ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ

ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ

ΥΠΟΛΟΓΙΣΤΩΝ

Υλοποίηση ρομποτικών αλγορίθμων όρασης σε FPGA: Εφαρμογή στον αλγόριθμο landmark matching

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Χαραλαμπίδης Ιγνάτιος

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

Υλοποίηση ρομποτικών αλγορίθμων όρασης σε FPGA: Εφαρμογή στον αλγόριθμο landmark matching

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Χαραλαμπίδης Ιγνάτιος

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

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 18η Ιουλίου 2012.

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

Αθήνα, Ιούλιος 2012

.....

Χαραλαμπίδης Ιγνάτιος

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

Abstract

Computer Vision algorithms introduce mentionable computational complexity, which is usually non-sufficiently implemented onto general-purpose CPUs. Thus, it is common to employ specialized hardware accelerators which aim to improve the performance of critical kernels of these algorithms.

The goal of this diploma thesis is to provide a sufficient hardware/software co-design implementation of landmark matching algorithm onto a reconfigurable platform. More specifically, the timing critical kernels, as they were already derived from profiling procedure, was developed at reusable VHDL and successfully mapped onto the target FPGA (Virtex 6- XC6VLX240T). By exploiting as much as possible the inherent parallelism found in this algorithm, in conjunction to a number of design techniques, lead to the maximum gains. Regarding the non-timing critical kernels of landmark matching, they continue to be executed onto a general-purpose CPU, since they do not affect the performance of entire system.

The thesis is organized as follows:

In Chapter 1, there is an introduction in FPGAs and in computer vision. Basic parts of each are being mentioned, while there is a special reference at their relationship. Chapter 2 gives the related work on landmark matching algorithm. Moreover, this chapter also highlights the motivation of this diploma thesis. The implementation of landmark matching core is discussed in Chapter 3. More specifically, first of all we provide a example about how this algorithm is executed, whereas then there is a detailed description about its architecture, as well as its implementation. Chapter 4 covers the integration and communication between software (running onto a general-purpose CPU) and hardware (executed onto Xilinx Virtex-6 XC6VLX240T device). The corresponding modules of driver, controller and C program used for the communication are being explained. Experimental results that prove the effectiveness of introduced hardware/software co-design are discussed in in Chapter 5. For shake of completeness, the results of our implementation are also compared against to relevant implementations found in literature (e.g. the C/C++

implementation of the same algorithm). Chapter 6 summarizes the work performed during this diploma thesis and provides some potential directions about upcoming research in this topic. Finally, there is an appendix which provides detailed and useful information to interest readers about how to compile the project of Landmark Matching into Xilinx ISE framework, as well as details about the employed script for performing the Ethernet-based communication between PC and FPGA.

Keywords: FPGA, Computer Vision, Landmark Matching, Virtex5, Virtex6, HW/SW co-design

Περίληψη

Οι αλγόριθμοι της Όρασης Υπολογιστών εισάγουν αξιοσημείωτη υπολογιστική πολυπλοκότητα, η οποία συνήθως υλοποιείται μη επαρκώς σε ΚΜΕ γενικού σκοπού. Για αυτό τον σκοπό είναι σύνηθες να χρησιμοποιούμε υλικό ειδικού σκοπού για να επιταχύνουμε την απόδοση των κρίσιμων κομματιών αυτών των αλγορίθμων.

Ο σκοπός αυτής της διπλωματικής είναι να προτείνει και να παρουσιάσει μία ολοκληρωμένη υλοποίηση ενός τέτοιου συστήματος συνεργασίας μεταξύ software και hardware για τον αλγόριθμο landmark matching. Η υλοποίηση θα γίνει σε επαναδιαμορφώσιμη πλατφόρμα. Ειδικότερα, έχοντας βρει τα κρίσιμα κομμάτια του αλγόριθμου, χρησιμοποιήσαμε επαναχρησιμοποιήσιμη VHDL και απεικονίστηκαν επιτυχώς σε ένα FPGA (Virtex 6- XC6VLX240T0). Εκμεταλλευόμενοι όσο το δυνατόν περισσότερο τον παραλληλισμό που υπάρχει στον αλγόριθμο, σε συνδυασμό με ένα αριθμό τεχνικών σχεδίασης, οδηγηθήκαμε στο μέγιστο δυνατό κέρδος. Τα μη κρίσιμα κομμάτια του αλγόριθμο συνεχίζουν να εκτελούνται σε ΚΜΕ γενικού σκοπού, καθώς δεν επιβαρύνουν παραπάνω το συνολικό σύστημα.

Η διπλωματική οργανώνεται ως ακολούθως:

Στο κεφάλαιο 1, υπάρχει μια εισαγωγή στα FPPGA και στην όραση υπολογιστών. ο τα βασικά κομμάτια του καθενός, ενώ υπάρχει ειδική αναφορά στην μεταξύ τους σχέση. Στο κεφάλαιο 2, παρουσιάζουμε την σχετική δουλειά που υπάρχει ήδη πάνω στον αλγόριθμο landmark matching. Η υλοποίηση του πυρήνα παρουσιάζεται στο κεφάλαιο 3. Ειδικότερα. Ξεκινάμε με ένα παράδειγμα που παρουσιάζεται η βασική λειτουργία του αλγορίθμου και στην πορεία παρουσιάζουμε την αρχιτεκτονική του αλγορίθμου και την κυρίως υλοποίηση του, με την ανάλυση των βασικών συστατικών του. Στο κεφάλαιο 4, περιγράφουμε την υλοποίηση της επικοινωνίας μεταξύ του software (που τρέχει σε μία γενικού σκοπού ΚΜΕ)και του hardware (που τρέχει στο Xilinx Virtex-6 XC6VLX240T). Εξηγούνται αναλυτικά στο

κεφάλαιο αυτό τόσο το πρόγραμμα C που τρέχει στον υπολογιστή, το πρόγραμμα του ελεγκτή, και το κομμάτι του driver για την επικοινωνία τους.

