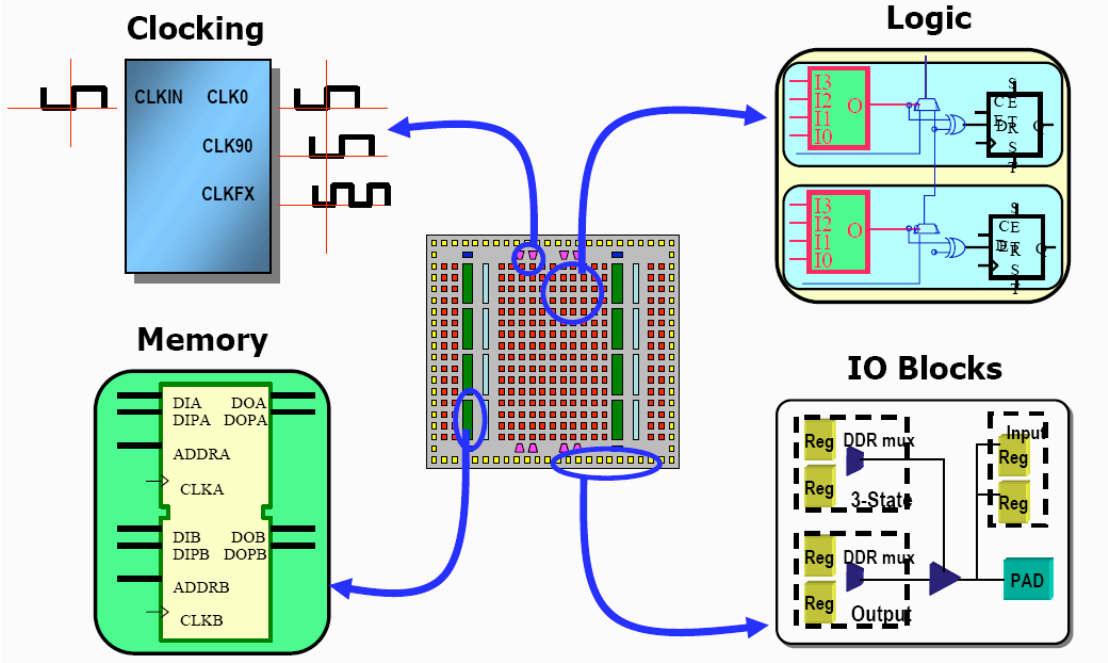



**Figure 1.1** FPGA structure

As FPGAs continue to evolve, the devices have become more integrated. Hard intellectual property (IP) blocks built into the FPGA fabric provide rich functions while lowering power and cost and freeing up logic resources for product differentiation. Newer FPGA families are being developed with hard embedded processors, transforming the devices into systems on a chip (SoC).

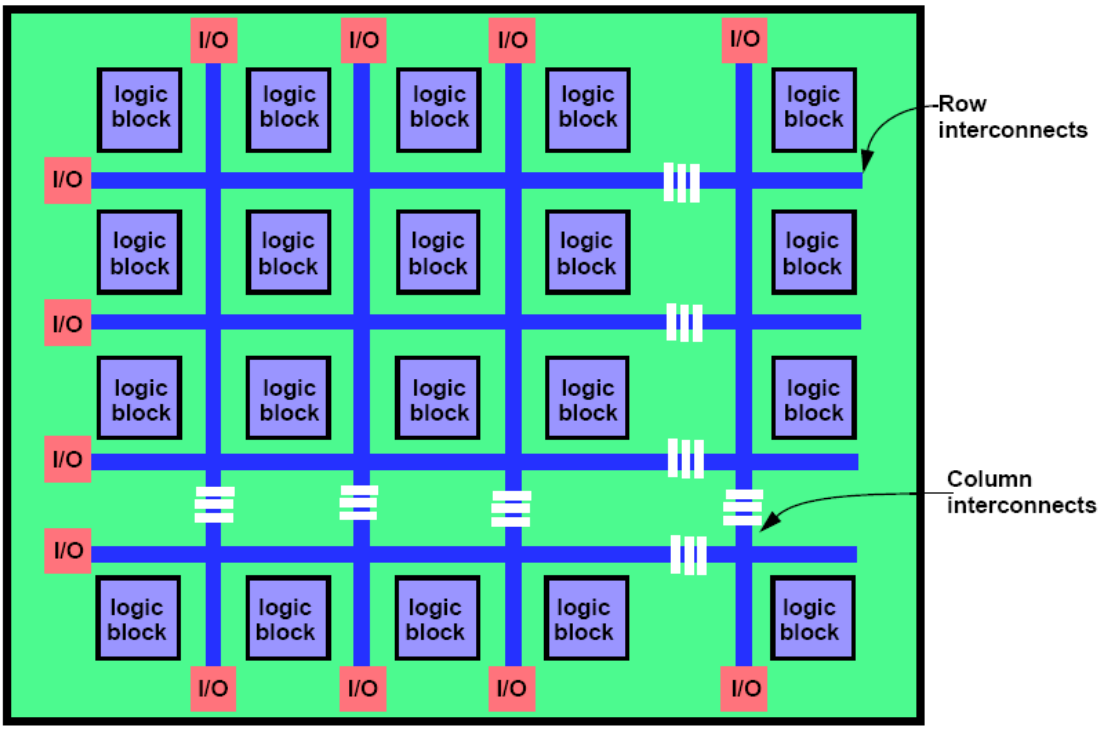



**Figure 1.2** FPGA Architecture

Compared to ASICs or ASSPs, FPGAs offer many design advantages, including:

- Rapid prototyping
- Shorter time to market
- The ability to re-program in the field for debugging
- Lower NRE costs
- Long product life cycle to mitigate obsolescence risk

## Applications

Applications of FPGAs include digital signal processing, software-defined radio, aerospace and defense systems, ASIC prototyping, medical imaging, computer vision, speech recognition, cryptography, bioinformatics, computer hardware emulation, radio astronomy, metal detection and a growing range of other areas.

FPGAs originally began as competitors to CPLDs and competed in a similar space, that of glue logic for PCBs. As their size, capabilities, and speed increased, they

began to take over larger and larger functions to the state where some are now marketed as full systems on chips (SoC). Particularly with the introduction of dedicated multipliers into FPGA architectures in the late 1990s, applications which had traditionally been the sole reserve of DSPs began to incorporate FPGAs instead.

Traditionally, FPGAs have been reserved for specific vertical applications where the volume of production is small. For these low-volume applications, the premium that companies pay in hardware costs per unit for a programmable chip is more affordable than the development resources spent on creating an ASIC for a low-volume application. Today, new cost and performance dynamics have broadened the range of viable applications.

## Short History of FPGAs

The FPGA concept emerged in 1985 with the XC2064TM FPGA family from Xilinx. At the same time, a company called Altera were also developing a programmable device, later to become EP1200 device which was the first high-density programmable logic device. Altera's technology was manufactured using 3-µm CMOS erasable programmable read-only-memory (EPROM) technology and required ultraviolet light to erase the programming whereas Xilinx's technology was based on conventional static RAM technology and required an EPROM to sore the programming. The co-founder of Xilinx, Ross Freeman argued that with continuously improving silicon technology, transistors were going to increasingly get cheaper and could be used to offer programmability. This is was the start of an FPGA market which was then populated by quite a number of vendors, including Xilinx, Altera, Actel, Lattice, Crosspoint, Algotronix, Prizm, Plessey, Toshiba, Motorola, and IBM. The market has now grown considerably and Gartner Dataquest indicated a market size growth to 4.5 billion in 2006, 5.2 billion in 2007 and 6.3 billion in 2008. There have been many changes in the market, including a severe rationalization of technologies with many vendors such as Crosspoint, Algotronix, Prizm, Plessey, Toshiba, Motorola, and IBM disappearing from the market and a reduction in the number of FPGA families as

well as the emergence of SRAM technology as the dominant technology largely due to cost. The market is now dominated by Xilinx and Altera and more importantly, the FPGA has grown from being a simple glue logic component to representing a complete System on Programmable Chip (SoPC) comprising on-board physical processors, soft processor, dedicated DSP hardware, memory and high-speed I/O. In the 1990s, ASIC was still seen for the key mass market areas where really high performance and energy considerations were seen as key drivers such as mobile communications. Thus graphs comparing performance metrics for FPGA, ASIC and processor were generated and used by vendors to indicate design choices.

The FPGA evolution is summarized in Table 1.1. It indicates three different eras of evolution of the FPGA. The age of *invention* where FPGAs started to emerge and were being used as system components. The age of *expansion* is where the FPGA started to approach the problem size and thus design complexity was key. The final evolution stage is described as the period of *accumulation* where FPGA started to incorporate processors and high-speed interconnection.

**Table 1.1** Three ages of FPGAs

| Period | Age | Comments |
|---|---|---|
| 1984–1991 | Invention | Technology is limited, FPGAs are much smaller than the application problem size<br>Design automation is secondary<br>Architecture efficiency is key |
| 1992–1999 | Expansion | FPGA size approaches the problem size<br>Ease-of-design becomes critical |
| 2000–2007 | Accumulation | FPGAs are larger than the typical problem size<br>Logic capacity limited by I/O bandwidth |

## Challenges of FPGAs

The emergence of the FPGA as a DSP platform was accelerated by the application of distributed arithmetic (DA) techniques (Goslin 1995, Meyer-Baese 2001). DA allowed efficient FPGA implementations to be realized using the LUT-

based/adder constructs of FPGA blocks and allowed considerable performance gains to be gleaned for some DSP transforms such as fixed coefficient filtering and transform functions such as Fast Fourier Transform (FFT). Whilst these techniques demonstrated that FPGAs could produce highly effective solutions for DSP applications, the concept of squeezing the last aspect of performance out of the FPGA hardware and more importantly, spending several person months for the creation of such innovative designs, meant that there was a growing gap in the scope offered by current FPGA technology and the designer's ability to develop efficient solutions using modern tools. This was similar to the 'design productivity gap' (ITRS 1999) identified in the ASIC industry where it was viewed that ASIC design capability was only growing at 25% whereas Moore's law growth was 60%. This is proved by even more recent data during the 2007 ITRS roadmap .

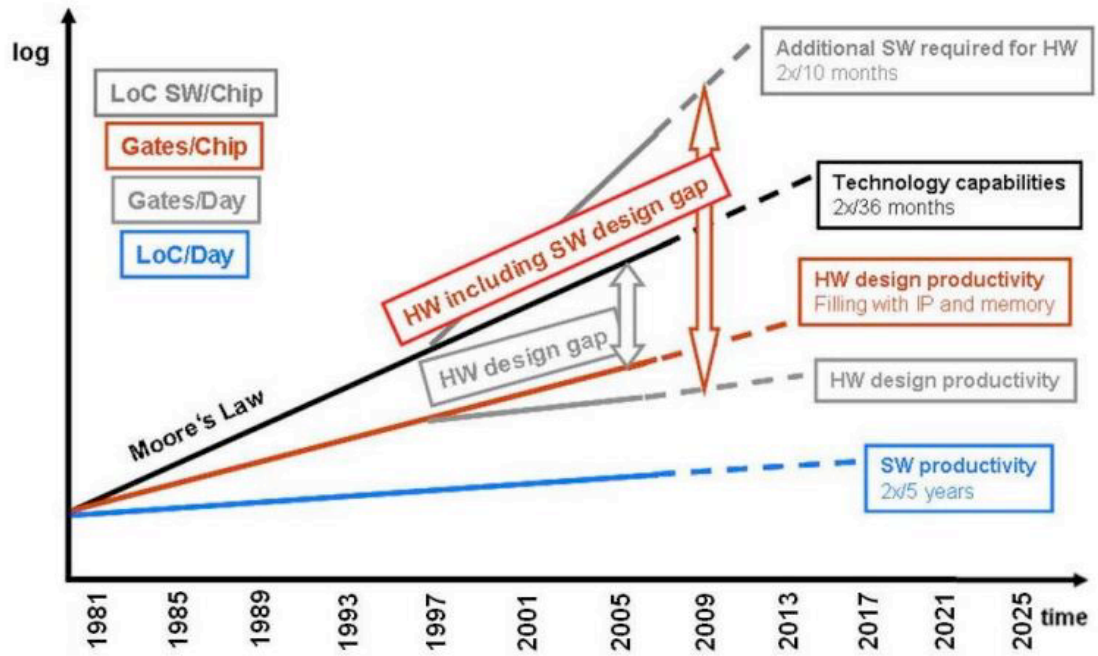



**Figure 1.3** The design productivity gap (ITRS 2007)

The problem is not as severe in FPGA implementation, because sub-micrometer design issues are missing. However, a number of key issues exist and include:

- *Design languages.* Currently hardware description languages such as VHDL and Verilog and their respective synthesis flows are well established. However, users are now looking at FPGAs with the recent increase in complexity resulting in the integration of both fixed and programmable microprocessors cores as a complete system, and looking for design representations that more clearly represent system description. Therefore, there is an increased EDA focus on using C as a design language.

- *Understanding how to map DSP functionality into FPGA.* Some of the aspects are relatively basic in this area, such as multiplications, additions and delays being mapped onto on-board multipliers, adder and registers and RAM components respectively. However, the understanding of floating-point versus fixed-point, word length optimization, algorithmic transformation cost functions for FPGA and impact of routing delay are issues that must be considered at a system level and can be much harder to deal with at this level.

- *Development and use of IP cores.* With the absence of quick and reliable solutions to the design language and synthesis issues, the IP market in SoC implementation has emerged to fill the gap and allow rapid prototyping of hardware. Soft cores are particularly attractive as design functionality can be captured using HDLs and efficiently translated into the FPGA technology of choice in a highly efficient manner by conventional synthesis tools. In addition, processor cores have been developed which allow dedicated functionality to be added. The attraction of these approaches are that they allow application specific functionality

to be quickly created as the platform is largely fixed.

- *Design flow.* Most of the design flow capability is based around developing FPGA functionality from some form of higher-level description, mostly for

  complex functions. The reality now is that FPGA technology is evolving at such a rate that systems comprising FPGAs and processors are starting to emerge as a SoC platform or indeed, FPGAs as a single SoC platform as they have on-board hard and soft processors, high-speed communications and programmable resource, and this can be viewed as a complete system. Conventionally, software flows have been more advanced for processors and even multiple processors as the architecture is fixed. Whilst tools have developed for hardware platforms such as FPGAs, there is a definite need for software for flows for heterogeneous platforms, i.e. those that involve both processors and FPGAs.

## 1.2 Introduction to Computer Vision Algorithms

Computer vision is formed by a field that includes methods for acquiring, processing, analyzing, understanding images and generally high-dimensional data from the real world, in order to produce numerical or symbolic information, *e.g.*, in the forms of decisions. An issue that has been raised in the process of the development of this field has been to duplicate the abilities of human vision by electronically perceiving and understanding an image. This image understanding can be seen as the disentangling of symbolic information from image data using models constructed with the aid of geometry, physics, statistics, and learning theory.

Nowadays, there are many implemented computer vision algorithms and obviously computer vision has endless applications. Some of them are shown at the diagram below.

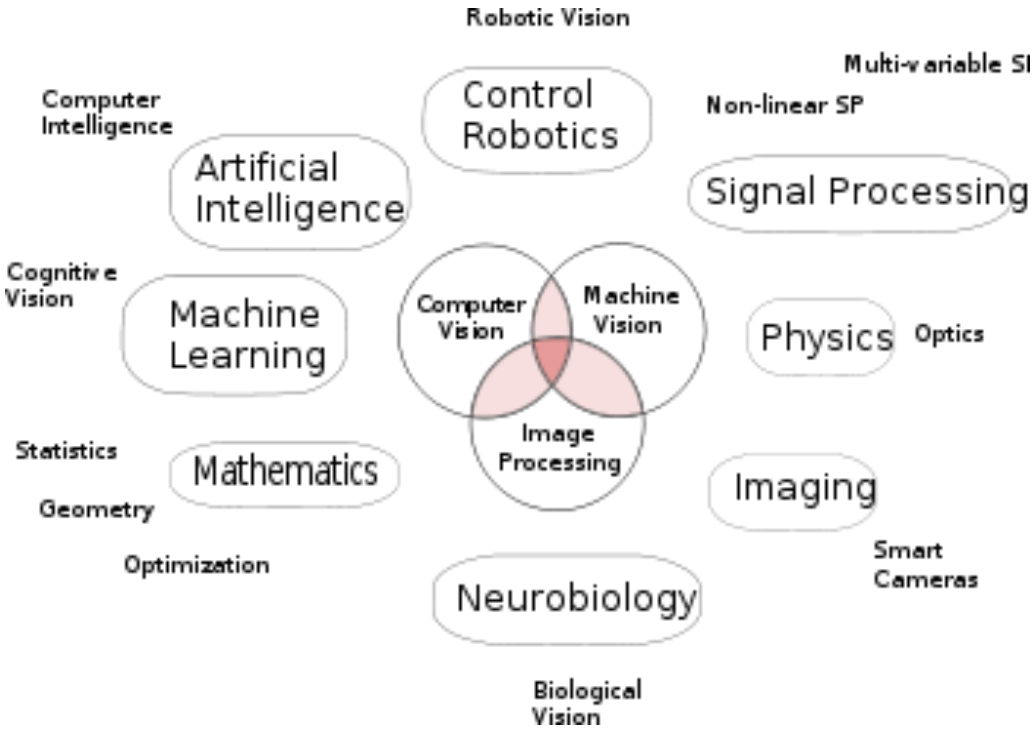



**Figure 1.4** Applications of Computer Vision

*Vision Localization algorithms.*

*Computer Vision and FPGA.*

Computer vision algorithms are natural candidates for high performance computing due to their inherent parallelism and intense computational demands. For example, a simple 3 x 3 convolution on a 512 x 512 gray scale image at 30 frames per second requires 67.5 million multiplications and 60 million additions to be performed in one second. Computer vision tasks can be classified into three categories based on their computational complexity and communication complexity: low-level, intermediate-level and high-level. Special-purpose hardware provides better performance compared to a general-purpose hardware for all the three levels of vision tasks. With recent advances in very large scale integration (VLSI) technology, an application specific integrated circuit (ASIC) can provide the best performance in terms of total execution time. However, long design cycle time, high development cost and in flexibility of a

dedicated hardware deter design of ASICs. In contrast, field programmable gate arrays (FPGAs) support lower design verification time and easier design adaptability at a lower cost. Hence, FPGAs with an array of reconfigurable logic blocks can be very useful compute elements. FPGA-based custom computing machines are playing a major role in realizing high performance application accelerators. Three computer vision algorithms have been investigated for mapping onto custom computing machines:

(i) template matching (convolution) a low level vision operation
(ii) texture-based segmentation { an intermediate-level operation, and
(iii) point pattern matching { a high level vision algorithm.

The advantages demonstrated through these implementations are as follows. First, custom computing machines are suitable for all the three levels of computer vision algorithms. Second, custom computing machines can map all stages of a vision system easily. This is unlike typical hardware platforms where a separate subsystem is dedicated to a specific step of the vision algorithm. Third, custom computing approach can run a vision application at a high speed, often very close to the speed of special-purpose hardware.

# Chapter 2

# Behind the Project

This chapter refers to the motivation behind this project and the applications of Linear Time Selection Algorithm. We also give a brief description of previous related work.

## 2.1 Applications

In statistics, the *k-th* **order statistic** of a statistical sample is equal to its *kth*-smallest value. Along with rank statistics, order statistics are among the most fundamental tools in non-parametric statistics and inference.

The problem of computing the *kth* smallest (or largest) element of a list is called the selection problem. In computer science, a **selection algorithm** is an algorithm for finding the *k*th smallest number in a list (such a number is called the *k*th *order statistic*). Although this problem is difficult for very large lists, sophisticated selection algorithms have been created that can solve this problem in time proportional to the number of elements in the list, even if the list is totally unordered.

Selection is a sub-problem of more complex problems like the nearest neighbor problem and shortest path problems.

Some examples of order statistics follows:

- Sample maximum and minimum
- Quantile
- Percentile
- Decile
- Quartile
- Median

Sample maximum and minimum

Applications:

- Summary statistics

Firstly, the sample maximum and minimum are basic summary statistics, showing the most extreme observations, and are used in the five-number summary and seven-number summary and the associated box plot.

- Prediction interval

The sample maximum and minimum provide a non-parametric prediction interval: in a sample set from a population, or more generally an exchangeable sequence of random variables, each sample is equally likely to be the maximum or minimum.

- Estimation

Due to their sensitivity to outliers, the sample extreme cannot reliably be used as estimators unless data is clean – robust alternatives include the first and last deciles.

- Normality testing
- Extreme value theory

Sample extreme play two main roles in extreme value theory. Firstly, they give a lower bound on extreme events – events can be at least this extreme, and for this size sample. Secondly, they can sometimes be used in estimators of probability of more extreme events.

Quantile

Quantiles are points taken at regular intervals from the cumulative distribution function (CDF) of a random variable. Dividing ordered data into essentially equal-sized data subsets is the motivation for -quantiles; the quantiles are the data values marking the boundaries between consecutive subsets. Put another way, the -quantile for a random variable is the value such that the probability that the random variable will be less than is at most and the probability that the random variable will be more than is at most . There are of the -quantiles, one for each integer satisfying .

Percentile

In statistics, a percentile (or centile) is the value of a variable below which a certain percent of observations fall. For example, the 20th percentile is the value (or score) below which 20 percent of the observations may be found. The term percentile and the related term percentile rank are often used in the reporting of scores from norm-referenced tests.

Quartile

In descriptive statistics, the quartiles of a set of values are the three points that divide the data set into four equal groups, each representing a fourth of the population being sampled. A quartile is a type of quantile.

In epidemiology, sociology and finance, the quartiles of a population are the four subpopulations defined by classifying individuals according to whether the value concerned falls into one of the four ranges defined by the three values discussed above.

Median

In statistics and probability theory, median is described as the numerical value separating the higher half of a sample, a population, or a probability distribution, from the lower half.

In the context of image processing there is a type of noise, known as the salt and pepper noise, when each pixel independently becomes black (with some small probability) or white (with some small probability), and is unchanged otherwise (with the probability close to 1). An image constructed of median values of neighborhoods (like 3×3 square) can effectively reduce noise in this case.