Στην συνέχεια παρουσιάζονται τα πειραματικά αποτελέσματα που αποδεικνύουν την αποτελεσματικότητα από την εισαγωγή της επικοινωνίας μεταξύ software/ hardware. Για να αποδειχτεί η ορθότητα τους, τα αποτελέσματα συγκρίνονται με αυτά άλλων υλοποιήσεων και συγκρίνουμε τις επιδόσεις τους. Το κεφάλαιο 6 περιέχει την αποτίμηση των αποτελεσμάτων. Στο τέλος υπάρχει σχετικό παράρτημα με τα απαραίτητες πληροφορίες για την πλατφόρμα που χρησιμοποιήθηκε. Οδηγίες για το τρέξιμο τόσο σε hardware, όσο και για την επικοινωνία.

Πίνακας Περιεχομένων

Chapter 1: Introduction…… 11

1.1. FPGA…….... 11

1.2. Computer Vision……13

1.3. Computer Vision and FPGA.....16

Chapter 2: Related Work 17

2.1 Related Work.....17

2.2 Motivation.....19

Chapter 3: FPGA implementation 19

3.1 Algorithm description.....19

3.2 Architecture of design……....21

3.3 Core implementation …….25

Chapter 4: Communication 31

4.1 Algorithm description.....31

4.2 Architecture of design……32

4.3 Core implementation …….36

Chapter 5: Implementation results 40

5.1 TestCase 1.....41

5.2 TestCase 2.....48

5.3 TestCase 3.....50

5.4 TestCase 4.....51

5.5 TestCase 5.....53

5.6 Overall results.....53

Chapter 6: Conclusion 60

Chapter 7: Appendix 62

7.1 Hardware platform.....62

7.2 Synthesis and simulation process…….....71

7.3 Software execution process…….....73

Reference: 79

Chapter 1: Introduction

FPGA

Introduction

A field-programmable gate array (FPGA) is an integrated circuit designed to be configured by a customer or a designer after manufacturing—hence "field-programmable". As opposed to Application Specific Integrated Circuits (ASICs), where the device is custom built for the particular design, FPGAs can be programmed to the desired application or functionality requirements. Although One-Time Programmable (OTP) FPGAs are available, the dominant types are SRAM-based which can be reprogrammed as the design evolves. FPGAs allow designers to change their designs very late in the design cycle– even after the end product has been manufactured and deployed in the field. So, they are silicon chips which have the flexibility of software running on a processor-based system, but it is not limited by the number of processing cores available. What is remarkable for FPGAs is that they are parallel from nature so different processes do not compete for the same resources. Each independent processing task is assigned to a dedicated section of the chip, and can function autonomously without any influence from other logic blocks.

History of FPGA

The FPGA industry sprouted from programmable read-only memory (PROM) and programmable logic devices (PLDs). PROMs and PLDs both had the option of being programmed in batches in a factory or in the field (field programmable), however programmable logic was hard-wired between logic gates. In late 1980's the first programmable logical gates were implemented and some years later, in 1985, the first commercially viable field programmable gate array was invented by Ross Freeman and Bernard Vonderschmitt. In the not so distant past, FPGA were marketed for primarily two uses:

- ✓ For prototyping ASIC's
- ✓ For use in systems to achieve time to market

- ✓ Knowing they would be replaced with an ASIC implementation at the earliest opportunity.

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. We can assume that development of FPGA's can be divided into different eras: 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.

Architecture and Basic blocks of FPGA

Every FPGA chip is made up of a finite number of predefined resources with programmable interconnects to implement a reconfigurable digital circuit and I/O blocks to allow the circuit to access the outside world. FPGAs have evolved far beyond the basic capabilities present in their predecessors, and incorporate hard (ASIC type) blocks of commonly used functionality such as RAM, clock management, and DSP.

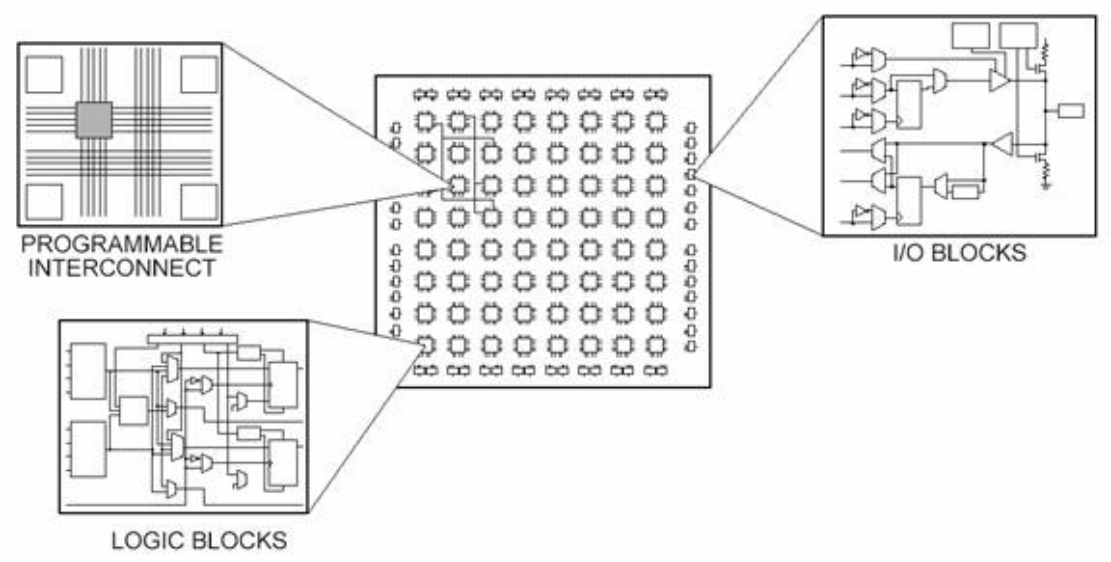


Figure 1.1: Different parts of an FPGA.

The organization of an FPGA architecture can be summarized as follows:

- *Configurable Logic Blocks (CLBs)*
- *Interconnect*
- *SelectIO (IOBs)*
- *Memory*
- *Complete Clock Management*

Configurable Logic Blocks (CLBs)

The CLB is the basic logic unit in a FPGA. Exact numbers and features vary from device to device, but every CLB consists of a configurable switch matrix with 4 or 6 inputs, some selection circuitry (MUX, etc), and flip-flops. The switch matrix is highly flexible and can be configured to handle combinatorial logic, shift registers or RAM. More architectural details can be found in the applicable device's data sheet.

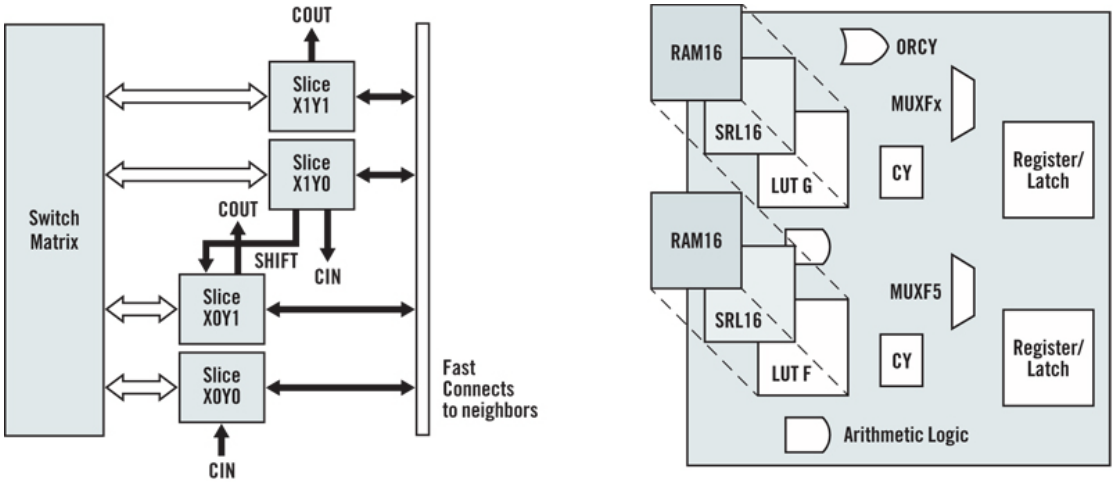


Figure 1.2: Basic Configurable Logic Block Structure

Interconnect

While the CLB provides the logic capability, flexible interconnect routing routes the signals between CLBs and to and from I/Os. Routing comes in several flavors, from that designed to interconnect between CLBs to fast horizontal and vertical long lines spanning the device to global low-skew routing for Clocking and other global signals. The design software makes the interconnect routing task hidden to the user unless specified otherwise, thus significantly reducing design complexity.

SelectIO (IOBs)

Today's FPGAs provide support for dozens of I/O standards thus providing the ideal interface bridge in your system. I/O in FPGAs is grouped in banks with each bank independently able to support different I/O standards. Today's leading FPGAs provide over a dozen I/O banks, thus allowing flexibility in I/O support.

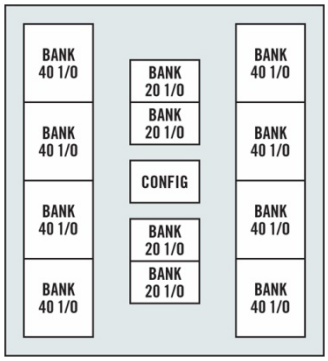


Figure 1.3: SelectIO Basic Block structure

Memory

Embedded Block RAM memory is available in most FPGAs, which allows for on-chip memory in your design. These allow for on-chip memory for your design.

Complete Clock Management

Digital clock management is provided by most FPGAs in the industry. The most advanced FPGAs offer both digital clock management and phase-looped locking that provide precision clock synthesis combined with jitter reduction and filtering.