*Median Filter:*

Real-time signal processing and image processing applications employ filtering in order to process and to manipulate the signals, or to remove noise from data. Median filter is a non-linear filter used for removing impulsive noise from data. This chapter provides a description of the median filter and median filtering techniques implemented on the hardware devices.

There are two types of filters: linear filters and non-linear filters. The median filter is a non-linear filter; it is a special case of rank order filters, whose rank is half the length of the sequence. In image processing applications, median filter is used to remove impulsive noise from images while preserving the edges.

One of the disadvantages of linear filters, such as the moving average filter when used to denoise the data, is that they not only smooth the noise, but also smooth the sudden and sharp transitions that were present in the original data, such as edges in images. Moreover, they are not as efficient as the median filters in removing certain types of noise, such as impulsive noise. Although median filters do not blur the edges as much as the linear filters do, as they still possess smoothing characteristics, such as the size of the filter increases, there may be significant image blurring. Impulsive noise can be classified into two types:

(1) salt and pepper noise

(2) random valued noise.

Salt and pepper noise pixels take only two values, either the minimum or

the maximum possible value, for example, in a gray scale image, salt and pepper noise pixels will be either 0 or 255.

However, random valued noise pixels take any random value, which is more difficult to remove than the salt and pepper noise. If p and q are the probabilities of occurrences of 255 and 0, where $0 \leq p, q \leq 1$ and p and q can be equal or different, a pixel may be replaced by 255 with a probability p and by 0 with a probability q.

The median of a given sequence is given by sorting the sequence and choosing the middle value from the sorted sequence. If there are odd number of elements in a sequence, then the median is the middle element in the sorted list. If there are even number of elements then the median is given by the arithmetic mean of the two middle elements in the sorted sequence.

In image processing, the 2D filtering operation is performed by sliding the window along all the rows and columns of the image until all the pixels are covered by the window. The filtering is done by sliding the window across the image, sorting all the pixels in the window, which consists of a center pixel and the neighborhood pixels, and then replacing the central pixel with the median intensity of the window. Since salt and pepper noise pixels take only either the maximum or the minimum possible value and the result of a median filter excludes the extreme value, median filtering provides a good reduction of the salt and pepper noise.

## 2.2 Related Work

In this section, we provide a short description of the academic related work to date.

As we have mentioned above we implement on FPGA a Linear time Selection Algorithm and we focus especially on the case of median filter as it is commonly used for image filtering, which plays an important role in image preprocessing.

Linear time Selection Algorithm is a solution to the Selection problem, which is to find the *kth* smallest element in a sequence of n numbers in arbitrary order. This is a typical problem in algorithm design and analysis and solutions based on divide and conquer strategy[31] and sorting algorithms [32] have been proposed.

The existing hardware designs of median filters can be classified into two main groups [1, 28, 30]: word-based and bit-based. Word-based (or bit-parallel) architectures process the bits of the input samples in parallel, but the samples are usually processed sequentially. On the contrary, bit-based filters process input samples in bit-serial but the samples included in the window are processed in parallel.

Next, median filters can be categorized as non-recursive and recursive. In the non-recursive filters, windows contain values of the input samples only, while in the recursive filters, the window contains the most recent median values as well as the input values [16]. Because the recursive filters reuse the hardware in iterative manner, they usually more compact but slower than non-recursive versions.

Furthermore, median filters can be divided into three categories: array architectures [2-12] sorting-network architectures [13-18] and stack-based architectures [20-26]. In array architectures each element of the window is associated with the rank, and with each window shift, the ranks are updated. Sorting network based architectures first range the samples and then select the sample of corresponding rank. Stack-based architectures translate filtering into binary domain through the use of threshold logic [21, 22, 24], majority elements

[23], hamming comparators [26], etc. Generally, array architectures consist of N processors but have a longer sampling period due to word-level comparison. Sorting network based architectures, in turn, are inherently pipelined and so have a higher throughput. However, they require a large number of compare-swap units. Stack based architectures provide reasonable performance with a limited modularity. In bit-parallel implementations of stack architectures, the number of processing levels grows exponentially, while bit-serial implementations are typically slow.

Experimental evaluation and comparisons between software and hardware executions show that high throughput FPGA circuits easily outperform even the most advanced DSP processors [29].

# Chapter 3

# Hardware and Software

## 3.1 The Platform

The device we used for the FPGA implementation of linear time selection algorithm is the *Virtex-6 xc6vlx240t*. For the integration with the driver we have also used *Virtex-5 xc5vlx50t* and *Spartan 3E.*

An overview[30] of the *Virtex-6* family devices follows:

*General Description*

The Virtex®-6 family provides the newest, most advanced features in the FPGA market. Virtex-6 FPGAs are the programmable silicon foundation for Targeted Design Platforms that deliver integrated software and hardware components to enable designers to focus on innovation as soon as their development cycle begins. Using the third-generation ASMBL™ (Advanced Silicon Modular Block) column based architecture, the Virtex-6 family contains multiple distinct sub-families. This overview covers the devices in the LXT, SXT, and HXT sub-families. Each sub-family contains a different ratio of features to most efficiently address the needs of a wide variety of advanced logic designs. In addition to the high-performance logic fabric, Virtex-6 FPGAs contain many built-in system-level blocks. These features allow logic designers to build the highest levels of performance and functionality into their FPGA-based systems. Built on a 40 nm state-of-the art copper process technology, Virtex-6 FPGAs are a programmable alternative to custom ASIC technology. Virtex-6 FPGAs offer the best solution for addressing the needs of high-performance logic designers, high-performance DSP designers, and high-performance embedded systems designers with unprecedented logic, DSP, connectivity, and soft microprocessor capabilities.

*Configuration*

Virtex-6 FPGAs store their customized configuration in SRAM-type internal latches. The number of configuration bits is between 26 Mb and 177 Mb, depending on device size but independent of the specific user-design implementation, unless compression mode is used. The configuration storage is volatile and must be reloaded whenever the FPGA is powered up. This storage can also be reloaded at any time by pulling the PROGRAM_B pin Low. Several methods and data formats for loading configuration are available, determined by the three mode pins. Bit-serial configurations can be either master serial mode where the FPGA generates the configuration clock (CCLK) signal, or slave serial mode where the external configuration data source also clocks the FPGA. For byte- and word-wide configurations, master SelectMAP mode generates the CCLK signal while slave SelectMAP mode receives the CCLK signal for the 8-, 16-, or 32-bit-wide transfer. Alternatively, serial-peripheral interface (SPI) and byte-peripheral interface (BPI) modes are used with industry-standard flash memories and are clocked by the CCLK output of the FPGA. JTAG mode uses boundary-scan protocols to load bit-serial configuration data. The bitstream configuration information is generated by the ISE® software using a program called BitGen. The configuration process typically executes the following sequence:

• Detects power-up (power-on reset) or PROGRAM_B when Low.

• Clears the whole configuration memory.

• Samples the mode pins to determine the configuration mode: master or slave, bit-serial or parallel, or bus width.

• Loads the configuration data starting with the bus-width detection pattern followed by a synchronization word, checks for the proper device code, and ends with a cyclic redundancy check (CRC) of the complete bitstream.

• Start-up executes a user-defined sequence of events: releasing the internal reset (or preset) of flip-flops, optionally waiting for the phase-locked loops (PLLs) to lock and/or the DCI to match, activating the output drivers, and

transitions the DONE pin High.

*Dynamic Reconfiguration Port*

The dynamic reconfiguration port (DRP) gives the system designer easy access to configuration bits and status registers for three block types: 32 locations for each clock tile, 128 locations for the System Monitor, and 128 locations for each serial GTX or GTH transceiver. The DRP behaves like memory-mapped registers, and can access and modify block-specific configuration bits as well as status and control registers.

*Encryption, Readback, and Partial Reconfiguration*

As a special option, the bitstream can be AES-encrypted to prevent unauthorized copying of the design. The Virtex-6 FPGA performs the decryption using the internally stored 256-bit key that can use battery backup or alternative non-volatile storage. Most configuration data can be read back without affecting the system's operation. Typically, configuration is an all-ornothing operation, but the Virtex-6 FPGA also supports partial reconfiguration. When applicable in certain designs, partial reconfiguration can greatly improve the versatility of the FPGA. It is even possible to reconfigure a portion of the FPGA while the rest of the logic remains active i.e., active partial reconfiguration.

*CLBs, Slices, and LUTs*

The look-up tables (LUTs) in Virtex-6 FPGAs can be configured as either one 6-input LUT (64-bit ROMs) with one output, or as two 5-input LUTs (32-bit ROMs) with separate outputs but common addresses or logic inputs. Each LUT output can optionally be registered in a flip-flop. Four such LUTs and their eight flip-flops as well as multiplexers and arithmetic carry logic form a slice, and two slices form a configurable logic block (CLB). Four flip-flops per slice (one per LUT) can optionally be configured as latches. In that case, the remaining four flip-flops in that slice must remain unused. Between 25–50% of all slices can also use their LUTs as distributed 64-bit RAM or as 32-bit shift registers (SRL32) or as two SRL16s. Modern synthesis tools take advantage of these highly efficient logic, arithmetic, and memory features. Expert designers can also instantiate

them.

*Clock Management*

Each Virtex-6 FPGA has up to nine clock management tiles (CMTs), each consisting of two mixed-mode clock managers (MMCMs), which are PLL based.

*Phase-Locked Loop*

The MMCM can serve as a frequency synthesizer for a wider range of frequencies and as a jitter filter for incoming clocks. The heart of the MMCM is a voltage-controlled oscillator (VCO) with a frequency from 600 MHz up to 1600 MHz, spanning more than one octave. There are three sets of programmable frequency dividers (D, M, and O). The pre-divider D (programmable by configuration) reduces the input frequency and feeds one input of the traditional PLL phase/frequency comparator. The feedback divider (programmable by configuration) acts as a multiplier because it divides the VCO output frequency before feeding the other input of the phase comparator. D and M must be chosen appropriately to keep the VCO within its specified frequency range. The VCO has eight equally-spaced output phases (0°, 45°, 90°, 135°, 180°, 225°, 270°, and 315°). Each can be selected to drive one of the seven output dividers, O0 to O6 (each programmable by configuration to divide by any integer from 1 to 128).

*MMCM Programmable Features*

The MMCM has three input-jitter filter options: low bandwidth, high bandwidth, or optimized mode. Low-bandwidth mode has the best jitter attenuation but not the smallest phase offset. High-bandwidth mode has the best phase offset, but not the best jitter attenuation. Optimized mode allows the tools to find the best setting. The MMCM can have a fractional counter in either the feedback path (acting as a multiplier) or in one output path. Fractional counters allow non-integer increments of 1/8 and can thus increase frequency synthesis capabilities by a factor of 8. The MMCM can also provide fixed or dynamic phase shift in small increments that depend on the VCO frequency. At 600 MHz the phase-shift timing increment is 30 ps; at 1600 MHz, it is 11.5 ps.

*Clock Distribution*

Each Virtex-6 FPGA provides five different types of clock lines (BUFG, BUFR, BUFIO, BUFH, and the high-performance clock) to address the different clocking requirements of high fanout, short propagation delay, and extremely low skew.

*Global Clock Lines*

In each Virtex-6 FPGA, 32 global-clock lines have the highest fanout and can reach every flip-flop clock, clock enable, set/reset, as well as many logic inputs. There are 12 global clock lines within any region. Global clock lines can be driven by global clock buffers, which can also perform glitchless clock multiplexing and the clock enable function. Global clocks are often driven from the CMT, which can completely eliminate the basic clock distribution delay.

*Regional Clocks*

Regional clocks can drive all clock destinations in their region as well as the region above and below. A region is defined as any area that is 40 I/O and 40 CLB high and half the chip wide. Virtex-6 FPGAs have between 6 and 18 regions. There are 6 regional clock tracks in every region. Each regional clock buffer can be driven from either of four clock-capable input pins, and its frequency can optionally be divided by any integer from 1 to 8.

*I/O Clocks*

I/O clocks are especially fast and serve only I/O logic and serializer/deserializer (SerDes) circuits, as described in the I/O Logic section. Virtex-6 devices have a high-performance direct connection from the MMCM to the I/O directly for low-jitter, high-performance interfaces.

*Block RAM*

Every Virtex-6 FPGA has between 156 and 1064 dual-port block RAMs, each storing 36 Kbits. Each block RAM has two completely independent ports that share nothing but the stored data.

*Synchronous Operation*

Each memory access, read and write, is controlled by the clock. All inputs, data, address, clock enables, and write enables are registered. Nothing happens without a clock. The input address is always clocked, retaining data until the next operation. An optional output data pipeline register allows higher clock rates at the cost of an extra cycle of latency. During a write operation, the data output can reflect either the previously stored data, the newly written data, or remain unchanged.

*Programmable Data Width*

• Each port can be configured as 32K x 1, 16K x 2, 8K x 4, 4K x9 (or 8), 2K x 18 (or 16), 1K x 36 (or 32), or 512 x 72 (or 64). The two ports can have different aspect ratios, without any constraints.

• Each block RAM can be divided into two completely independent 18 Kb block RAMs that can each be configured to any aspect ratio from 16K x 1 to 512 x 36. Everything described previously for the full 36 Kb block RAM also applies to each of the smaller 18 Kb block RAMs.

• In 18 Kb block RAMs, only simple dual-port mode can provide data width of >36 bits. In this mode, one port is dedicated to read and the other port is dedicated to write operation. In SDP mode one side (read or write) can be variable while the other is fixed to 32/36 or 64/72. There is no read output during write. The dual-port 36 Kb RAM both sides can be of variable width.

• Two adjacent 36 Kb block RAMs can be configured as one cascaded 64K Å~ 1 dual-port RAM without any additional logic.

*Error Detection and Correction*

Each 64 bit-wide block RAM can generate, store, and utilize eight additional Hamming-code bits, and perform single-bit error correction and double-bit error detection (ECC) during the read process. The ECC logic can also be used when writing to, or reading from external 64/72-wide memories. This works in simple dual-port mode and does not support read-during-write.

*FIFO Controller*

The built-in FIFO controller for single-clock (synchronous) or dual-clock (asynchronous or multirate) operation increments the internal addresses and provides four handshaking flags: full, empty, almost full, and almost empty. The almost full and almost empty flags are freely programmable. Similar to the block RAM, the FIFO width and depth are programmable, but the write and read ports always have identical width. First-word fall-through mode presents the first-written word on the data output even before the first read operation. After the first word has been read, there is no difference between this mode and the standard mode.

*Digital Signal Processing—DSP48E1 Slice*

DSP applications use many binary multipliers and accumulators, best implemented in dedicated DSP slices. All Virtex-6 FPGAs have many dedicated, full-custom, low-power DSP slices combining high speed with small size, while retaining system design flexibility. Each DSP48E1 slice fundamentally consists of a dedicated 25 x 18 bit two's complement multiplier and a 48-bit accumulator, both capable of operating at 600 MHz. The multiplier can be dynamically bypassed, and two 48-bit inputs can feed a single-instruction-multiple-data (SIMD) arithmetic unit (dual 24-bit add/subtract/accumulate or quad 12-bit add/subtract/accumulate), or a logic unit that can generate any one of 10 different logic functions of the two operands. The DSP48E1 includes an additional pre-adder, typically used in symmetrical filters. This new pre-adder improves performance in densely packed designs and reduces the logic slice count by up to 50%. The DSP48E1 slice provides extensive pipelining and extension capabilities that enhance speed and efficiency of many applications, even beyond digital signal processing, such as wide dynamic bus shifters, memory address generators, wide bus multiplexers, and memory-mapped I/O register files. The accumulator can also be used as a synchronous up/down counter. The multiplier can perform logic functions (AND, OR) and barrel shifting.

*Input/Output*

The number of I/O pins varies from 240 to 1200 depending on device and package size. Each I/O pin is configurable and can comply with a large number of standards, using up to 2.5V. The Virtex-6 FPGA SelectIO Resources User Guide describes the I/O compatibilities of the various I/O options. With the exception of supply pins and a few dedicated configuration pins, all other package pins have the same I/O capabilities, constrained only by certain banking rules.

All I/O pins are organized in banks, with 40 pins per bank. Each bank has one common VCCO output supply-voltage pin, which also powers certain input buffers. Some single-ended input buffers require an externally applied reference voltage (VREF). There are two VREF pins per bank (except configuration bank 0). A single bank can have only one VREF voltage value.

*I/O Electrical Characteristics*

Single-ended outputs use a conventional CMOS push/pull output structure driving High towards VCCO or Low towards ground, and can be put into high-Z state. The system designer can specify the slew rate and the output strength. The input is always active but is usually ignored while the output is active. Each pin can optionally have a weak pull-up or a weak pulldown resistor. Any signal pin pair can be configured as differential input pair or output pair. Differential input pin pairs can optionally be terminated with a 100Ω internal resistor. All Virtex-6 devices support differential standards beyond LVDS: HT, RSDS, BLVDS, differential SSTL, and differential HSTL.

*Digitally Controlled Impedance*

Digitally controlled impedance (DCI) can control the output drive impedance (series termination) or can provide parallel termination of input signals to VCCO, or split (Thevenin) termination to VCCO/2. DCI uses two pins per bank as reference pins, but one such pair can also control multiple banks. VRN must be resistively pulled to VCCO, while VRP must be resistively connected to ground. The resistor must be either 1x or 2x the characteristic trace impedance, typically close to 50Ω.

*I/O Logic*

*Input and Output Delay*

This section describes the available logic resources connected to the I/O interfaces. All inputs and outputs can be configured as either combinatorial or registered. Double data rate (DDR) is supported by all inputs and outputs. Any input or output can be individually delayed by up to 32 increments of ~78 ps each. This is implemented as IODELAY. The number of delay steps can be set by configuration and can also be incremented or decremented while in use. For using either IODELAY, the system designer must instantiate the IODELAY control block and clock it with a frequency close to 200 MHz. Each 32-tap total IODELAY is controlled by that frequency, thus unaffected by temperature, supply voltage, and processing variations.

*ISERDES and OSERDES*

Many applications combine high-speed bit-serial I/O with slower parallel operation inside the device. This requires a serializer and deserializer (SerDes) inside the I/O structure. Each input has access to its own deserializer (serial-to-parallel converter) with programmable parallel width of 2, 3, 4, 5, 6, 7, 8, or 10 bits. Each output has access to its own serializer (parallel-to-serial converter) with programmable parallel width of up to 8 bits wide for single data rate (SDR), or up to 10 bits wide for double data rate (DDR).

*System Monitor*

Every Virtex-6 FPGA contains a System Monitor circuit providing thermal and power supply status information. Sensor outputs are digitized by a 10-bit 200kSPS analog-to-digital converter (ADC). This fully tested and specified ADC can also be used to digitize up to 17 external analog input channels. The System Monitor ADC utilizes an on-chip reference circuit thereby eliminating the need for any external active components. On-chip temperature and power supplies are monitored with a measurement accuracy of ±4°C and ±1% respectively. By default the System Monitor continuously digitizes the output of all on-chip sensors. The most recent measurement results together with maximum and

minimum readings are stored in dedicated registers for access at any time through the DRP or JTAG interfaces. User defined alarm thresholds can automatically indicate over temperature events and unacceptable power supply variation. A specified limit (for example: 125°C) can be used to initiate an automatic power down. The System Monitor does not require explicit instantiation in a design. Once the appropriate power supply connections are made, measurement data can be accessed at any time, even pre-configuration or during power down, through the JTAG test access port (TAP).

*Low-Power Gigabit Transceivers*

Ultra-fast serial data transmission between ICs, over the backplane, or over longer distances is becoming increasingly popular and important. It requires specialized dedicated on-chip circuitry and differential I/O capable of coping with the signal integrity issues at these high data rates. All but one Virtex-6 device has between 8 to 72 gigabit transceiver circuits. Each GTX transceiver is a combined transmitter and receiver capable of operating at a data rate between 480 Mb/s and 6.6 Gb/s. Lower data rates can be achieved using FPGA logic-based oversampling. Each GTH transceiver is a combined transmitter and receiver capable of operating at a rate between 2.488 Gb/s and 11.18 Gb/s. The GTX transmitter and receiver are independent circuits that use separate PLLs to multiply the reference frequency input by certain programmable numbers between 4 and 25, to become the bit-serial data clock. The GTH transceiver is a purpose-built design for 10 Gb/s rates and shares a single high-performance PLL between four transmitter and receiver circuits. Each GTX and GTH transceiver has a large number of user-definable features and parameters. All of these can be defined during device configuration, and many can also be modified during operation.

*Transmitter*

The GTX transmitter is fundamentally a parallel-to-serial converter with a conversion ratio of 8, 10, 16, 20, 32, or 40. The GTH transmitter offers bit widths of 16, 20, 32, 40, 64, or 80 to allow additional timing margin for high-performance designs. These transmitter outputs drive the PC board with a

single-channel differential current-mode logic (CML) output signal. TXOUTCLK is the appropriately divided serial data clock and can be used directly to register the parallel data coming from the internal logic. The incoming parallel data is fed through a small FIFO and can optionally be modified with the 8B/10B, 64B/66B, or the 64B/67B (GTX only) algorithm to guarantee a sufficient number of transitions. The bit-serial output signal drives two package pins with complementary CML signals. This output signal pair has programmable signal swing as well as programmable pre-emphasis to compensate for PC board losses and other interconnect characteristics.

*Receiver*

The receiver is fundamentally a serial-to-parallel converter, changing the incoming bit serial differential signal into a parallel stream of words, each 8, 10, 16, 20, 32, or 40 bits wide. The GTH transceiver offers 16, 20, 32, 40, 64, and 80 bit widths to allow greater timing margin. The receiver takes the incoming differential data stream, feeds it through a programmable equalizer (to compensate for PC board and other interconnect characteristics), and uses the FREF input to initiate clock recognition. There is no need for a separate clock line. The data pattern uses non-return-to-zero (NRZ) encoding and optionally guarantees sufficient data transitions by using the selected encoding scheme. Parallel data is then transferred into the FPGA logic using the RXUSRCLK clock. The serial-to-parallel conversion ratio for GTX transceivers can be 8, 10, 16, 20, 32, or 40. The serial-to-parallel conversion ratio for GTH transceivers can be 16, 20, 32, 40, 64, or 80 for GTH.