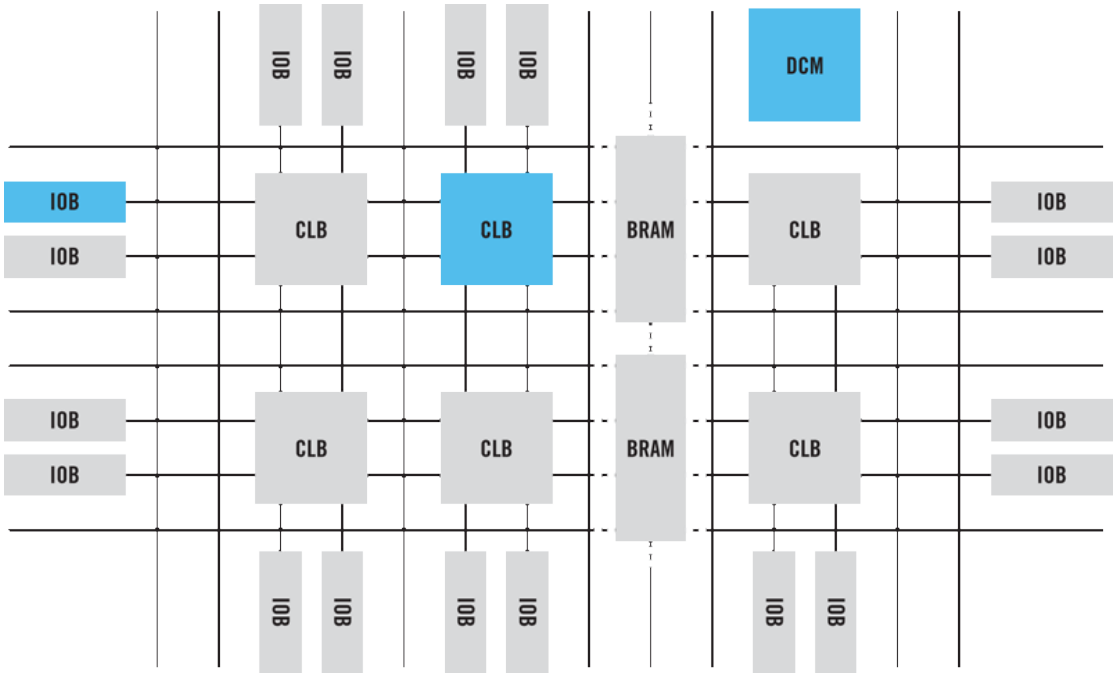


Figure 1.4: Overall FPGA block structure

Disadvantages of FPGA

Although FPGAs offer many advantages, there are naturally some disadvantages. They are slower than equivalent ASICs (Application Specific Integrated Circuit) or other equivalent ICs, and additionally they are more expensive. (However ASICs are very expensive to develop by comparison). This means that the choice of whether to use an FPGA based design should be made early in the design cycle and will depend on such items as whether the chip will need to be re-programmed, whether equivalent functionality can be obtained elsewhere, and of course the allowable cost. Sometimes

manufacturers may opt for an FPGA design for early product when bugs may still be found, and then use an ASIC when the design is fully stable.

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.

Computer Vision

Introduction.

Computer vision is a field that includes methods for acquiring, processing, analysing, and understanding images and, in general, high-dimensional data from the real world in order to produce numerical or symbolic information, *e.g.*, in the forms of decisions. A theme in 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.