*Out-of-Band Signaling*

The GTX transceivers provide Out-of-Band (OOB) signaling, often used to send low-speed signals from the transmitter to the receiver, while high-speed serial data transmission is not active, typically when the link is in a power-down state or has not been initialized. This benefits PCI Express and SATA/SAS applications.

*Integrated Interface Blocks for PCI Express Designs*

The PCI Express standard is a packet-based, point-to-point serial interface standard. The differential signal transmission uses an embedded clock, which eliminates the clock-to-data skew problems of traditional wide parallel buses. The PCI Express Base Specification Revision 2.0 is backwards compatible with Revision 1.1 and defines a configurable raw data rate of 2.5 Gb/s, or 5.0 Gb/s per lane in each direction. To scale bandwidth, the specification allows multiple lanes to be joined to form a larger link between PCI Express devices. All Virtex-6 devices (except the XC6VLX760) include at least one integrated interface block for PCI Express technology that can be configured as an Endpoint or Root Port, compliant to the PCI Express Base Specification Revision 2.0. The Root Port can be used to build the basis for a compatible Root Complex, to allow custom FPGA-FPGA communication via the PCI Express protocol, and to attach ASSP Endpoint devices such as Fibre Channel HBAs to the FPGA. This block is highly configurable to system design requirements and can operate 1, 2, 4, or 8 lanes at the 2.5 Gb/s data rate and the 5.0 Gb/s data rate. For high-performance applications, advanced buffering techniques of the block offer a flexible maximum payload size of up to 1024 bytes. The integrated block interfaces to the GTX transceivers for serial connectivity, and to block RAMs for data buffering. Combined, these elements implement the Physical Layer, Data Link Layer, and Transaction Layer of the PCI Express protocol. Xilinx provides a light-weight, configurable, easy-to-use LogiCORE™ wrapper that ties the various building blocks (the integrated block for PCI Express, the GTX transceivers, block RAM, and clocking resources) into an Endpoint or Root Port solution. The system designer has control over many configurable parameters: lane width, maximum payload size, FPGA logic interface speeds, reference clock frequency, and base address register decoding and filtering.

*10/100/1000 Mb/s Ethernet Controller (2,500 Mb/s Supported)*

An integrated Tri-mode Ethernet MAC (TEMAC) block is easily connected to the FPGA logic, the GTX transceivers, and the SelectIO resources. This TEMAC block saves logic resources and design effort. All of the Virtex-6 devices (except the

XC6VLX760) have four TEMAC blocks, implementing the link layer of the OSI protocol stack. The CORE Generator™ software GUI helps to configure flexible interfaces to GTX transceiver or SelectIO technology, to the FPGA logic, and to a microprocessor (when required). The TEMAC is designed to the IEEE Std 802.3-2005 specification. 2,500 Mb/s support is also available.

## 3.2 Software Design Flow

The 13.4 version of the Xilinx design suite, Integrated Software Environment, has been used to implement the design in software. The design should be created, tested and verified in the software before the hardware can be configured.

The first step in the design flow is the HDL description of the circuit. In this step, the design files are created using one of the hardware description languages. For this thesis work, the language which was used is VHDL. These source files can be simulated to verify the functionality of the design in software. However, successful behavioral simulation does not guarantee successful implementation on the hardware.

The next step is to synthesize the design files that were created in the previous step. During this step, the software checks syntax errors, applies user constraints and optimizes the logic to the target device. The constraints include a requirement about the value of the clock frequency and placement of input and output pins based on the physical connections of FPGA pins to circuits on the development board. These connections are described in the documentation for the development board. The output files from this step will be used in the next step.

The third step is the implementation step. During this step, the software verifies whether the design can be implemented on the hardware, for example, it checks how the design will be routed on the chip and optimizes the design according to the timing specifications. The design suite provides tools such as the Floorplan editor and FPGA editor that let the designer to create constraints, and see how the design will be placed and routed on the FPGA, and let the designer perform

placing and routing manually. The software generates detailed analysis reports about the implementation.

The final step in the software design is to generate the programming file to be used configure the FPGA. Therefore , the programming file generated is then downloaded onto the FPGA through JTAG cable.

More information about each step:

Translate : The Translate process merges all of the input netlists and design constraints and outputs a Xilinx Native Generic Database (NGD) file, which describes the logical design reduced to Xilinx primitives.

**Table 3.1** Translate properties

| | |
|---|---|
| Command line tool | NGDBuild |
| Tcl command | process run "Translate" |
| Input files | EDIF, SEDIF, EDN, EDF, NGC, UCF, NCF, URF, NMC, BMM |
| Output files | BLD (report), NGD |
| Tools available after running process | Constraints Editor, PlanAhead software |

Map : The Map process maps the logic defined by an NGD file into FPGA elements, such as CLBs and IOBs. The output design is a Native Circuit Description (NCD) file that physically represents the design mapped to the components in the Xilinx FPGA.

**Table 3.2** Map properties

| | |
|---|---|
| Command line tool | MAP |
| Tcl command | process run "Map" |
| Input files | NGD, NMC, NCD, NGM<br>Note **:** The NCD and NGM files are for guiding. |
| Output files | NCD, PCF, NGM, MRP (report), GRF, MAP, PSR |
| Tools available after running process | FPGA Editor, PlanAhead software, Timing Analyzer |

Place and Route : The Place and Route process takes a mapped NCD file, places and routes the design, and produces an NCD file that is used as input for bitstream generation.

**Table 3.3** Place & Route properties

| | |
|---|---|
| Command line tool | PAR |
| Tcl command | process run "Place & Route" |
| Input files | NCD, PCF<br>Note **:** In addition to the NCD file from MAP, PAR also accepts an NCD file for guiding. |
| Output files | NCD, PAR (report), PAD, CSV, TXT, GRF, |

| | |
|---|---|
| | DLY |
| Tools available after running process | FPGA Editor, PlanAhead software, Timing Analyzer, TRACE, XPower Analyzer |

Generate Programming File : The Generate Programming File process produces a bitstream for Xilinx device configuration. After the design is completely routed, you must configure the device so it can execute the desired function.

**Table 3.4** Generate Programming file properties

| | |
|---|---|
| Command line tool | BitGen |
| Tcl command | process run "Generate Programming File" |
| Input files | NCD, PCF, NKY |
| Output files | BGN, BIN, BIT, DRC, ISC, LL, MSD, MSK, NKY, ISC, RBA, RBB, RBD, RBT |
| Tools available after running process | iMPACT |

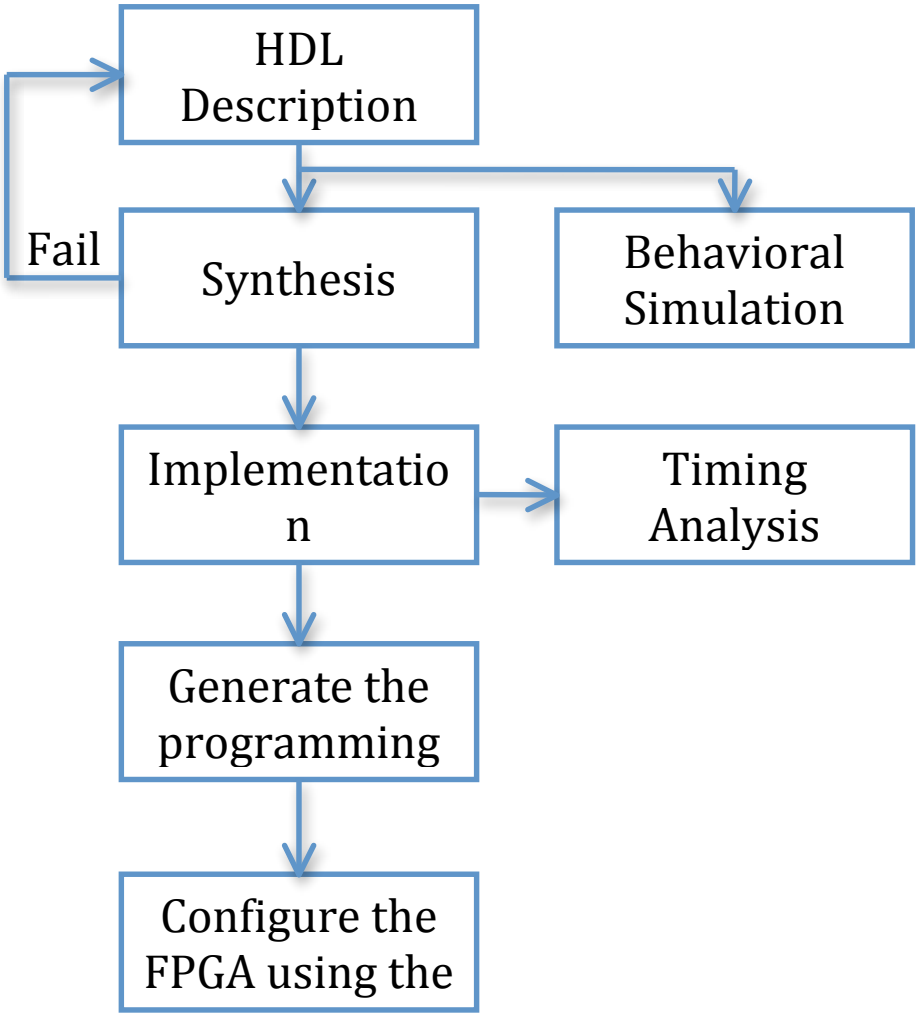



**Figure 3.1** Design Flow

# Chapter 4

# Algorithm Analysis

## The Selection problem

If a *set of "n" unordered numbers* is given, the Linear Time Selection Algorithm, also called Median Finding Algorithm, finds the $k^{th}$ smallest element in the sequence.

## Other Solutions:

If a sequence S = ( a1 , a2 , . . . , an ) of n elements is given, then the *kth s*mallest of the sequence is an element *b* in the sequence such that there are at most *k -1* values for *i* for which $a_i < b$, and at least k values of *i* for which $a_i <= b$. For example, 4 is the second and third smallest element of the sequence 7, 4, 2, 4 .

One obvious solution is to sort the sequence into non-decreasing order and then locate the element at the *kth* position. We already know that this will take *O(n log n*) time both in the average and the worst case. For some values of *k* it is easy to solve the problem more efficiently.

For example, k =1 and k = n correspond, respectively, to finding the minimum and the maximum element, and these can be done in *O(n*) time. Also, when *k* is close to 1, we can construct a *min-heap* (i.e., a heap with the minimum at the root) of the given elements in O(n) time, then do *k deletemin* operations in *O(k log n)* time for a total time of *O(n + k log n*). Note that this *is O(n),* when k is *O(n/ log n)*. Symmetrically, if *k* is close to n we can use a *max-heap* and *deletemax* operations. An important special case occurs when *k = [n/2 ],* in which case we are interested in finding the median. Can we find the median in linear time?

## Why Linear Time Selection Algorithm ?

If a set of "n" unordered numbers is given, the Linear Time Selection Algorithm, also called Median Finding Algorithm, finds the $k^{th}$ smallest element in *O (n)* time in worst case. In order to cut down the running time substantially, it uses elimination and Divide and Conquer strategy.[33]

Another algorithm which will take Order of *O(n)* time is the *quickselect* algorithm. But this algorithm is linear-time only on average. It can require quadratic time with poor pivot choices (consider the case of pivoting around the smallest element at each step).

## Properties of pivot

The chosen pivot is both less and greater than half of the elements in the list of medians, which is around elements for each half. Each of these elements is a median of 5, making it less than 2 other elements and greater than 2 other elements outside the block. Hence, the pivot is less than elements outside the block, and greater than another elements outside the block. Thus the chosen median splits the elements somewhere between 30%/70% and 70%/30%, which assures worst-case linear behavior of the algorithm.

## Basic steps to solve the problem :

Step 1 : If n is small (n < 6 ), then just sort and return the $k^{th}$ smallest number in constant time. ( Bound time – 7 )

Step 2 : If n is not small ( n > 5), then group the given numbers in subsets of 5. ( Bound time n/5 )

Step 3 : Sort the numbers within each group and select the middle elements, which are the medians of each group. ( Bound time – 7n/5 )

Step 4 : Call your *"Selection"* routine recursively to find the median of n/5 medians and call it m. ( Bound time - $T_{n/5}$ )

Step 5 : Compare all n-1 elements with the median of medians m and determine the sets L and R, where L contains all elements <m, and R contains all elements >m. Clearly, the rank of m is $r=|L|+1$ (|L| is the size or cardinality of L). ( Bound time - n )

Step 6 : If $k=r$, then return m, else if $k<r$ then return $k^{th}$ smallest of the set L (Bound time $T_{7n/10}$ ), else return $k-r^{th}$ smallest of the set R.

For example :

We have n elements and the given set is the following: (……..2,5,9,19,24,54,5,87,9,10,44,32,21,13,24,18,26,16,19,25,39,47,56,71,91,61,44,28……).

Because n > 5, we arrange the numbers in groups of five:

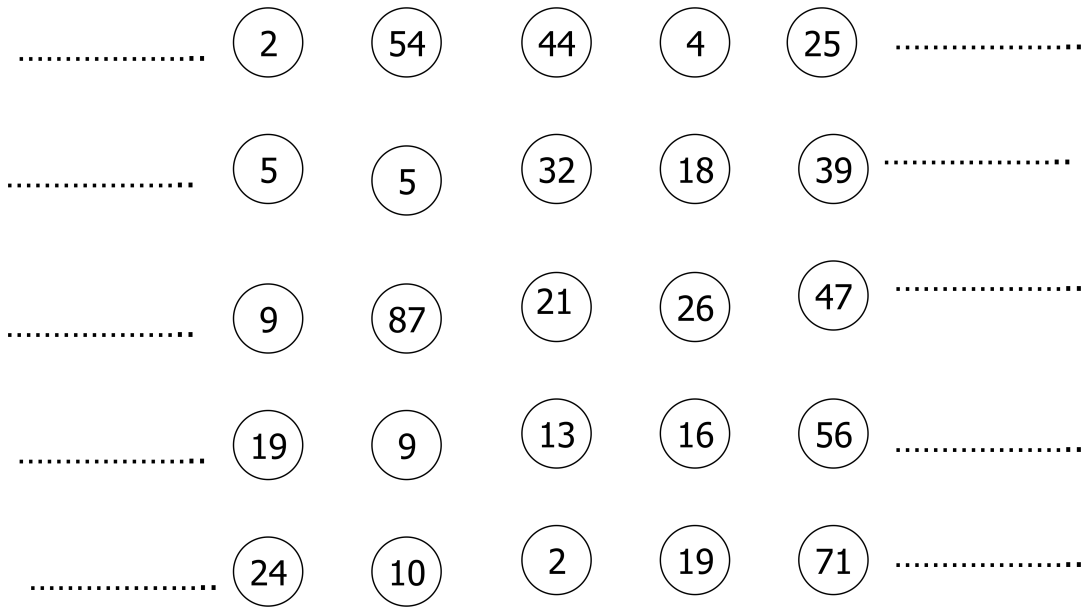



**Figure 4.1** Arrangement in groups of 5

Then we find the medians of each group :

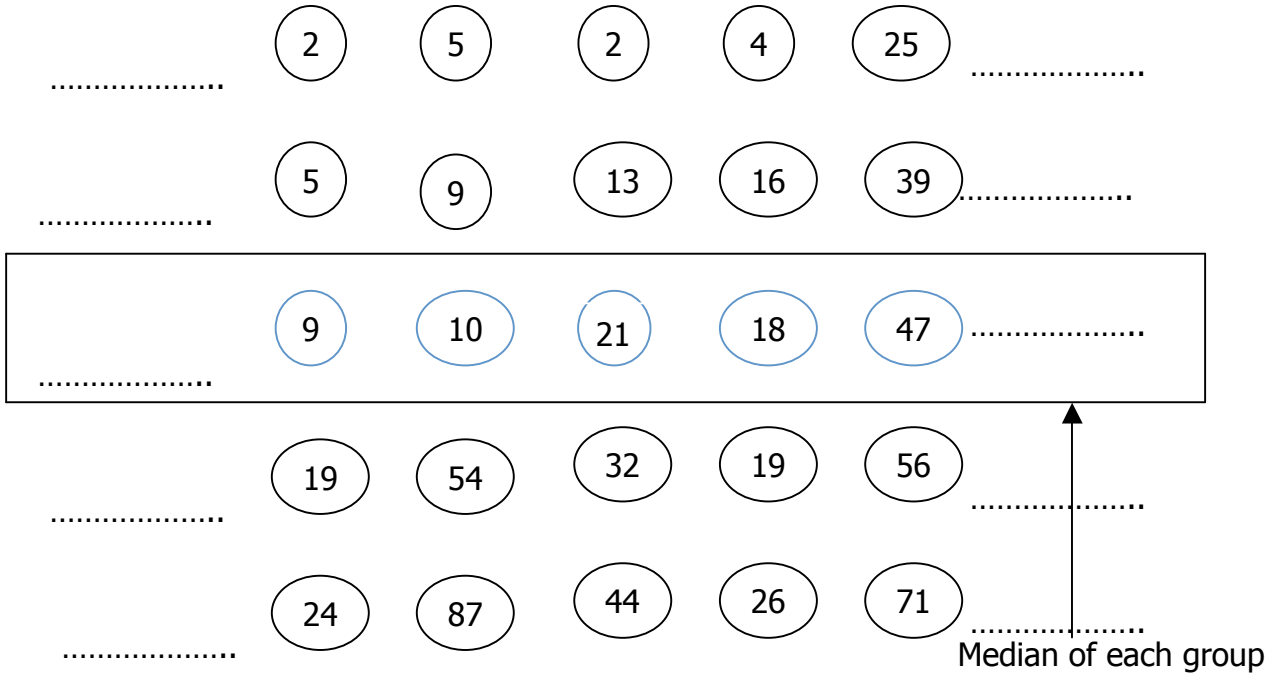



**Figure 4.2** Find the medians of each group

We continue until median of medians is found:

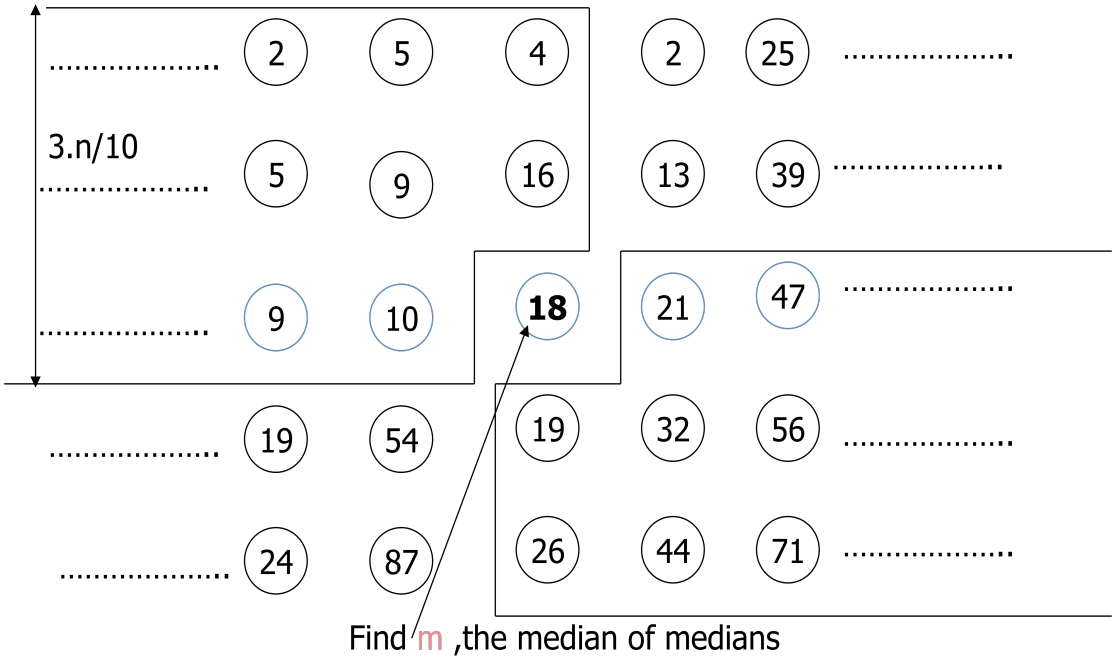
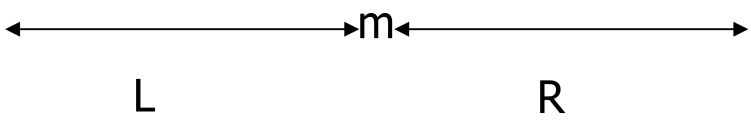



**Figure 4.3** Find the median of medians

We compare each n-1 elements with the median m and find two sets L and R such that every element in L is smaller than M and every element in R is greater than m.




3n/10<L<7n/10 3n/10<R<7n/10

**Figure 4.4** Find LESS and GREATER subsets.

Finally, according to Step 6 we check if the $k^{th}$ smallest number is found, else we have to choose the right subset to continue recursively until we find the wanted number.

## Recursive formula:

T (n)=O (n) + T (n/5) +T (7n/10)

We will solve this equation in order to get the complexity.

We assume that T (n)< C*n

T (n) = a*n + T (n/5) + T (7n/10)

C*n >= T(n/5) +T(7n/10) + a*n

C >= C*n/5+ C*7*n/5 + a*n

C >= 9*C/10 +a

C/10 >= a

C >= 10*a

There is such a constant that exists…. so T (n) = O (n)

## Why groups of five and not some other term?

If we divide elements into groups of 3 then we will have

T (n) = O (n) + T (n/3) + T (2n/3) so T (n) > O (n)…..

If we divide elements into groups of more than 5, the value of constant 5 will be more, so grouping elements in to 5 is the optimal situation.

# Chapter 5

# Implementation

We have implemented the Linear Time Selection Algorithm with two different architectures in mind, thus we have two versions. Version 1 uses less RAM memories than Version 2, although Version 2 is faster.

## 5.1 Version 1: Implementation with minimum memory requirements

In Version 1, we firstly store all the numbers in Initial_RAM or LESS and GREATER memories. Then we start arranging them in groups of 5. The components we have used are the following: median_top, m_finder, mergesort, compexch, Init_RAM, SRAM and S_RAMEM components.

Our program is parametric, therefore we are able to change easily the width of the words and the depth of our memories.

Parameters (Generic) :

- width : determines the size of the word.
- bits_depth : determines the size of the addresses of Initial RAM and SRAM memories.
- depth_small : determines the size of the addresses of S_RAMEM memories.