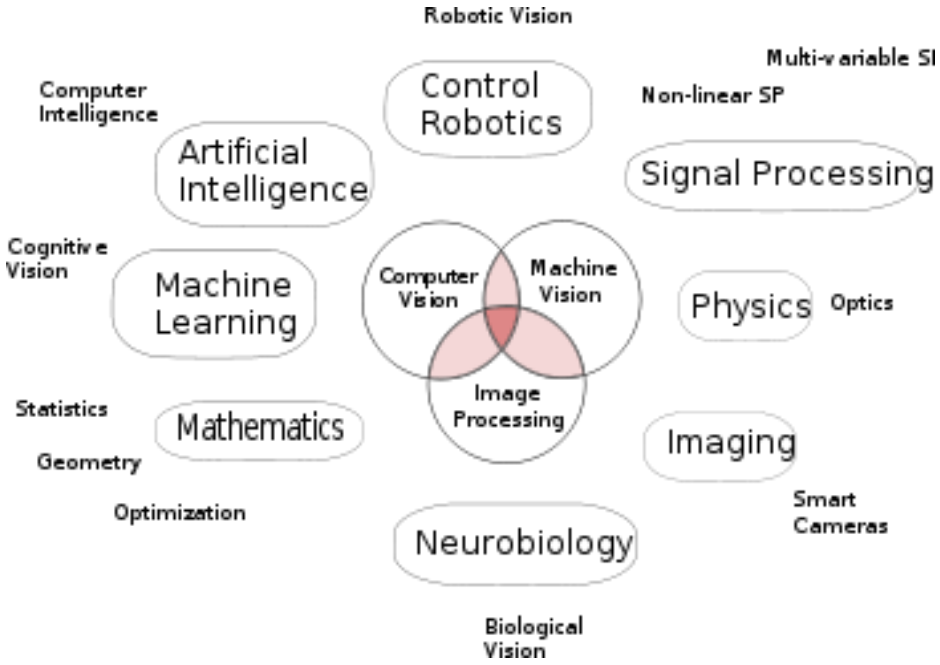


Figure 1.5 Applications of Computer Vision

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: Related Work

Related Work

Searching on the web, there were not many resources available regarding landmark matching algorithm either its software implementation or its hardware. Landmark matching algorithm is used as part of vision algorithms regarding localization. Landmark matching was found as part of the so-called LTRC algorithm: Landmark Matching, Triangulation, Reconstruction, and Comparison. Visual Image data has the potential to disambiguate objects for localization, as it provides high resolution, and additional information such as color, texture, and shape. To compensate for accumulated navigation errors, mobile robots must use external sensors to estimate their position. Active ranging devices give direct distance measurements and have found widespread use for robot localization. However, these sensors do not provide features needed to resolve ambiguities between objects. In order to understand how landmark matching works, we must first understand some aspects of localization algorithms. Global localization, provides the initial position estimate for conventional robot-tracking algorithms (e.g., extended Kalman filtering) and enables the robot to identify its own position when previous odometry readings are either inaccurate or even not available (e.g., due to wheel slippage, or just after powering up). In terms of functionality, localization can be classified as global, incremental, or simultaneous localization and mapping (SLAM). Global localization identifies the robot position with respect to some external frame using only the *current* sensory data. Unlike the incremental methods, an historical position estimate is not required. The global localization application is targeted not only because it is essential to many robot navigation systems, but its independence from historical position estimates also clarifies the evaluation of the proposed algorithm in the presence of non-unique landmarks. Localization methods are often classified either as iconic or feature-based. The iconic method directly compares the raw data with the map, whereas the feature-based method considers mainly the prominent features

A pair of landmarks is said to be similar if the difference between their individual signatures (median color scalars) is sufficiently small. Expecting a consecutive match

along the entire landmark sequence is not realistic, due to partial occlusion and slight environmental changes. On the other hand, non-unique landmarks are a common occurrence. To reduce the rate of mismatching, only landmark sequences with at least three consecutive matched features are considered. The landmark signature list of the current image is compared with that generated from each of the reference images. The operation often finds multiple sets of consecutive matched landmarks. Robot position estimates are obtained from the triangulation of these matched landmark sets, and the best one is selected during the reconstruction and comparison stage.

Motivation

As already been said, computer vision algorithms use many computer resources and they are considered to be time-consuming. Aiming at low-cost and efficiency, this diploma proposes the use of field-programmable gate array device (FPGA) . We describe the translation of computer vision algorithms to VHDL and detail the design of a working prototype. We present results showing that an FPGA device provides hardware speed to user applications, delivering real-time speeds for image segmentation at an affordable cost. An efficiency comparison is made among the hardware-implemented and a software-implemented (C language) system using the same algorithms.

Challenges for the diploma thesis were:

- The fact that there was not a single implantation of the specific algorithm on FPGA.
- We would like to present real results on the speed up of a project if specific operations are performed on a FPGA. For that reason we will compare our results with both Matlab and C results.
- In order to take advantage of HW/SW co-design, our goal is to provide a fully integrated alternative between an FPGA device with a host-PC in a way that PC can send data to our device just by calling a script. In this way, we take advantage of software\hardware co-design as we can call our module implemented on FPGA just like any other function in a high-level program.

Chapter 3: Implementation of Landmark Matching

The algorithm implemented in this diploma thesis is that of landmark matching. This algorithm was developed from scratch in reusable VHDL [25] and was successfully mapped onto Virtex-5 (XC5VLX50T) and Virtex-6 (XC6VLX240T) FPGA platforms. The design goals of this implementation are summarized as follows:

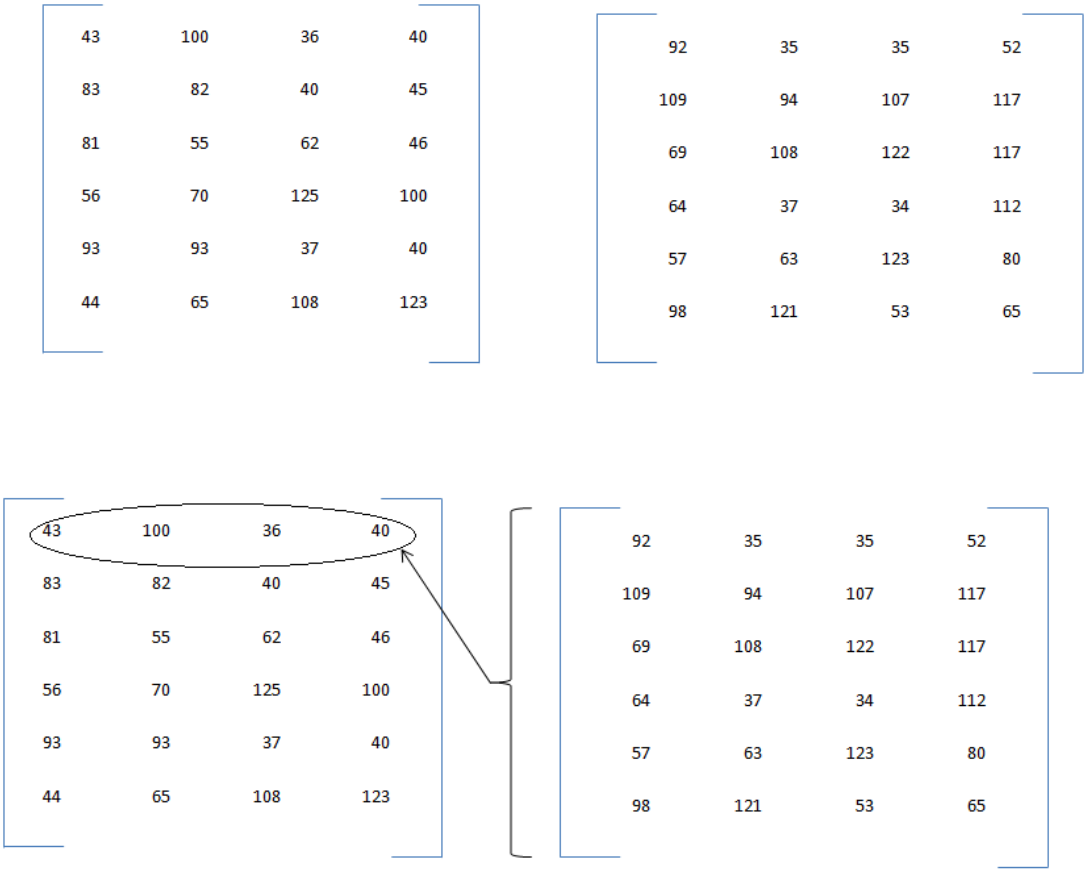
- ✓ High-Performance
- ✓ Low-Power
- ✓ Sufficient area utilization
- ✓ Integration through Ethernet protocol into a HW/SW system