The picture below depicts the connectivity among different kernels:

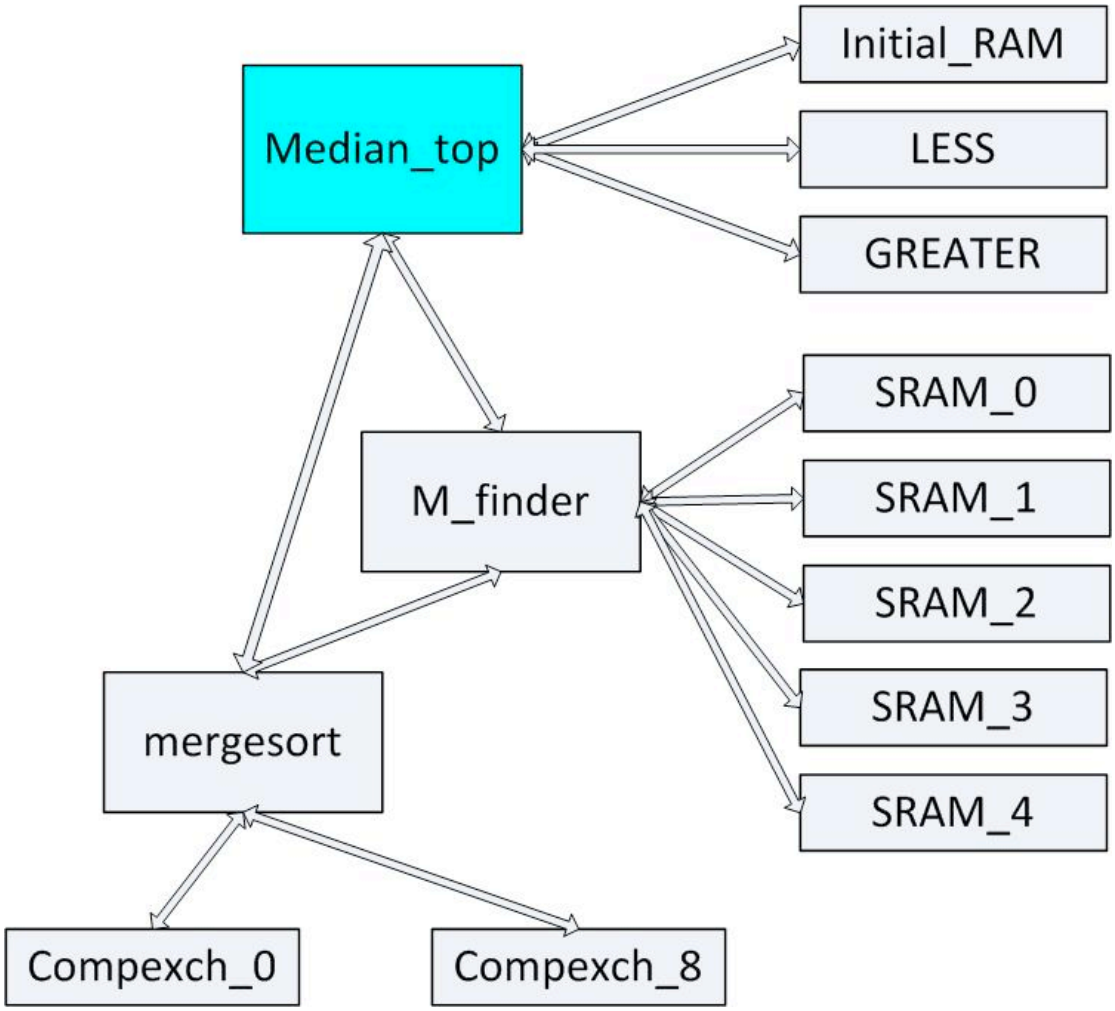



**Figure 5.1** Connectivity among kernels

### 5.1.1 Median_top component

Median_top is the top module. At the beginning all the numbers are inserted here and median_top writes them to the Initial_RAM memory. When writing is done, we examine if we have less than 6 or more than 5 elements. In the first case mergesort component is enabled and the value of k is assigned to wanted_merge signal. The output of mergesort is the result of median finding algorithm too. Its value is assigned to the signal mdn_out and writeback becomes '1'. In the other case(more than 5 numbers) m_finder component is enabled and we start

supplying it with data. When signal mdn_found is equal to '1' then pivot is found. Later on, we compare its value with each element and find two sets. In the first one, every element is smaller than pivot (LESS), whilst in the other one, every element is greater than pivot (GREATER). To store these two sets we use the 2 SRAM memories. When the comparison is over, we have to examine if the wanted result is found, otherwise we search in which subset k (the position of the number we are looking for) belongs to. If it belongs in LESS subset ( lls <= '1' , grt <= '0') then we will read from LESS, else ( lls <= '0' , grt <= '1') from GREATER instead of Initial_RAM. The process continues running recursively until the $k^{th}$ smallest number that we asked for is found.

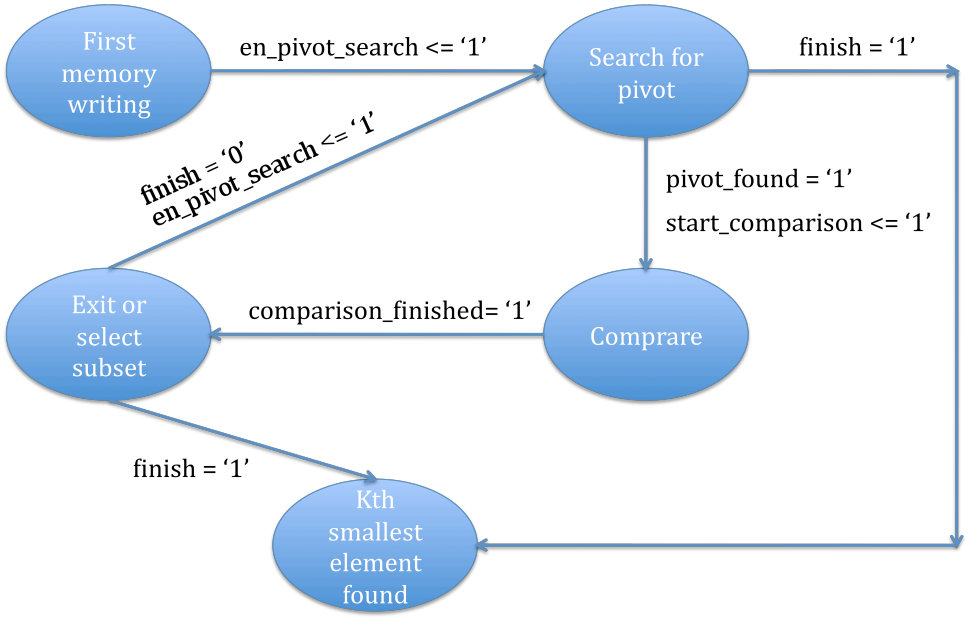



**Figure 5.2** Main steps of the operation of median_top component

Component Description

Inputs

- rst (std_logic) : reset signal.
- clock (std_logic)

- lmt ((std_logic_vector(bits_depth-1 downto 0)) : if we want to restrict our search in a smaller set of numbers than the initially given set (or if we have not filled the initially defined memory in the case we use the hardware testbench) then lmt signal must have a value different than zero ( instead of "n" elements, if lmt > 0 then we will have (lmt+1) ).
- max_addr (std_logic_vector (bits_depth-1 downto 0)) : it is equal to n -1.
- wanted_pos (std_logic_vector (bits_depth-1 downto 0)) : indicates the position of the number we are looking for.
- mdn_datain (std_logic_vector(width-1 downto 0)) : input data.
- median_wr_en (std_logic) : shows when writing data to median_top 's local memory is enabled.

Outputs :

- num_mdn_out (std_logic_vector(0 downto 0)) : number of outputs. For our case it is equal to one.
- writeback (std_logic) : request to send data back.
- mdn_out ((std_logic_vector(width-1 downto 0)): result of the median finding algorithm


**Figure 5.3** Median_top component

Next, Figure 5.4 shows an output from Modelsim.

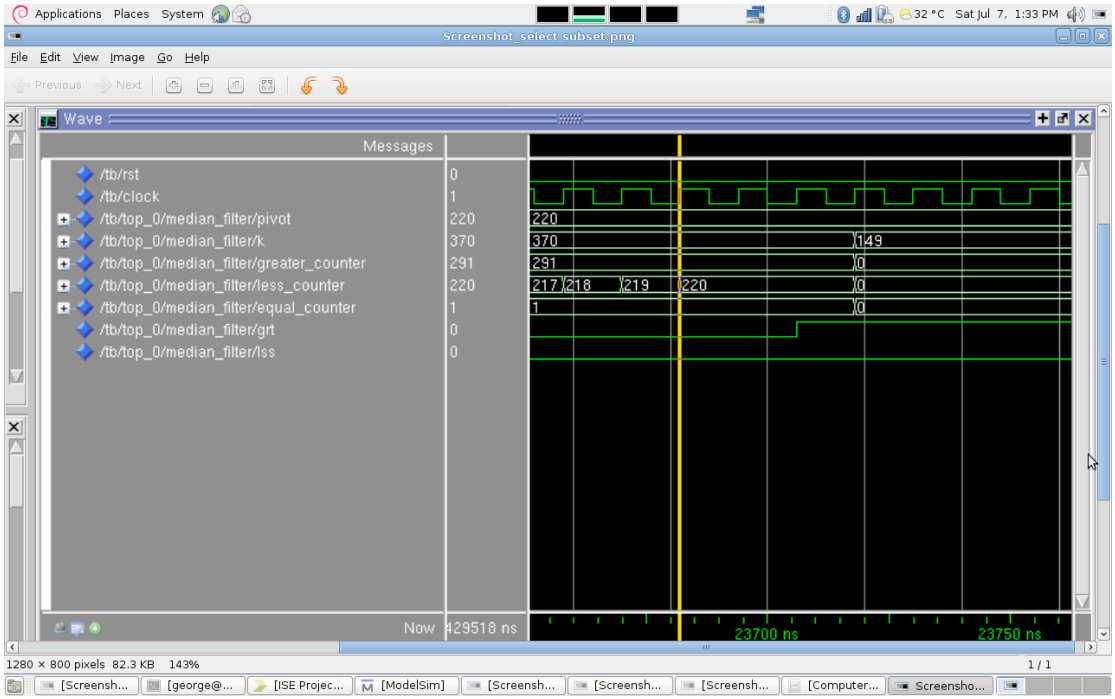



Figure **5.4** Select the right subset, if the wanted number is not found yet.

### 5.1.2 M_finder component

In case we have more than 5 numbers, m _finder component is used. Firstly, it arranges the numbers in groups of five and secondly, mergesort is applied. For the partitioning of the numbers the 5 S_RAMEM memories are used. The results of mergesort (medians) are stored back to the 5 S_RAMEM memories (overwrite). The same process continues running until the median of medians is found. When this is found we assign the value '1' to the signal mdn_found and the value of the median of medians to the signal f_median.

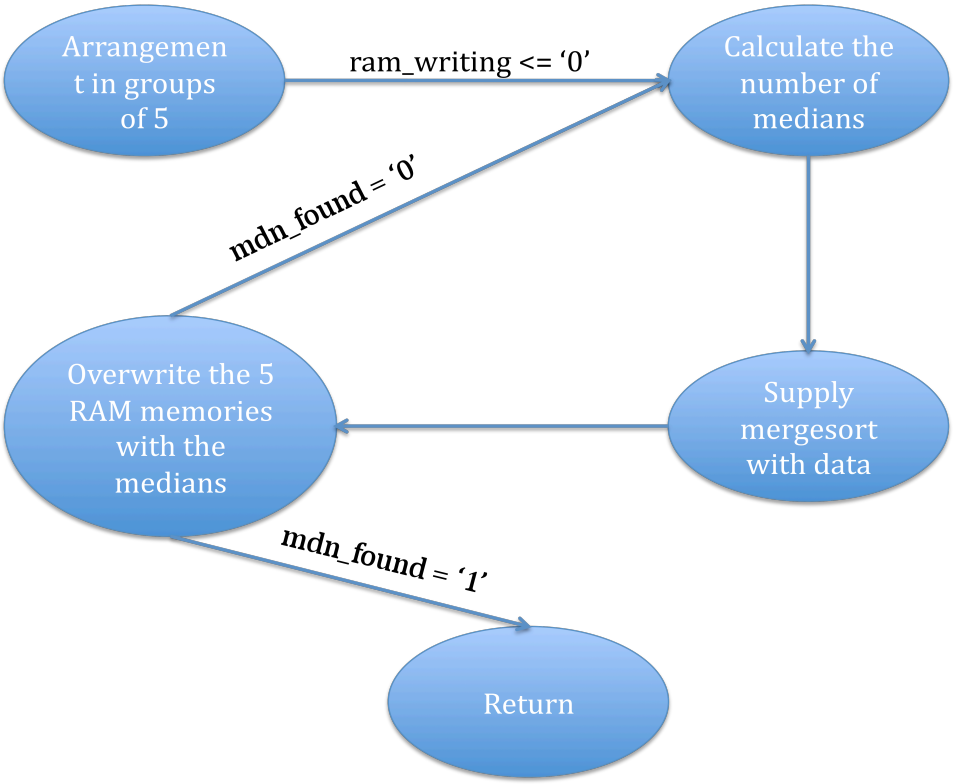



**Figure 5.5** Main steps of the operation of median_top component

Component Description

Inputs

- rst (std_logic) : reset signal.
- clock (std_logic)
- inp_num (std_logic_vector(bits_depth-1 downto 0)) : number of input data.
- new_input (std_logic_vector(width-1 downto 0))
- m_f_en (std_logic) : shows when median_top component sends data to m_finder.

Outputs

- f_median (std_logic_vector(width-1 downto 0)) : median of medians.
- mdn_found (std_logic) : indicates when median of medians is found.

Next, Figure 5.6 and Figure 5.7 show how m_finder component supplies the mergesort with data and how it stores medians back to the 5 S_RAMEM memories.

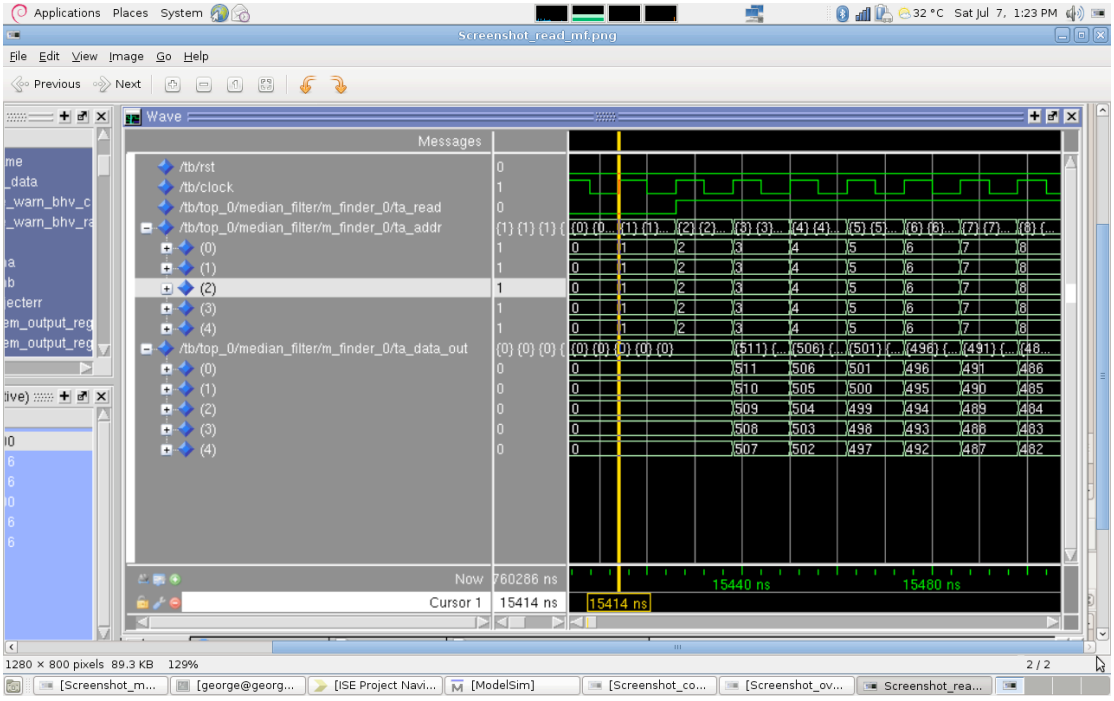



**Figure 5.6** Read from the 5 RAM memories and supply with data the mergesort component.

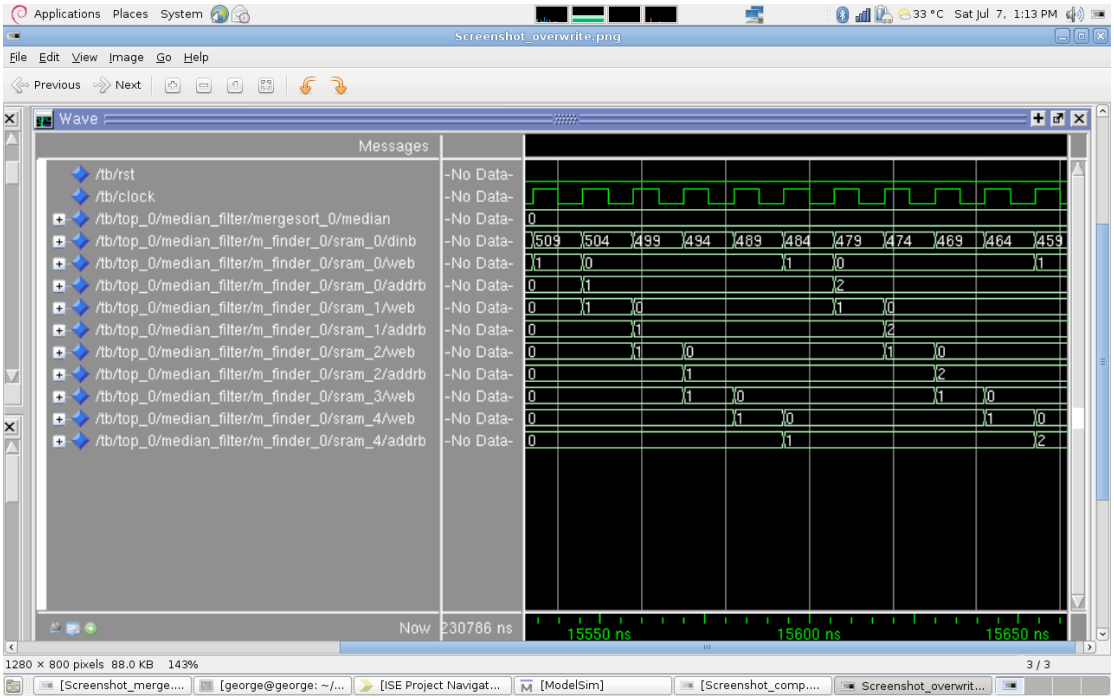



**Figure 5.7** Overwrite the 5 RAM memories with the results of mergesort component, until median of medians is found.

### 5.1.3 Mergesort component

We chose to use merge sort for the sorting of each group. Merge sort parallelizes well due to use of divide-and-conquer method and is one of the most efficient methods for the sorting of 5 or less elements .It also allows us to implement pipelined sorting of many groups. For the pipelined and parallelized implementation we use 9 compexch components. If we have 5 input numbers, then we select as median the third number. If we have 3 or 4 input numbers, then we select as median the second number. In case we have less than 3 input numbers, we select as median the first number. The only exception is when mergesort component is enabled by the median_top component. In this case the value of k is assigned to wanted signal.

According to the mergesort algorithm:

**Table 5.1** Comparisons per clock cycle

| Clock cycles | t = t1 | t = t2 | t = t3 | t = t4 | t = t5 |
|---|---|---|---|---|---|
| **Compexch Inputs** | [0, 4] | [0, 2] | [2, 4] | [2, 3] | [1, 2] |
| **Compexch Inputs** | [1, 3] | | [0, 1] | [1, 4] | [3, 4] |

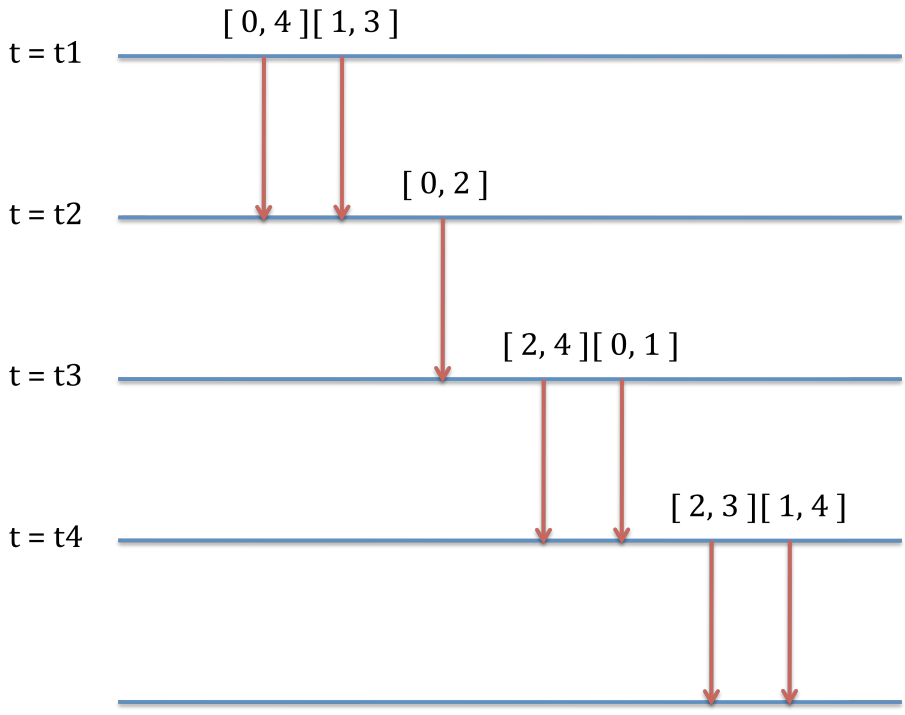



**Figure 5.8** Steps of mergesort for 5 elements

Component Description

Inputs

- rst (std_logic) : reset signal.
- clock (std_logic)
- wanted (std_logic_vector(2 downto 0)) : which number we want as an output after the sorting.
- rest (std_logic_vector(2 downto 0)) : number of data, which we will be sorted.
- arr0a , arr1a, arr2a, arr3a, arr4a (std_logic_vector(width-1 downto 0)) : input data.

Outputs

- median (std_logic_vector(width-1 downto 0)) : the middle element selected after the sorting.

Next figure shows the inputs and outputs of mergesort component.

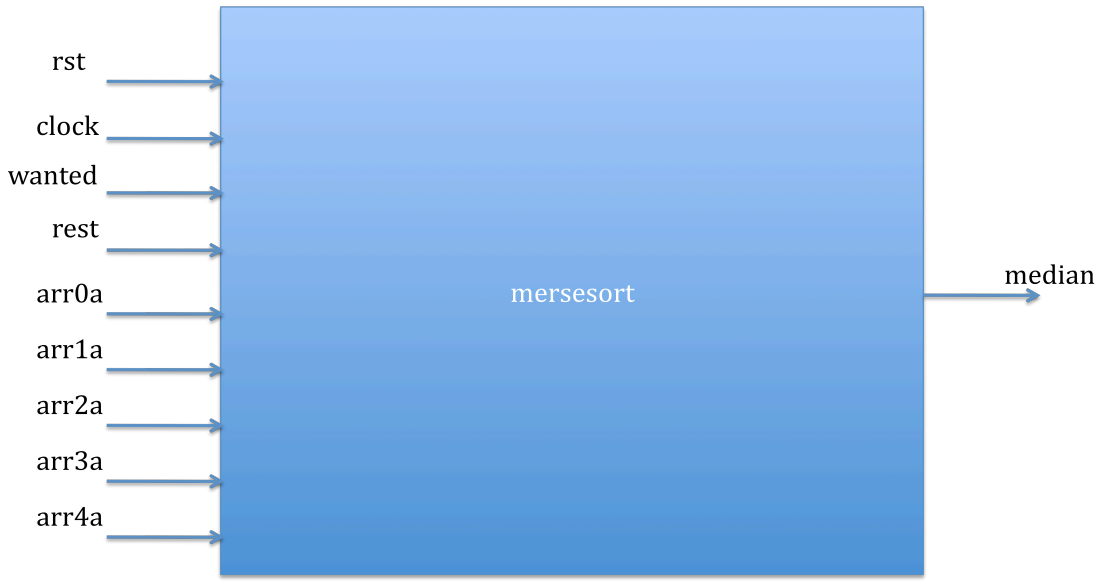



**Figure 5.9** Mergesort component.

Figure 5.10 shows how the pipelined operation of mergesort component. In each cycle a new group of 5 elements inserts.

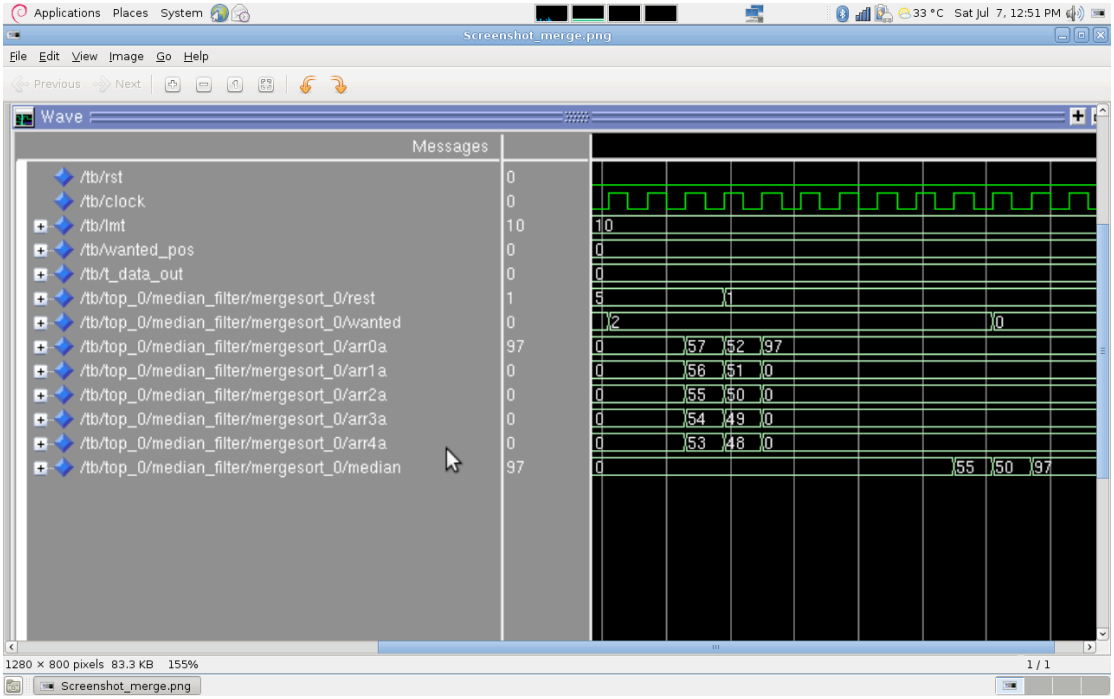



**Figure 5.10** Pipeline. In every clock cycle merge sort is applied to a new group of numbers.

## 5.1.4 Compexch component

Compexch component compares the values of a and b. If a > b , then it switches their position.

Component Description

Inputs

- rst (std_logic) : reset signal.
- clock (std_logic)
- a, b (std_logic_vector(conv_integer(width)-1 downto 0)) : input data.

Outputs

- c, d (std_logic_vector(conv_integer(width)-1 downto 0)) : output data.

Next figure shows the inputs and outputs of compeexch component.


**Figure 5.11** Mergesort component.

Figure 5.12 shows the operation of compexch component.

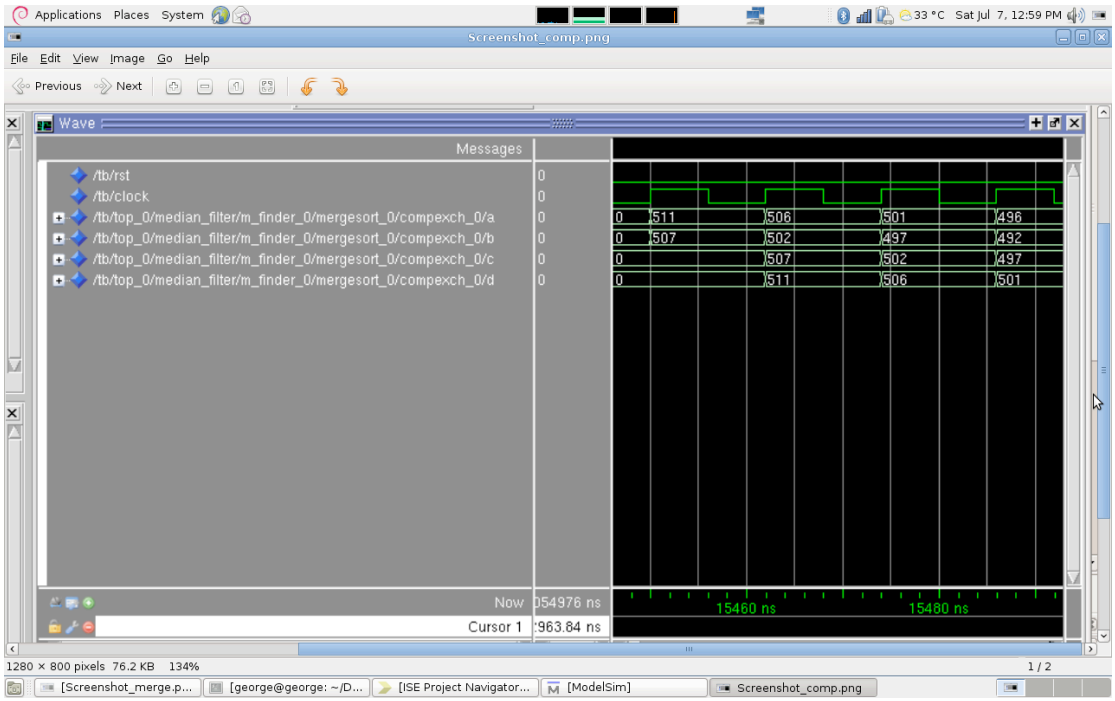



**Figure 5.12** Compare a and b inputs and if it is needed switch their positions.

### 5.1.5 Init_RAM component

Single port RAM memory created with ISE's Block Memory Generator ( Write first). Always enabled. Used for Initial_RAM.

Component Description

Inputs

- clka (std_logic)
- regcea (std_logic) : read.
- wea (std_logic_vector(0 downto 0)): write.
- addra (std_logic_vector(bits_depth-1 downto 0)) : address.
- dina (std_logic_vector(width-1 downto 0) : input.

Outputs

- douta (std_logic_vector(width-1 downto 0) : output.

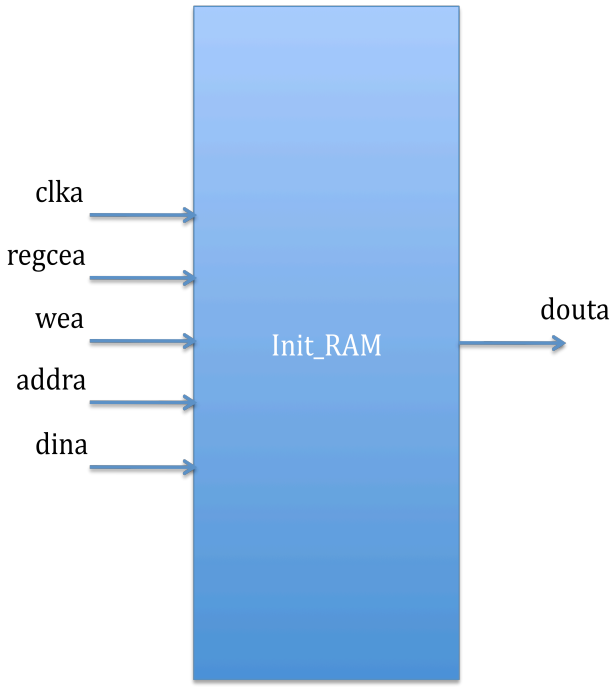



**Figure 5.13** Init_RAM component.

### 5.1.6 SRAM component

True dual port RAM memory created with ISE's Block Memory Generator (Read first). Always enabled. Common clock. Used for LESS and GREATER.

Component Description

Inputs

- clka (std_logic) : clock
- rsta (std_logic) : reset.
- regcea (std_logic) : read from address addra.
- wea (std_logic_vector(0 downto 0)): write to address addra.
- addra (std_logic_vector(bits_depth-1 downto 0)) : address a.
- dina (std_logic_vector(width-1 downto 0) : input to address addra.
- clkb (std_logic) : clock
- rsta (std_logic) : reset.
- regceb (std_logic) : read from address addrb.
- web (std_logic_vector(0 downto 0)): write to address addrb.
- addrb (std_logic_vector(bits_depth-1 downto 0)) : address b.
- dinb (std_logic_vector(width-1 downto 0) : input to address addrb.

Outputs

- douta (std_logic_vector(width-1 downto 0) : output when regcea = '1'.
- doutb (std_logic_vector(width-1 downto 0) : output thwn regceb = '1'.

.

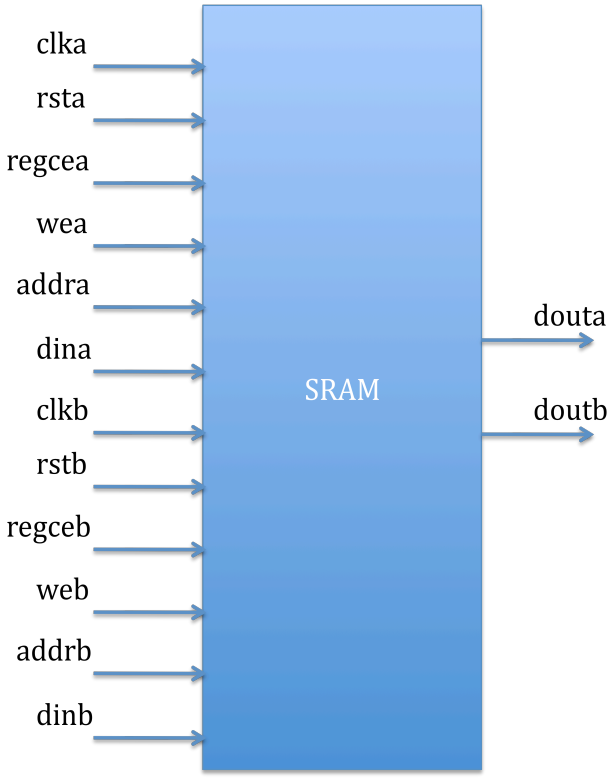



**Figure 5.14** SRAM component.

### 5.1.7 S_RAMEM component

True dual port RAM memory created with ISE's Block Memory Generator ( Read first). Used for SRAM_0, SRAM_1, SRAM_2, SRAM_3, SRAM_4. S_RAMEM memories need to have 1/5 of the size of SRAM memories. Always enabled. Common clock. They are used for the arrangement of the numbers in groups of 5.

Component Description

Inputs

- clka (std_logic) : clock.
- regcea (std_logic) : read from address addra.
- wea (std_logic_vector(0 downto 0)): write to address addra.
- addra (std_logic_vector(depth_small-1 downto 0)) : address a.
- dina (std_logic_vector(width-1 downto 0) : input to address addra.

- clkb  (std_logic) : clock.
- regceb (std_logic) : read from address addrb.
- web (std_logic_vector(0 downto 0)): write to address addrb.
- addrb (std_logic_vector(depth_small-1 downto 0)) : address b.
- dinb (std_logic_vector(width-1 downto 0) : input to address addrb.

Outputs

- douta (std_logic_vector(width-1 downto 0) : output when regcea = '1'.
- doutb (std_logic_vector(width-1 downto 0) : output thwn regceb = '1'.

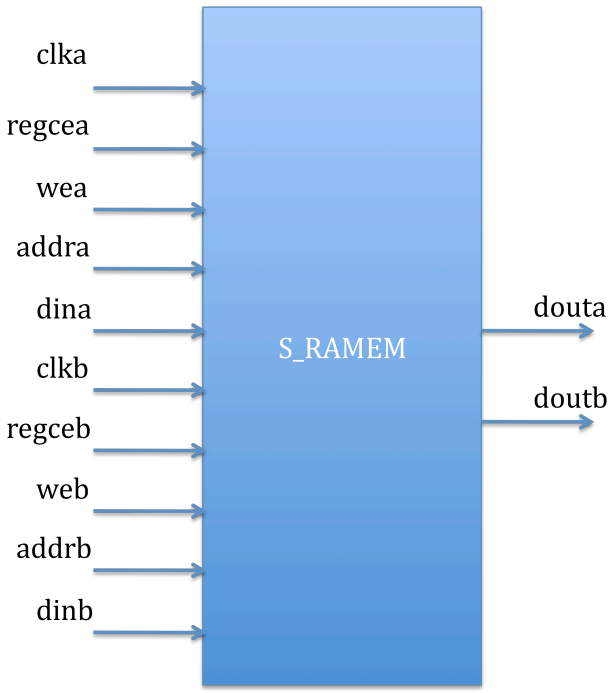



**Figure 5.15** S_RAMEM component.

## 5.2 Version 2: High-performance

In the implementation of the second version we took advantage of the capability to arrange the elements in groups of 5 in parallel with their storage either in Initial_RAM or LESS and GREATER memories. In this case we save much time, but we need to use S_RAMEM component 5 more times. Mergesort, compexch, Init_RAM, SRAM and S_RAMEM components are the same as Version 1. Also, both versions have the same parameters.

Their differences on what concerns their architecture are the following:

- Version 2 uses 10 S_RAMEM memories and Version 1 uses 5 of them.
- Version 2 does not have the m_finder component.
- The two versions have different top modules.

The picture below depicts the connectivity among different kernels:

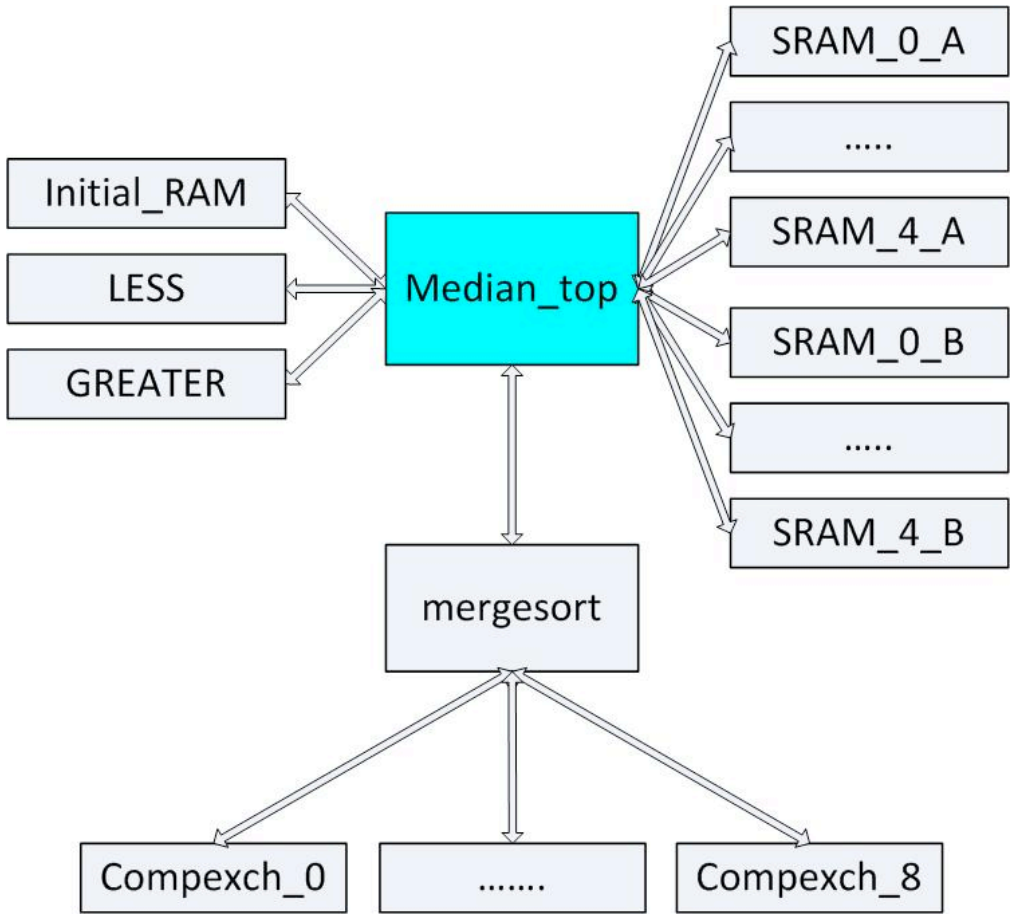



**Figure 5.16** Connectivity among kernels.

## Median_top component

Median_top is the top module. In the beginning all the numbers insert here and median_top writes them in parallel to Initial_RAM and to SRAM_0_A, SRAM_1_A, SRAM_2_A, SRAM_3_A, SRAM_4_A memories. By this time the arrangement of numbers in groups of 5 is already done. When writing has been finished, we examine if we have less than 6 or more than 5 elements. In the first case mergesort component is enabled and the value of k is assigned to wanted_merge signal. Rest signal takes the value of input_num( shows the number of inputs). The output of mergesort is the result of median finding algorithm too. Its value, is assigned to the signal mdn_out and writeback becomes '1'. In the other case mergesort component is again enabled, but now all the outputs (medians of each group) are written back to the 5 S_RAMEM memories (overwrite the old data that we do not need anymore). This task is over, when median of medians is found. Then signal mdn_found is equal to '1' . After that we compare its value with each element and find two sets. In the first one every element is smaller than pivot (LESS) and in the other one, every element is greater than pivot (GREATER). To store these two sets we use the 2 SRAM memories. In parallel we arrange the elements again in groups of 5. The numbers that belong to LESS subset are written to SRAM_0_B, SRAM_1_B, SRAM_2_B, SRAM_3_B, SRAM_4_B memories and the numbers that belong to GREATER subset to the SRAM_0_A, SRAM_1_A, SRAM_2_A, SRAM_3_A, SRAM_4_A memories. When the comparison is done, we examine if the wanted result is found. If not, we search in which subset k (the position of the number we are looking for) belongs to. If it belongs to LESS subset ( lls <= '1' , grt <= '0') then in the next run we will read from LESS and SRAM_0_B, SRAM_1_B, SRAM_2_B, SRAM_3_B, SRAM_4_B memories, otherwise ( lls <= '0' , grt <= '1') from GREATER and SRAM_0_A, SRAM_1_A, SRAM_2_A, SRAM_3_A, SRAM_4_A memories instead of Initial_RAM. The process continues running until the $k^{th}$ smallest number that we demanded is found.

Component Description

Inputs

All the input signals are the same with the Version 1.

Outputs

All the output signals are the same with the Version 1.

## 5.3 Comparison between Version 1 vs. Version 2

In the following table and figures we present the differences of the two implemented version with more details. As we have referred and above, all the changes that affect the operation of our core have been done in the top module.

**Table 5.2** Version 1 vs Version 2

| Step | Operation | Comparison | |
|---|---|---|---|
| | | Version 1 | Version 2 |
| **1** | First memory writing | We store the input numbers in Initial_RAM. | We store the input numbers in Initial_RAM and in SRAM_0_A, SRAM_1_A, SRAM_2_A, SRAM_3_A, SRAM_4_A (arrangement in groups of 5) in parallel. |
| **2** | Search for pivot | We read from Initial_RAM or LESS or GREATER and supply | We read from the the 5 S_RAMEM memories and enable the mergesort. The |

| | | | |
|---|---|---|---|
| | | m_finder with data. The grouping and the enable of mergesort is doing there. | results of mergesort are written back to the 5 S_RAMEM memories. We continue until median of medians(pivot) is found. |
| **3** | Compare | We compare the value of pivot with the value of the numbers read from Initial_RAM(lss='0' and grt ='0') or LESS(lss='1') or GREATER(grt='1') memory and we find the new two subsets. The numbers compared are stored either to LESS or GREATER memory. | Version 2 does the same with Version 1, but during the comparison it writes the numbers and to the 10 S_RAMEM memories in parallel. SRAM_0_A, SRAM_1_A, SRAM_2_A, SRAM_3_A, SRAM_4_A are used for the storage of greater than pivot numbers. SRAM_0_B, SRAM_1_B, SRAM_2_B, SRAM_3_B, SRAM_4_B are used for the storage of smaller than pivot numbers. |

The figures below show the different way that the two version act in the second step of their operation, as we have described it above.