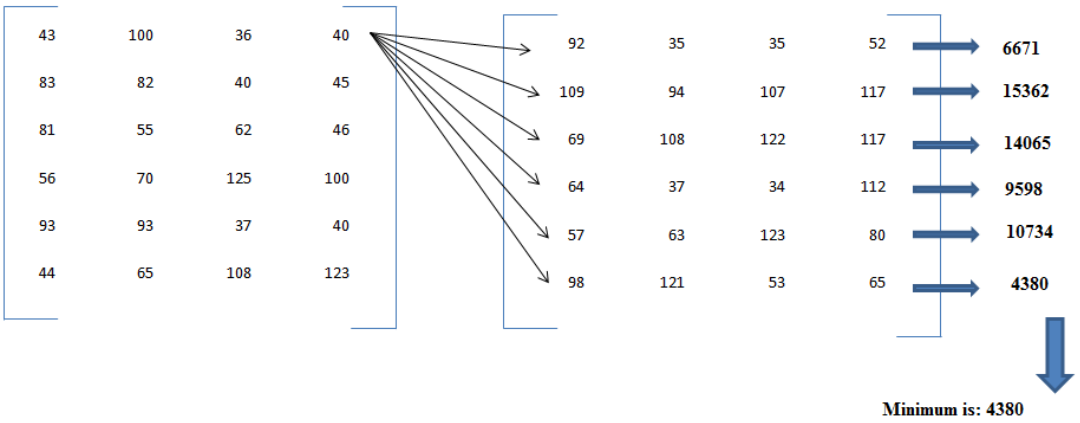
Algorithm Description

In this diploma thesis we are going to implement landmark matching algorithm. Landmark matching uses many different frames of a single image and tries to identify certain spots in it.

The core works like this: it takes as an input two arrays, each one consisting of some vectors. For every vector of array A, it computes and returns the minimum Euclidian distance between vectors of array B. This minimum distance is so-called *"match"* in Computer Vision. So, landmark matching finds the perfect matches.

An example of running our core is shown in the next figure:





Next, we describe in more details the architecture, as well as the implementation of Landmark Matching algorithm.

Architecture of the design

Landmark matching algorithm, as described in the previous sections, requires the simultaneous run of many processes, and for a considerable amount of times. As first approach, we tried to run all the required processed during the same cycle. Therefore we had a process responsible for subtracting the data, square them and added them to the previous sum. Such a design would drastically reduce the complexity of our design. But as a result of this, our clock was terrible and we were wasting so much time while we could run thing in parallel. For this reason, we used a different approach for our design and we used a form of pipeline. In the figure 1 there is a schematic example of our pipeline.

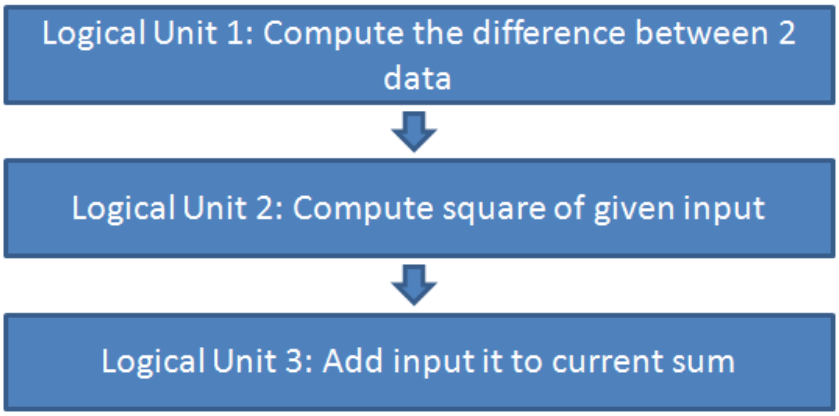


Figure 3.1: different stages and logical units that are pipelined.

The functionality of pipeline could be explained as follow:

- ✓ In cycle *X*, we compute the difference between our input data.
- ✓ In the next cycle, this difference is being forwarded to the next stage and the square difference is computed. In the previous stage, new data have been imported and the new difference is being computed.
- ✓ In cycle *X*+2, the square difference of cycle *X+1*, is being forwarded to next stage and we added to the previous sums. Like before, a new square difference and a new difference (that of our new data) are computed.

By doing this, we minimize our clock cycle because each logical unit performs an independent operation and will take only the time that it needs. Besides, there is no waiting for all the other logical units to finish their work. When each logical unit finishes its operation, it forwards the result to the next stage and waits for new data.