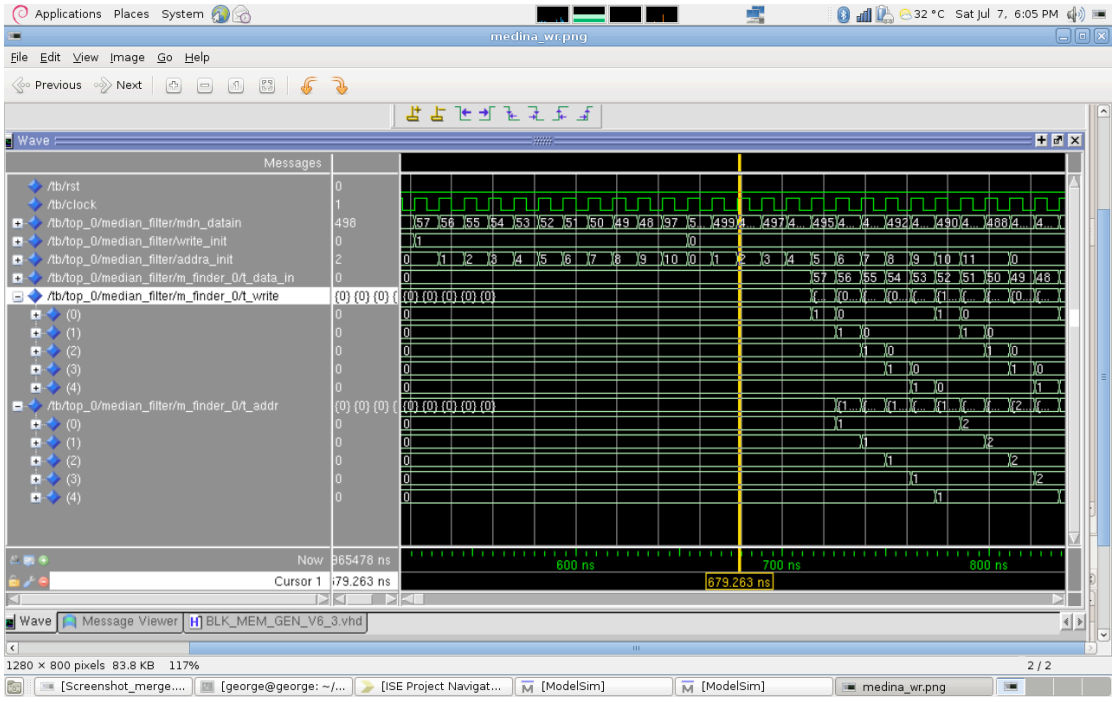



**Figure 5.17** Step 2 by Version 1.

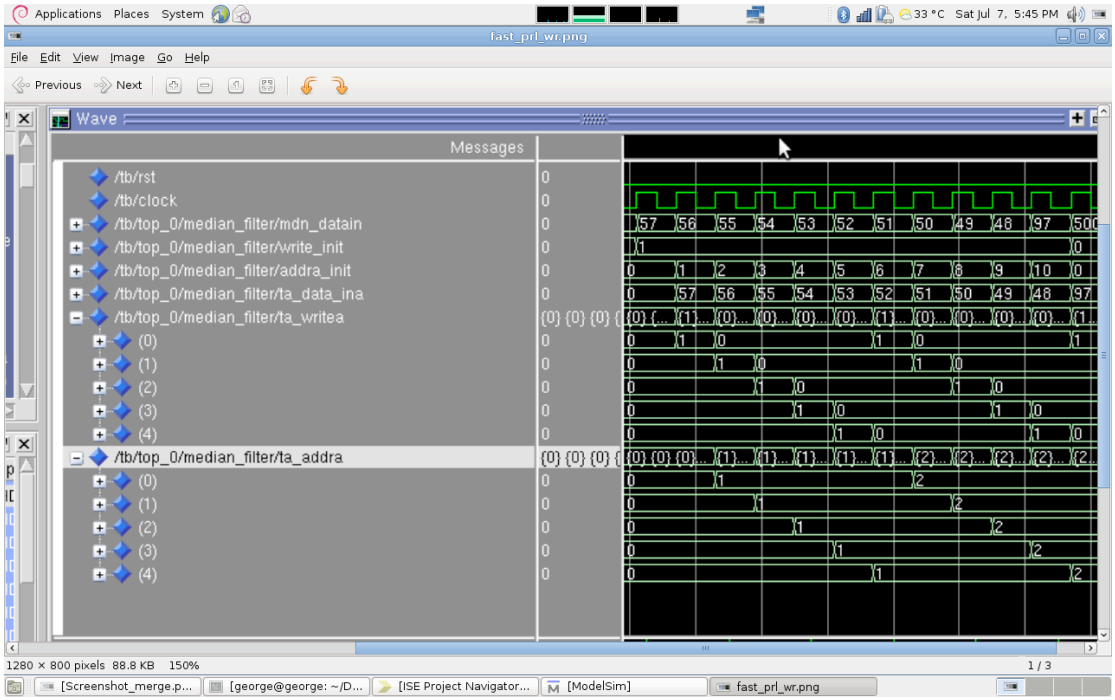



**Figure 5.18** Step 2 by Version 2.

# Chapter 6

# Communication

As it has been previously mentioned, the goal of this thesis is not only the FPGA implementation of a core, but also to provide a full integration between software and hardware. In order to achieve that, we have implemented several other modules, for enabling communication between software and hardware.

Thus, we need a driver that will be able to transfer data between the local host and the FPGA device and a controller who will manage the data flow. In the case we have more than one cores attached, the controller will be responsible and for the management of the writeback priorities.

## 6.1 Integration with Ethernet

Driver Description :

The main goal of the driver is to read the data from the local host and sending them over Ethernet cable to our controller. Moreover, it is responsible for getting data from the controller and send them to the local host over the Ethernet again. Use of Ethernet provides us with greater flexibility and will make core simulations easier as there is no need for someone to be familiar with VHDL or how to run a VHDL program. The only thing the user has to know is a high level language. In this thesis, the program was written in C language. So, driver is responsible for reading data that are generated by the C program.

First of all, driver reads 4 bytes containing the number of data that are going to be transferred. After that, driver reads packages of 4 bytes until data are finished. Every 32bit data that is read is being transferred in the controller. We split them in 4 words(each word has 8 bits width) and we start supplying the cores.

When the reading is finished the driver gets in a state where variables for write back process are initialized, such as number of output data. Driver reads data from the controller and stores them in a char buffer until the driver receives and acknowledgement signal. At this time, data are being transferred back.

When process is done, driver returns to its initial state. Obviously, driver is described with the use of an FSM, with each state's functionality to be described above.

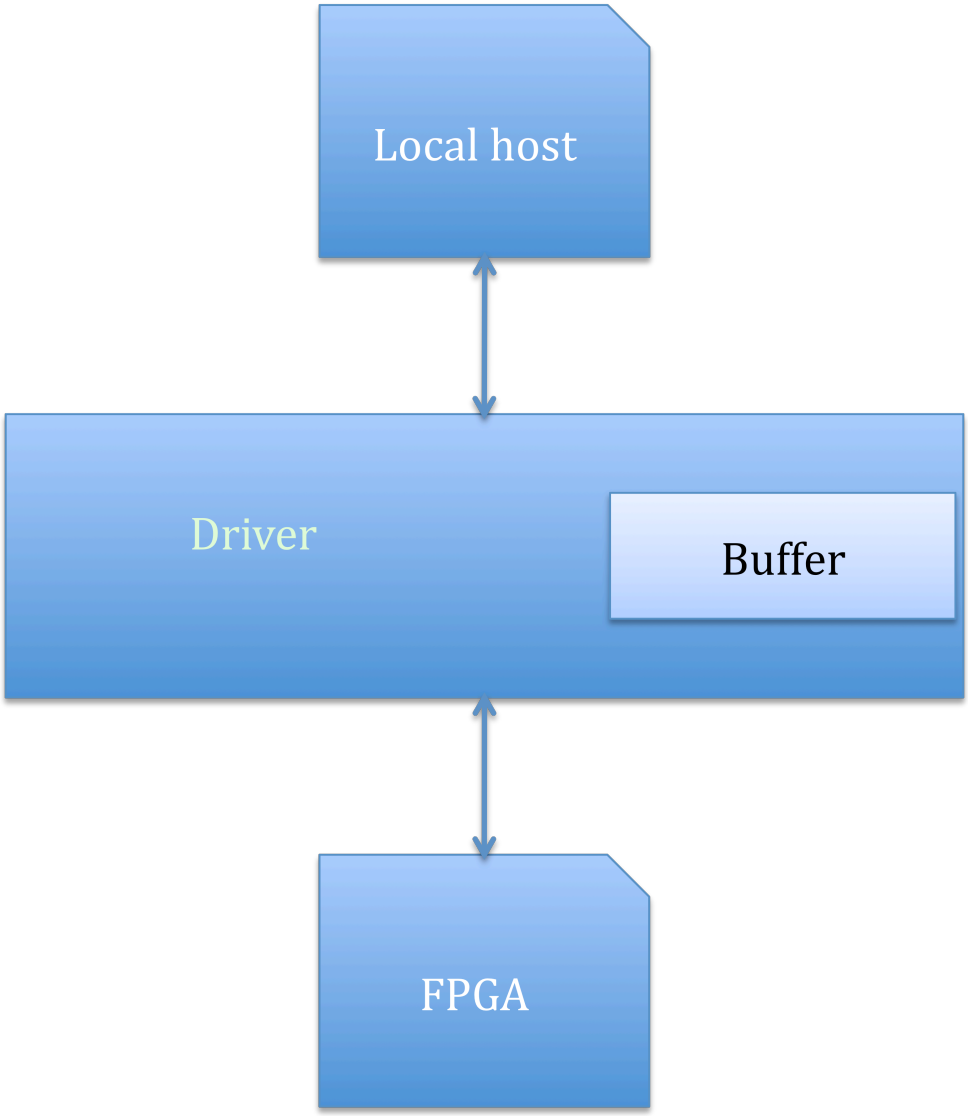



**Figure 6.1** Connection

Now we give a description of the main steps:

*Step 1*: we trigger the reception process (state 0)

*Step 2* : we wait until the local host is ready to send data to the FPGA (state 1).

*Step 3 :* In the next step we wait until a frame is received (state 2) and we check for proper frame reception (state 3). This frame should contain information about the number of data that will be sent.

*Step 4 :* This information is stored (state 5)

*Step 5 :* After that we start receiving data (state 6) and we always check for proper frame reception. All the inputs are stored in a buffer.

*Step 6 :* We read from the buffer and we start supplying the cores with data. In the buffer the word size is 32 bits so we have to split each one in 4 x 8 bits (states 14, 15, 16 , 17), when we sent the numbers to the cores. When the driver is sending data to the cores the value of enable_wr signal is '1'.

*Step 7 :* We check if we have read all the bytes. If there are more number to be read then we repeat the process described above (state 18).

*Step 8 :* We wait for the core to finish its task. When the core is ready to send data to the driver then writeback signal is equal to '1'.

*Step 9 :* We store the core's output to the driver's buffer (state 27)

*Step 10 :* The transmission from the FPGA to the local host starts and we always check if there is any error. Because the output of our core is only 1, we know that only one 1 byte need to be sent.

*Step 11 :* We go back to state 1 and wait for new request for data transmission.

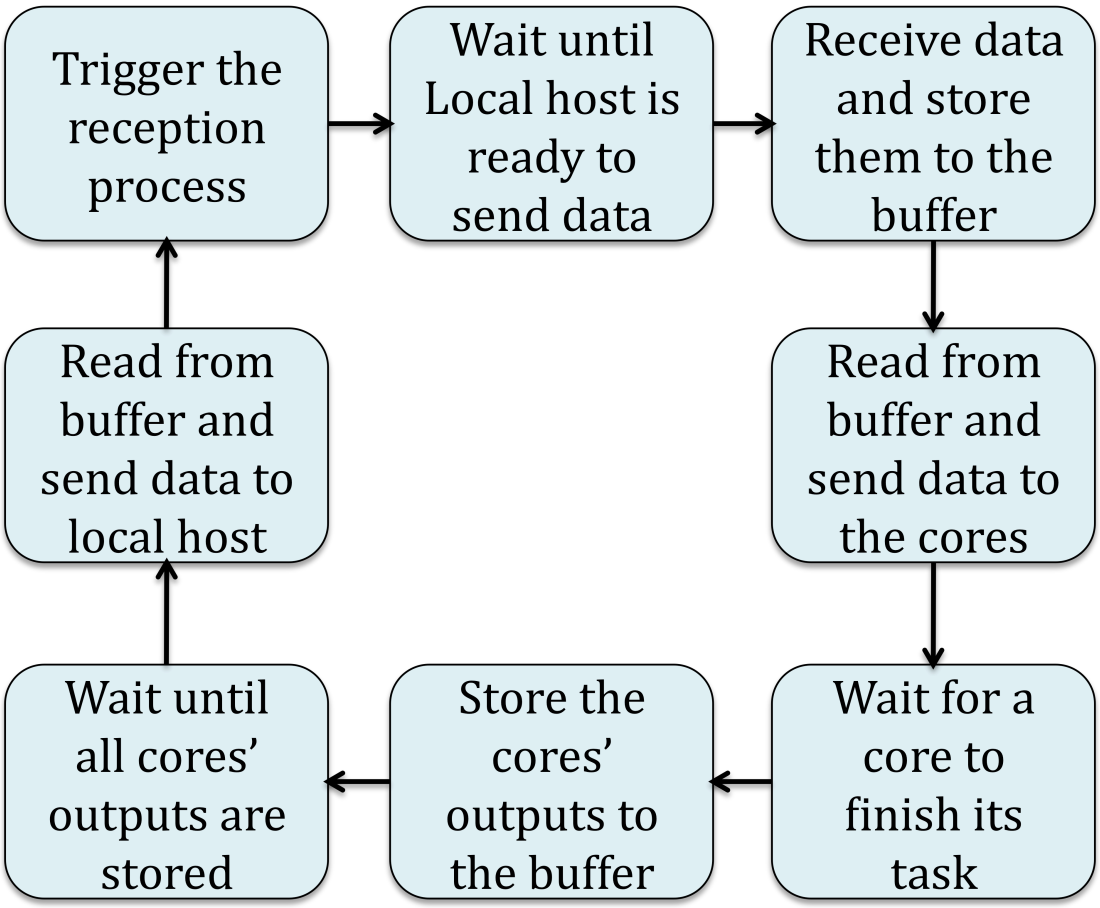



**Figure 6.2** Main steps

To send the data from the Local host to the FPGA we have to run the tx_file_plus_len, followed by the name of the file that we want to send.

To receive data from the FPGA we have to run the rx_file, followed by the name of the file that the results will be written, the number of expected bytes and a value for the append flag.

The C programs that we run for the communication send and read ASCII characters.

Controller Component Description :

Inputs :

- clk (std_logic)
- rst (std_logic)
- E_COL (std_logic) : Collision Detected. The PHY asynchronously asserts the collision signal E_COL after the collision has been detected on the media. When deasserted, no collision is detected on the media.
- E_CRS (std_logic) : Carrier Sense. The PHY asynchronously asserts the carrier sense E_CRS signal after the medium is detected in a non-idle state. When deasserted, this signal indicates that the media is in an idle state (and the transmission can start).
- E_MDC (std_logic) : Management Data Clock. This is a clock for the E_MDIO serial data channel.
- E_MDIO (std_logic) : Management Data Input/Output. Bi-directional serial data channel for PHY/STA communication.
- E_RX_CLK (std_logic) : Transmit Nibble or Symbol Clock. The PHY provides the E_Tx_Clk signal. It operates at a frequency of 25 MHz (100 Mbps) or 2.5 MHz (10 Mbps). The clock is used as a timing reference for the transfer of E_TXD[3:0], E_TX_EN, and E_TX_ER.
- E_RX_DV (std_logic) : Receive Data Valid. The PHY asserts this signal to indicate to the Rx MAC that it is presenting the valid.
- E_RXD (std_logic) : Receive Data Nibble. These signals are the receive data nibble. They are synchronized to the rising edge of E_RX_CLK. When E_RX_DV is asserted, the PHY sends a data nibble to the Rx MAC. For a correctly interpreted frame, seven bytes of a preamble and a completely formed SFD must be passed across the interface.
- E_TX_CLK (std_logic) : Transmit Nibble or Symbol Clock. The PHY provides the E_Tx_Clk signal. It operates at a frequency of 25 MHz (100 Mbps) or 2.5 MHz (10 Mbps). The clock is used as a timing reference for the transfer of E_TXD[3:0], E_TX_EN, and E_TX_ER.

- E_RX_ER (std_logic) : Receive Error. The PHY asserts this signal to indicate to the Rx MAC that a media error was detected during the transmission of the current frame. E_RX_ER is synchronous to the E_RX_CLK and is asserted for one or more E_RX_CLK clock periods and then deasserted.

Outputs :

- E_MDIO (std_logic) : Management Data Input/Output. Bi-directional serial data channel for PHY/STA communication.
- E_TX_EN (std_logic) : Transmit Enable. When asserted, this signal indicates to the PHY that the data E_TXD[3:0] is valid and the transmission can start. The transmission starts with the first nibble of the preamble. The signal remains asserted until all nibbles to be transmitted are presented to the PHY. It is deasserted prior to the first E_TX_CLK, following the final nibble of a frame.
- E_TXD (std_logic) : Transmit Data Nibble. Signals are the transmit data nibbles. They are synchronized to the rising edge of E_TX_CLK. When E_TX_EN is asserted, PHY accepts the E_TXD.
- E_TX_ER (std_logic) : Transmit Coding Error. When asserted for one E_TX_CLK clock period while E_TX_EN is also asserted, this signal causes the PHY to transmit one or more symbols that are not part of the valid data or delimiter set somewhere in the frame being transmitted to indicate that there has been a transmit coding error.
- PHYA_RESET (std_logic)


**Figure 6.3** Controller component

## 6.2 Arbiter

In case we have more than one cores attached to the driver, we need to manage the write-back priorities of the cores. For this reason we have developed a controller. This controller reads the write-back requests of each core and if it is its turn to write back then it stores its outputs to the driver's Tx buffer. The only difference compared to the implementation described above is that in this case we need one more signal for the communication of the driver with the cores. This signal indicates when the core should start sending data back or if it must wait. The Tx buffer of the driver has also 32 bits long words.

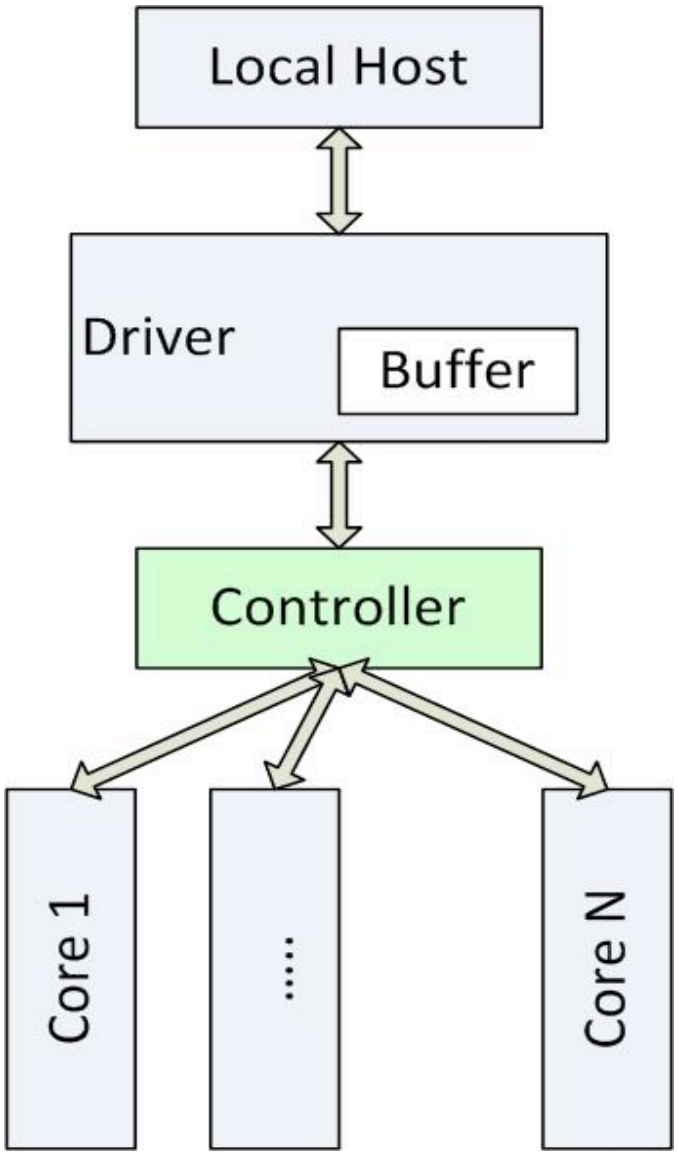



**Figure 6.3** Arbiter

# Chapter 7

# Implementation Results

The results of the implementation methods discussed in the previous chapters are displayed here. In our charts and tables we have also included the results of the C/C++ implementation of both Linear Time Selection Algorithm and Quicksort Algorithm. Thus, we compare the performance of the method we chose to implement with another common method(quickselect which is also another popular solution is based on the quicksort algorithm, so we can also conclude about how *k* and *number of inputs* affect and this method's execution time) and in parallel we compare the software and hardware runtimes.

We have measured the performance for the following cases:

- Search for the minimum
- Search for the median
- Search for the 25th percentile

All input data are random numbers from 0 to 255 (8bits) and are created with the use of *rand()* function.

## 7.1 Performance Results

We have measured the cycles needed to complete the process for all the implementations mentioned before for a variety of different number of inputs.

The charts below show the increase of the cycles needed to complete the process as we send more data to the core. We have used both linear and log scale in order to present better their performance differences.

**Table 7.1** Results of the search for the minimum number.

| Number of Inputs | Cycles for quicksort (C/C++) | Cycles for selection algorithm (C/C++) | Cycles for selection algorithm (VHDL V1) | Cycles for selection algorithm (VHDL V2) |
|---|---|---|---|---|
| | | | | |
| **10** | 3200 | 5400 | 158 | 67 |
| **100** | 32000 | 26400 | 667 | 474 |
| **500** | 179200 | 111400 | 2990 | 1999 |
| **1000** | 231200 | 177600 | 5629 | 3700 |
| **5000** | 2175200 | 877600 | 26938 | 17358 |
| **10000** | 4809600 | 709600 | 54308 | 34803 |
| **50000** | 20918400 | 3279200 | 271641 | 173386 |
| **100000** | 68388800 | 8198400 | 535694 | 342327 |
| **200000** | 284828800 | 12285600 | 1068560 | 682837 |
| **500000** | 1615546400 | 32924000 | 2678545 | 1710784 |

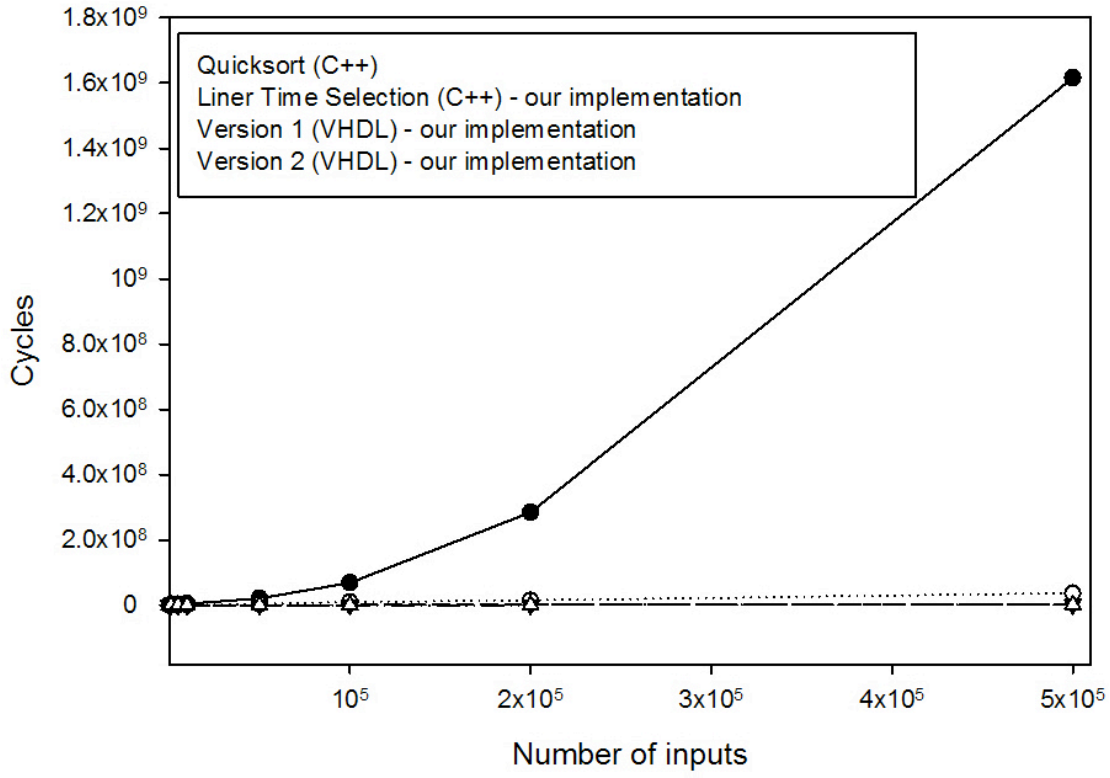



**Figure 7.1a** Results of the search for the minimum number. Linear scale.

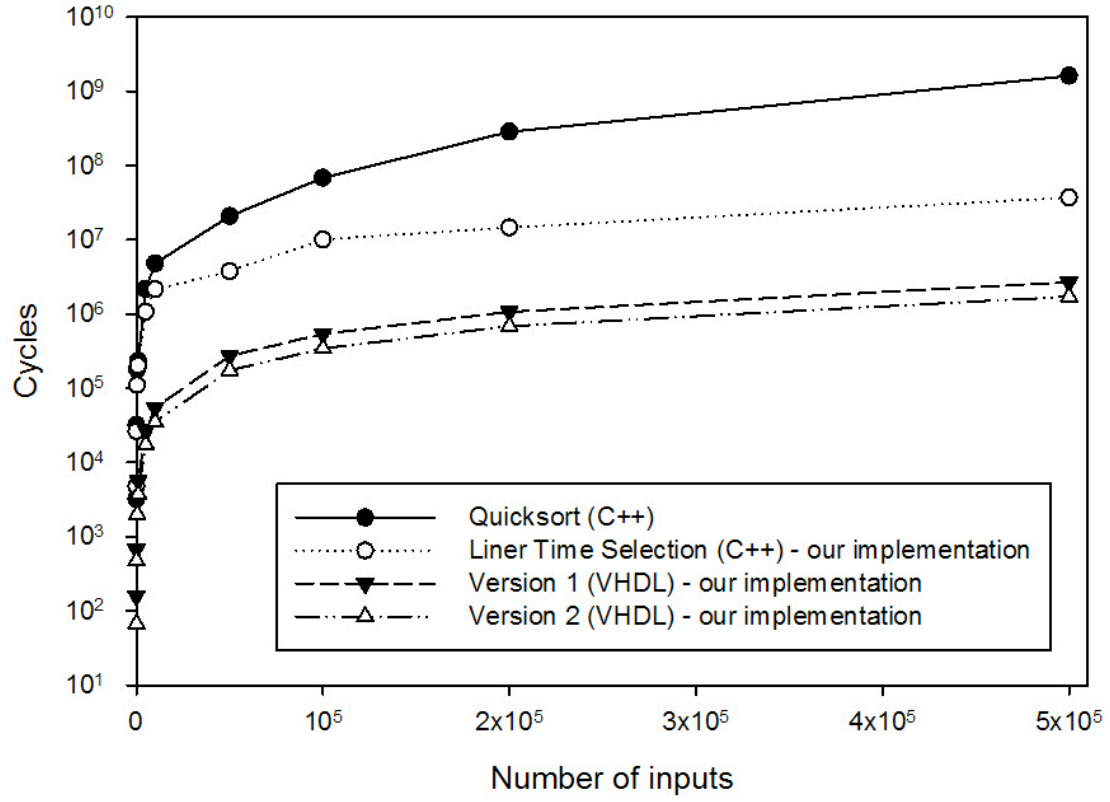



**Figure 7.1b** Results of the search for the minimum number. Log scale.

**Table 7.2** Results of the search for the median.

| Number of Inputs | Cycles for quicksort (C/C++) | Cycles for selection algorithm (C/C++) | Cycles for selection algorithm (VHDL V1) | Cycles for selection algorithm (VHDL V2) |
|---|---|---|---|---|
| | | | | |
| **10** | 4000 | 8600 | 193 | 93 |
| **100** | 32800 | 17400 | 370 | 269 |
| **500** | 179200 | 105600 | 2978 | 1968 |
| **1000** | 382400 | 262800 | 5776 | 3981 |
| **5000** | 2204800 | 969600 | 27749 | 17837 |
| **10000** | 4556800 | 1094400 | 32579 | 22581 |
| **50000** | 21837600 | 2676000 | 269096 | 171951 |
| **100000** | 80164800 | 6545600 | 540401 | 344946 |
| **200000** | 252299200 | 13148800 | 1082806 | 690750 |
| **500000** | 1451534400 | 36572800 | 2723650 | 1735245 |

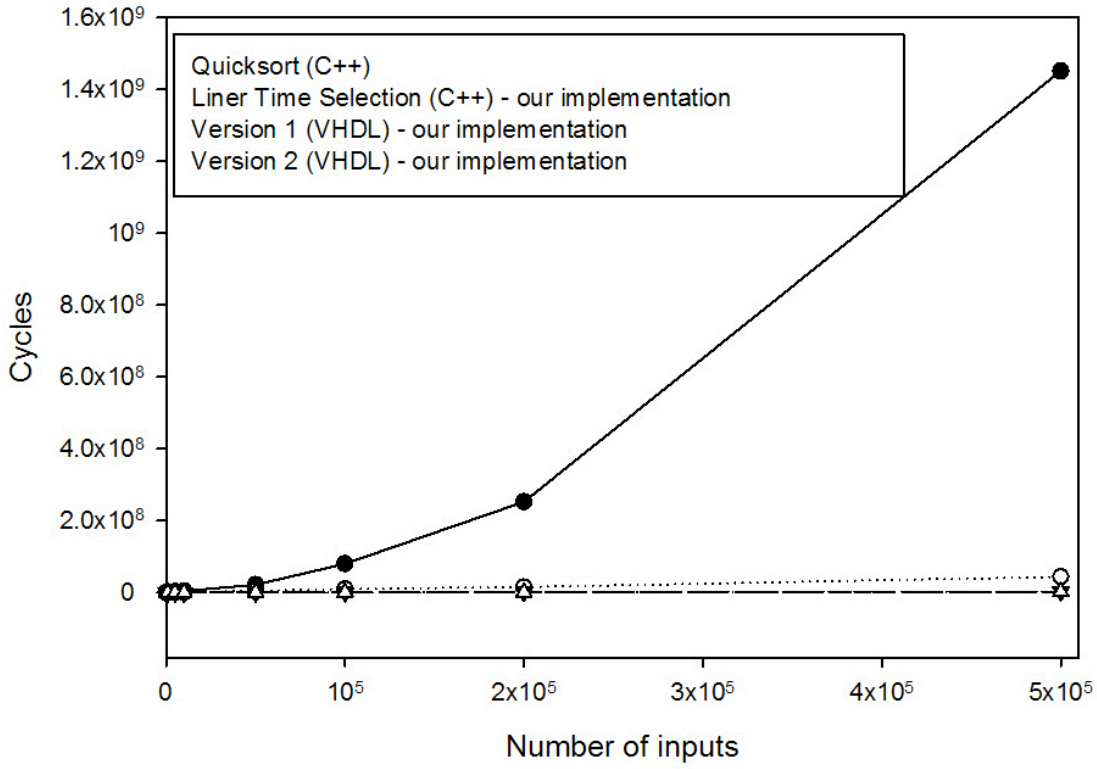



**Figure 7.2.a** Results of the search for the median number. Linear scale.

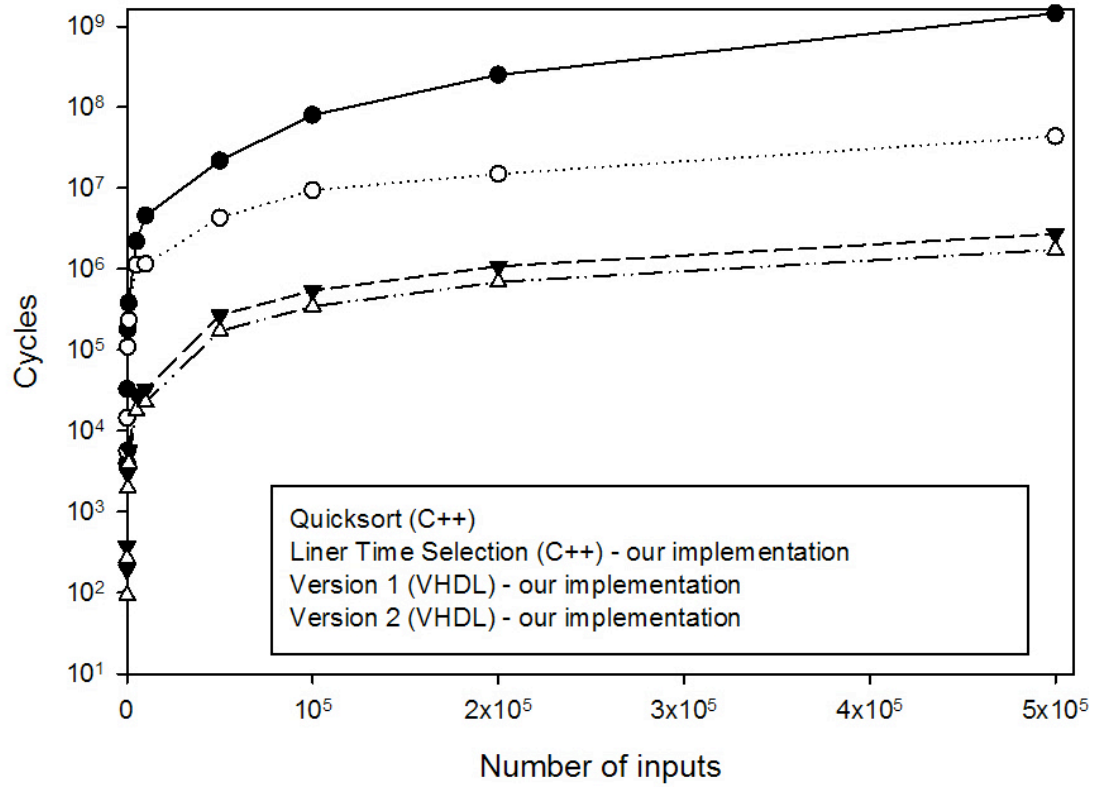



**Figure 7.2.b** Results of the search for the median number. Log scale

**Table 7.3** Results of the search for the 25th percentile.

| Number of Inputs | Cycles for quicksort (C/C++) | Cycles for selection algorithm (C/C++) | Cycles for selection algorithm (VHDL V1) | Cycles for selection algorithm (VHDL V2) |
|---|---|---|---|---|
| | | | | |
| **10** | 4000 | 6600 | 143 | 55 |
| **100** | 32800 | 32600 | 750 | 537 |
| **500** | 178400 | 113600 | 3089 | 2060 |
| **1000** | 283200 | 266500 | 5752 | 3762 |
| **5000** | 1755200 | 1038400 | 27188 | 17513 |
| **10000** | 5368000 | 1133600 | 55032 | 35226 |
| **50000** | 20691200 | 4512800 | 218085 | 143460 |
| **100000** | 68938400 | 7859200 | 540870 | 345211 |
| **200000** | 277820000 | 15831200 | 1087437 | 693274 |
| **500000** | 1458095200 | 40347600 | 2680974 | 1711191 |

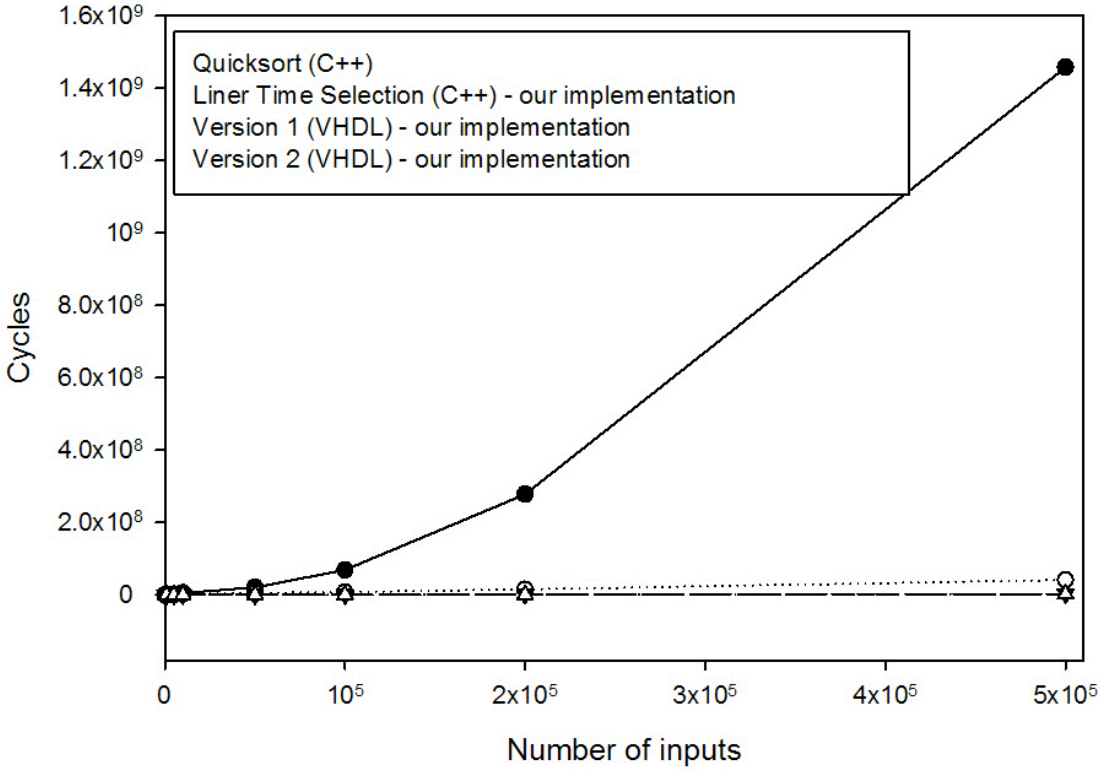



**Figure 7.3a** Results of the search for the 25th percentile. Linear scale.

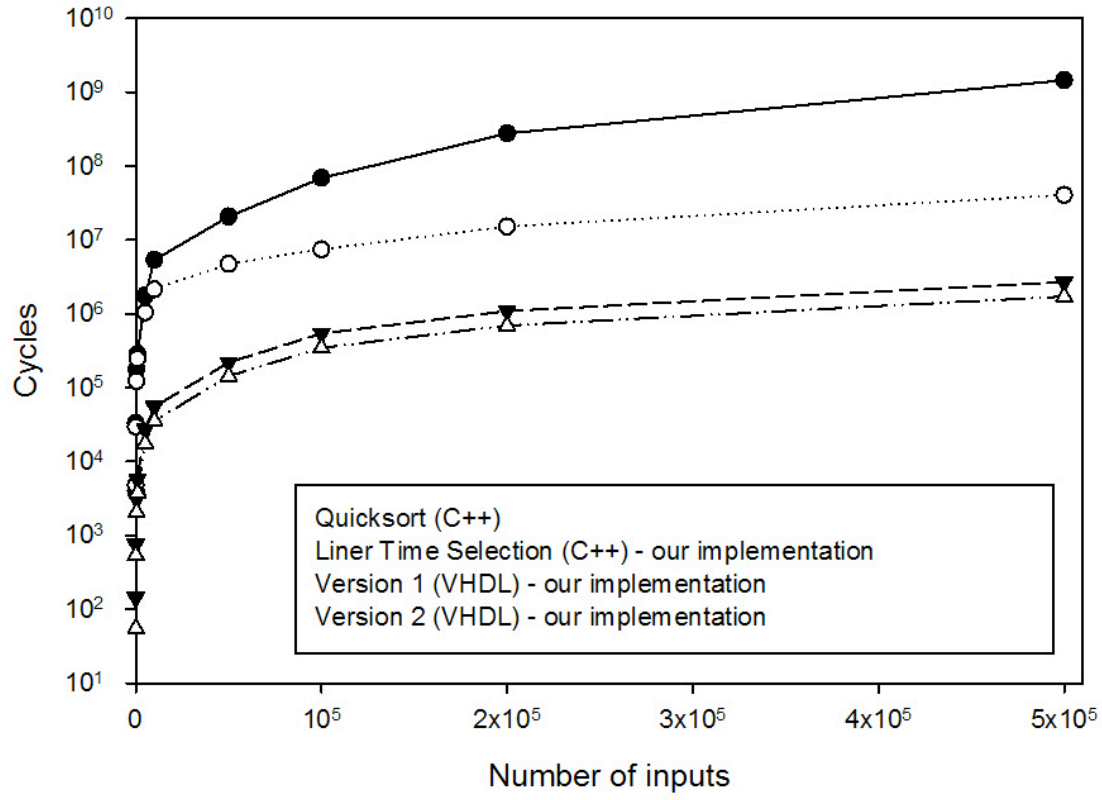



**Figure 7.3b** Results of the search for the 25th percentile. Log scale.

From the above Tables and Figures, we notice that for less than 10 inputs quicksort is faster than selection algorithm, while for more than 10 inputs the opposite happens. This is expected because the quicksort algorithm takes O(n logn) time to sort an unorded given set of numbers , while Selection finding algorithm takes 0(n) in worst case. Also because of this difference in their order times we notice that as the number of inputs increases, the performance gain with the use of Selection algorithm instead of quicksort is bigger.

Moreover, we notice that both VHDL implementations are much faster than the C/C++ implementations. The reason is that VHDL runs concurrently, while C/C++ runs sequentially.