Figure 3.2 schematic example of our pipeline

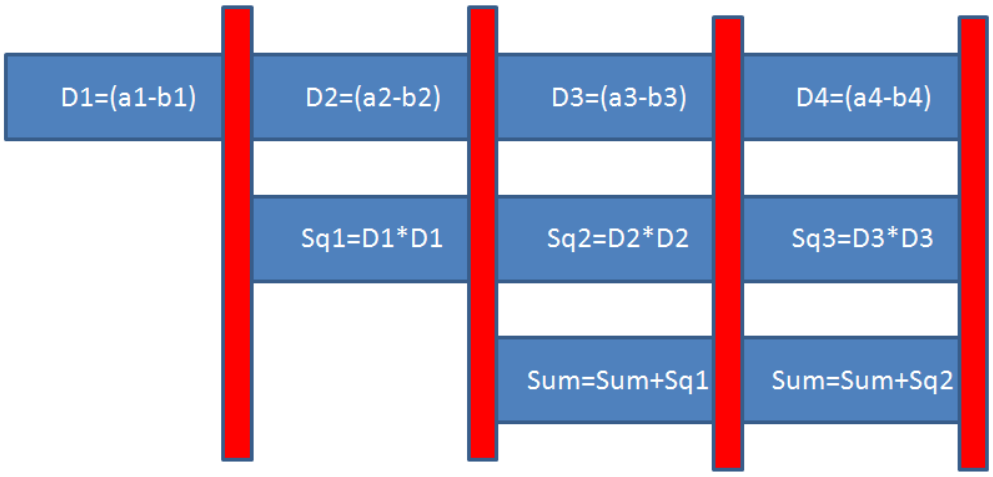


Figure 3.3 Schematic example of our pipeline.

Core implementation:

If we consider our core as a black box, we can see that it consists of the following inputs/outputs:

```
entity landmark_1 is
 generic
                (data_length :integer := 16;
                address_length:integer:=9 ;
                comp_num:integer:=64;
                total_number_A:integer:=13;
                total_number_B:integer:=9);
                port ( clk:in std_logic;
                rst:in std_logic;
                mem_data:in std_logic_vector(data_length-1 downto 0);
                set: in std_logic;
                mem_address: in std_logic_vector(address_length downto 0);
                wea: in std_logic;
                finished:in std_logic;
                new_result: out std_logic;
```

```
                dout: out std_logic_vector(4*data_length-1 downto 0);
                done: out std_logic
                );
end landmark_1;
```

Let's take a closer look at each one:

As far as the generic constants are concerned:

- ✓ *Data_length*: Our length data. Our architecture supports generic data. That means that in synthesis time, length of input data has to be decides. In our simulations we chose as length that of 8, 16 bits for specific reasons explained in the appropriate section.
- ✓ *Address_length*: At synthesis time, length of our addresses has to be declared.
- ✓ *Comp_num*: It is the number of each vector's coordinates. In our simulation was set to 64, although our architecture can support and arbitrary value.
- ✓ *Total_number_A, Total_number_B*: integers declaring the total number of vectors in array A and in array B of our input. They must also be set before synthesis.

Next, we analyze the input and output signals found in our developed core.

- ➢ *Clk:* Clock of our design
- ➢ *Rst*: Reset signal, used for initialization or setting the design back to is default value.
- ➢ *Mem_data*: Input data of data_length bits. It represents the data that are going to be stored in the local single port memory. Use of this signal is described in the following section.
- ➢ *Set*: Logical input used for identifying which array my data belong. When set to L data are elements of input array A, while when H data belong to array B.

- *Mem_address*: Signal of address_length bits, used for representing the address in which data are stored in the local memory.
- *Wea:* Write enable signal for the local memories.
- *Finished*: Logical signal, used as flag for computation process to start. When set to H, all data have been written in Ram and we can safely start our computation.
- *New_result*: Output logical signal. It is H only for a single cycle, when a new result has been computed. It is used from our controller in order to identify that a new result has arisen. That's why its use will be explained in details in the section of the controller.
- *Dout:* Our output signal. Whenever a result is available (a minimum Euclidian distance for each vector of array A), its value is assigned to dout.
- *Done*: Output logical signal. Used for knowing the end of the core process. When set to H, computation has finished and all data have been transferred to the upper level (that of controller) or the standard output.

What is worth mentioning is the fact that *dout* has been chosen to be of 4x data_length bits. We decided that for the following reasons:

1. First of all, we took in mind the worst case scenario. Given data of *data_length* bits, worst case result when subtracting in order not to lose accuracy because of overflow is to keep data_length+1 bits for my result. Later, this result is squared so in that worst case we need $2 * (data\_length + 1)$ bits. That result must be added to the previous sums, and the maximum number od adds that are going to be performed are *comp_num*. In all of our simulation *comp_num* was set to 64. So worst case scenario is perform 64 adding's of these results. So we need $2 * (data\_length + 1) + 6$ bits which gives us $2 * data - length + 8$ bits, which in case of 8 bits is $3 * data\_length$.

2. As it is clear from the beginning the basic goals of this diploma thesis was integration with the Ethernet. In order to use the Ethernet driver the limitations were the following:

   i) Input data must be of 8 bits.

   ii) Result written back in a character buffer must be of 32 bits.

Combining the above reasons, we concluded in the choice of $4 * data\_length\ bits$ which are exactly the bits I need for the simulations of 8-bit data.