The following figures show their differences:

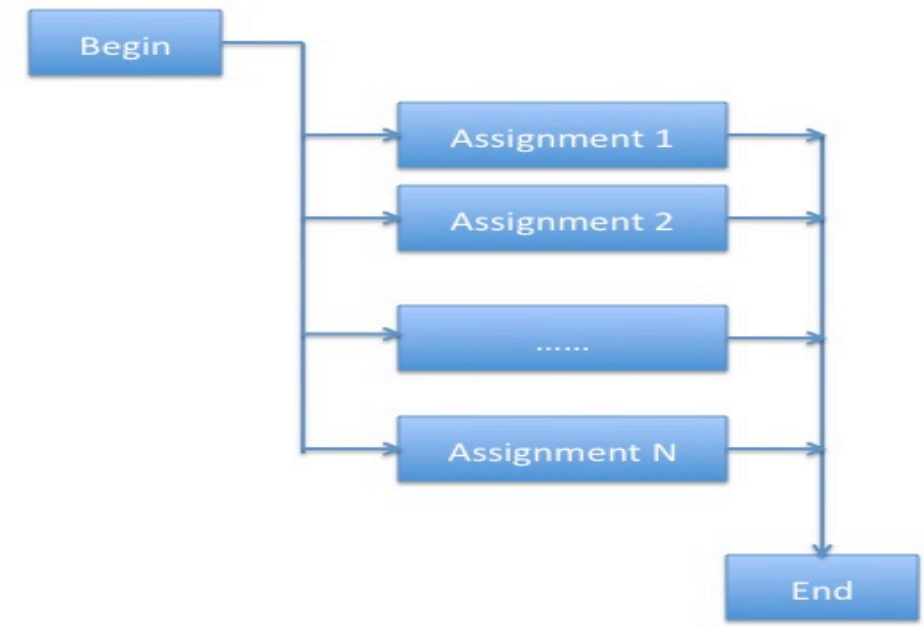



**Figure7.4** Concurrent

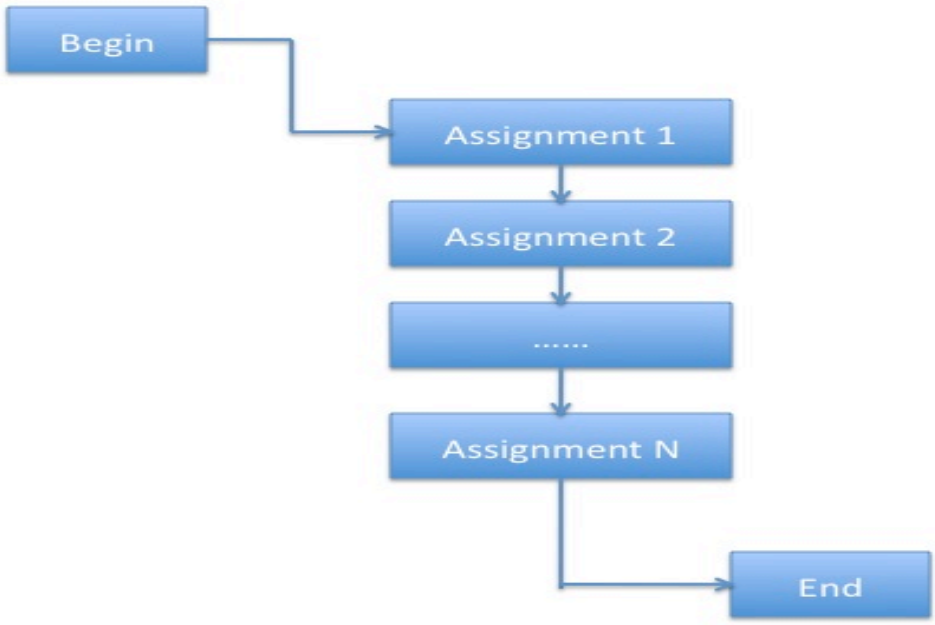



**Figure7.5** Sequential

Finally, we notice that Version 2 is faster than Version 1. As we have mentioned and in the previous chapter Version 2 is more time efficient than Version 1, as it is build with a concept that enables the parallel writing to different memories.

## 7.2 Power and Leakage Results

Although, in this thesis we have not focused on the management of power consumption on FPGAs, we have measured both total power and leakage power for all the above scenarios, as power has become a major concern for semiconductor vendors and customers For these measurements we have used the XPower Analyzer tool.

**Table 7.4** Leakage and total power results

| Number of Inputs | Leakage Power (W) | Total Power(W) |
|---|---|---|
| **10** | 2,88 | 2,979 |
| **100** | 2,88 | 2,989 |
| **500** | 2,88 | 3,001 |
| **1000** | 2,881 | 3,011 |
| **5000** | 2,881 | 3,008 |
| **10000** | 2,881 | 3,021 |
| **50000** | 2,891 | 3,252 |
| **100000** | 2,902 | 3,536 |
| **200000** | 2,903 | 3,556 |

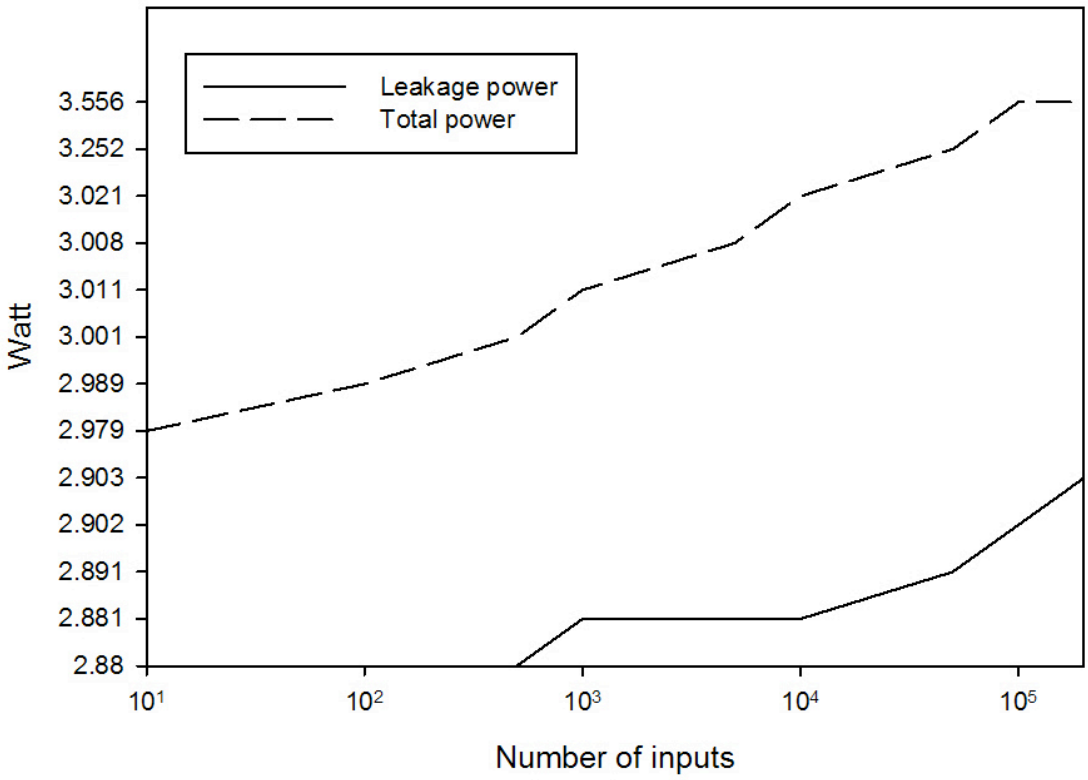



**Figure7.6** Leakage and power results

As it is expected Leakage power and the total power increases as the number of inputs becomes bigger. Larger designs suffer even more with these issues.

## 7.3 Implementation Reports

In the next table we present a summary about the device utilization for both Version 1 and Version 2.

**Table 7.5** Number of Slice LUTs, Number of Occupied Slices.

| Number of Inputs | Number of Slice LUTs | | Number of Occupied Slices | |
|---|---|---|---|---|
| | Version 1 | Version 2 | Version 1 | Version 2 |
| 100 | 1,358 (1%) | 1,239 (1%) | 627(1%) | 524(1%) |
| 500 | 1,579 (1%) | 1,654 (1%) | 625(1%) | 529(1%) |
| 1000 | 1,634 (1%) | 1,831 (1%) | 644(1%) | 589(1%) |
| 5000 | 1,864 (1%) | 2,14 (1%) | 726(1%) | 753(1%) |
| 10000 | 1,892 (1%) | 2,166 (1%) | 756(2%) | 780(2%) |
| 50000 | 2,065 (1%) | 2,727 (1%) | 850(2%) | 1,177(3%) |
| 100000 | 2,072 (1%) | 2,632 (1%) | 856(2%) | 1,095(2%) |
| 200000 | 2,162 (1%) | 2,343 (1%) | 933(2%) | 1,020(2%) |

**Table 7.6** Memory utilization

| Number of Inputs | Number of RAMB36E1 | | Number of RAMB18E1 | |
|---|---|---|---|---|
| | Version 1 | Version 2 | Version 1 | Version 2 |
| 100 | 0(0%) | 0(0%) | 8(1%) | 13(1%) |
| 500 | 0(0%) | 0(0%) | 8(1%) | 13(1%) |
| 1000 | 0(0%) | 0(0%) | 8(1%) | 13(1%) |
| 5000 | 0(0%) | 0(0%) | 14(1%) | 19(2%) |
| 10000 | 0(0%) | 0(0%) | 20(2%) | 25(3%) |
| 50000 | 24(5%) | 24(5%) | 52(6%) | 77(9%) |
| 100000 | 97(23%) | 122(29%) | 3(1%) | 3(1%) |
| 200000 | 197(47%) | 247(59%) | 0(0%) | 0(0%) |

We notice that Version 1 uses less memory sources than Version 2. This is expected because Version 1 uses 5 S_RAMEM memories, while Version 2 uses 10 of them.

The table below shows the maximum operation frequencies for both versions. If want to achieve ever higher frequency, we should use the pipeline option in the ISE Coregen, while we generate our Block Ram memories.

**Table 7.7** Maximum operation frequency

| Number of Inputs | Maximum operation Frequency (MHz) | |
|---|---|---|
| | Version 1 | Version 2 |
| 100 | 225 | 225 |
| 500 | 200 | 205 |
| 1000 | 199 | 223 |
| 5000 | 163 | 190 |
| 10000 | 170 | 189 |
| 50000 | 179 | 182 |
| 100000 | 174 | 194 |
| 200000 | 164 | 145 |

## 7.4 Conclusion

According to our measurements we conclude that the experimental results confirm the theory and our expectations. Linear time Selection algorithm is a much more efficient solution to the selection problem than the quicksort algorithm. Also with the FPGA implementation we achieve important performance improvement. As we have noticed and above with the 2nd Version the performance improvement is even bigger, while the 1st Version allows us to use tha FPGA implementation for larger sets of numbers, which is also very important due to inherent limitation about memory blocks found in FPGAs. To be more specific Version 1 presents 21% memory savings compared to Version 2, whilst Version 2 is up to 45 % faster.

Another conclusion is that Linear time Selection Algorithm needs approximately the same time to find any *kth* number from the given set of numbers, while Quicksort's (and so by extension Quickselect's) execution time is affected by the value of k and the number of input data(the pivot is chosen at random). This is shown and in the next figures.

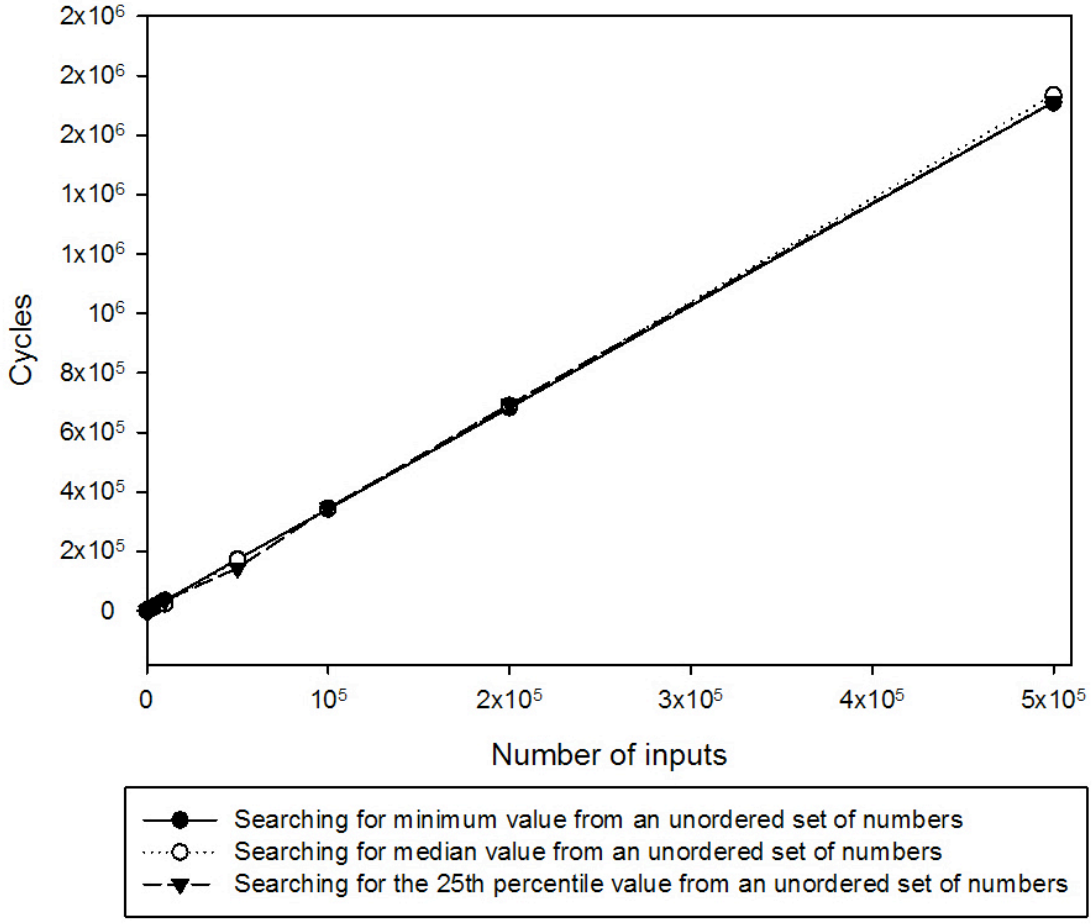



**Figure7.7** Comparison between the runtimes for the search of kth smallest number with the use of Linear time Selection Algorithm.

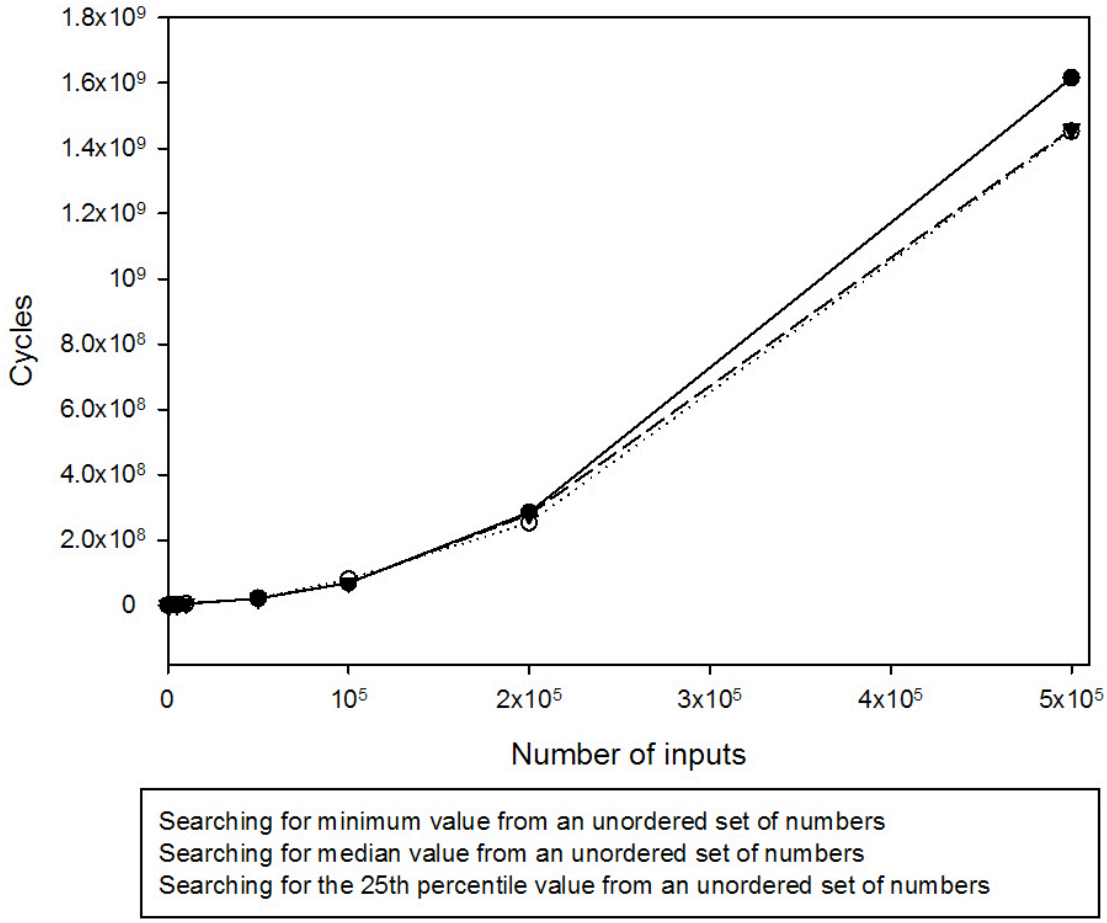



**Figure7.8** Comparison between the runtimes for the search of kth smallest number with the use of Quicksort Algorithm.

# References

[1] D.Richards, “VLSI median filters”, IEEE Trans. on Acoustics, Speech, and Signal Processing, vol. 38, 1990, pp.145-153.

[2] G.G. Boncelet, Jr., “Recursive algorithm and VLSI implementation for median filtering”, Proc. IEEE Int. Symp.on Circuits and Systems, 1988, pp. 1745-1747.

[3] A. Fisher, “Systolic algorithms for running order statistics in signal and image processing”, J. Digital Syst., vol. 4, 1982, pp.251-264.

[4] J.N.Hwang, J.M. Hong, “Systolic architectures for 2-D rank order filtering”, Proc. Int. Conf. Application Specic Array Processors, 1990, pp. 90-99

[5] S.Y. Kung, “VLSI Array processors”, Prentice Hall, 1989

[6] L. Chang, and J. Lin, "A bit-level systolic array for median filter", IEEE Trans. Signal Proc.., vol. 40, pp. 2079-2083, Aug. 1992

[7] C.L. Lee and C.W. Jen, "Bit-Sliced Median Filter Design based on Majority Gate," IEE Proceedings on Circ., Devices and Systems, vol. 1391, 1992, pp. 63-71.

[8] L.Lucke, K.Parhi, “A new VLSI architecture for rank-order and stack filter”, Proc. IEEE Int.Conf. Circuits and Systems, 1992.

[9] E.H. Lu, J.Y. Lee, and Y. Yang, "VLSI Architecture for Median Filters with Linear Complexity," TECNON '96, Proc. on Digital

[10] R. Roncella, R. Salemi, and P. Terreni, "70-MHz 2μm CMOS Bit- Level Systolic Array Median Filter," IEEE J.Solid-State Circ., vol. 28, no. 5, 1993, pp. 530-536.

[11] C-T.Chen, L-G.Chen, J-H.Hsiao, VLSI Implementation of a selective median filter, IEEE Trans.Consumer Electronics, vol.42, no.1, pp. 34-42, 1996.

[12] Oflazer, Design and implementation of a single chip 1-D median filter, IEEE

Trans.Acoust., Speech, Signal Proc., vol.ASSP-31, pp.1164-1168, 1983.

[13] M.Karaman, L.Onural, and A.Atalar, “Design and implementation of a general-purpose median filter unit in CMOS VLSI”, IEEE J. Solid State Circuits, vol.25, no.2. 505-513.

[14] L.Lucke, K.Parhi, “Parallel structures for rank order and stack filters”, IEEE Trans.Signal Proc., vol.42, pp.1178-1189, 1994.

[15] C.Lee, Hsieh, P. and Tsai, J., High-speed median filter designs using shiftable content-addressable memory. IEEE Trans. Circ. Syst. Video Technol. vol.4, no.6., 544-549., 1994 .

[16] G.R. Arce and P.J. Warter, "A median filter architecture suitable for VLSI implementation," Proc. the 23rd Annual Allerton Conf. Comm. Control Computing , 1984, pp. 172-181.

[17] C.Chakrabarti, “Sorting network based architectures for median filters”, IEEE Trans.Circ.Syst.II Analog Digital Signal Proc. vol.40, pp. 723-727, 1994

[18] L.Breveglieri, V.Piuri, “Digital Median Filters”, J.VLSI Signal Proc., vol.31, no..3, pp.191-206, 2002.

[19] V.V.Teja, K.C.Ray, I.Chakrabarti, A.S.Dhar, “High throughput VLSI architecture for one dimensional median filter,” Int.Conf. Signal Processing, Comm. and Networking, 2008, pp.339 - 344

[20] P.D.Wendt, E.J.Coyle, and N.C.Gallager, “Stack filters”, IEEE Trans. Signal Proc., 1986

[21] K.Chen, Bit-serial realizations of a class of non-linear filters based on positive boolean functions, IEEE Trans.Circ.Syst.,vol.36,no.6,pp.785- 794, 1989

[22] L.W.Chang and S.S.Yu, “A new implementation of generalized order statistic filter by threshold decomposition”, IEEE Trans. Signal Proc., vol.40, no.12, pp.3062-3066, Dec.1992.

[23] A.Gasteratos, I.Andreadis, P.Tsalides, “Realization of rank-order filters based on magority gate”, Pattern Recognition, vol.30, no.9, pp.1571-1576, 1997.

[24] I.Hatirnaz, F.K.Gurkaynak, and Y.Leblebici, “A compact modular architecture for the realization of high-speed binary sorting engines based on rank ordering”, Proc. IEEE Int.Conf. Circuits and Systems, 2000, vol.4, pp.685-688

[25] A Hiasat, O.Hasan, “Bit-serial architecture for rank order and stack filters”, Integration, the VLSI Journal, vol.36 , is.1-2, pp.3-12, 2003.

[26] V.A. Pedroni, “Compact Hamming-comparator-based rank order filter for digital VLSI and FPGA implementations”, IEEE Int. Symp.on Circuits and Systems, vol. 2, 2004, pp. 585-588.

[27] N.Weste, K.Eshragian, “Principle of CMOS VLSI design: a system perspective”, Addison-Wesley, 2 edition, 1993.

[28] Vasily G. Moshnyaga and Koji Hashimoto, ”An Efficient Implementation of 1-D Median Filter”, Circuits and Systems, 2009. MWSCAS '09. 52nd IEEE International Midwest Symposium

[29] D. Dhanasekaran, Akilesh Krishnamurthy, J. Ramkumar, “High speed pipeline architecture for adaptive median filter”, ICAC ‘09Proceedings of the International Conference on Advances in Computing, Communication and Control, Pages 597-600

[30] http://www.xilinx.com/products/silicon-devices/fpga/virtex-6/index.htm

[31] Xiaohui Wang , “A Selection Problem for Management Based on Divide and Conquer Algorithm”, Control, Automation and Systems Engineering, 2009. CASE 2009. IITA International Conference

[32] Dietz, P.F., “Very fast optimal parallel algorithms for heap construction”, Parallel and Distributed Processing, 1994. Proceedings. Sixth IEEE Symposium

[33] Arjun Saraswat, Nishant Kapoor, “Median finding Algorithm”