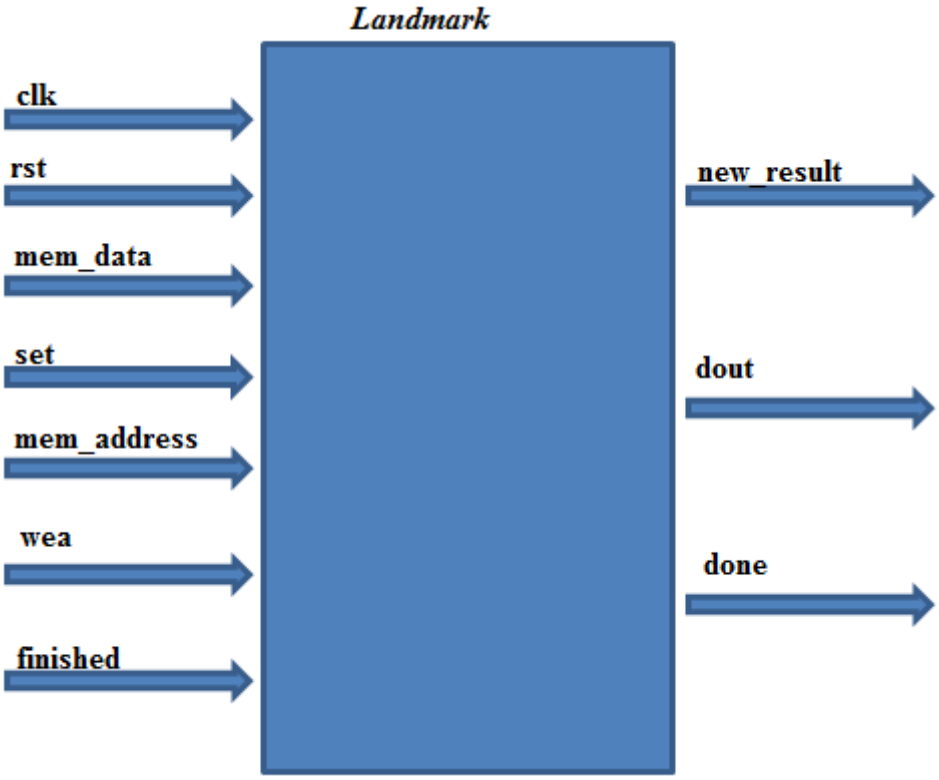


Figure 3.4 Input and Output signals of landmark module.

Next, we are going to analyze the basic components of our design. As already mentioned, we have used two (2) single port RAM memories, each one for each array. Input data are first saved in the memory, before out computation starts. Memory for vectors of array B is necessary because these vectors need to be recalled for every vector of array A. So it's much more efficient to fetch them directly from a local memory. Another approach in order to reduce the memory utilization would be not to store vectors of array A, because each one is used only once. When computation of its

Euclidian distance has finished, there is no need for using it again. In that case, our input has to been modified because we would be unable to read our data with the same order as provided for example at Matlab simulation. That's why we decided that it is more helpful to store these vectors in a memory despite the fact that they are not going to be used repetitively.

```
component ram_A IS
  PORT (
    clka : IN STD_LOGIC;
    wea : IN STD_LOGIC_VECTOR(0 DOWNTO 0);
    addra : IN STD_LOGIC_VECTOR(9 DOWNTO 0);
    dina : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
    douta : OUT STD_LOGIC_VECTOR(15 DOWNTO 0)
  );
END component;
component ram_B IS
  PORT (
    clka : IN STD_LOGIC;
    wea : IN STD_LOGIC_VECTOR(0 DOWNTO 0);
    addra : IN STD_LOGIC_VECTOR(9 DOWNTO 0);
    dina : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
    douta : OUT STD_LOGIC_VECTOR(15 DOWNTO 0)
  );
END component;
```

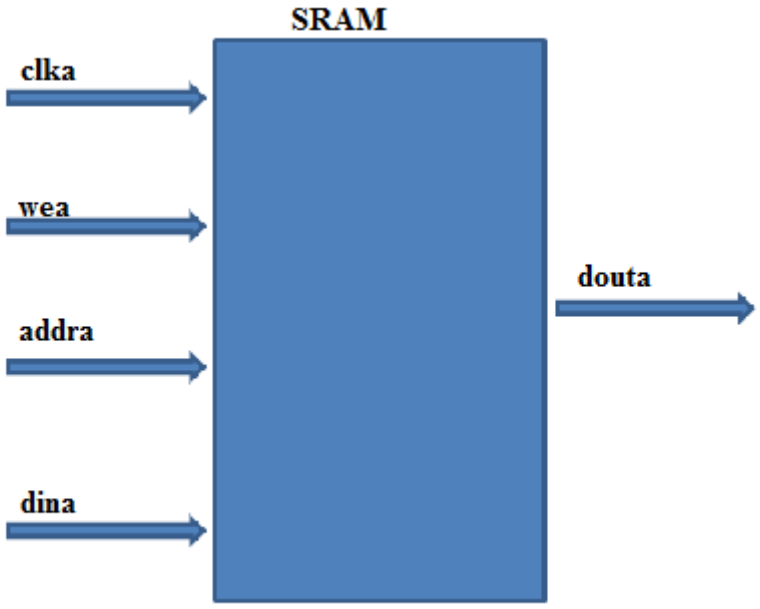


Figure 3.5 Input and Output signals of SRAMs.

To sum up, our core received data and stores them in 2 local memories. When finished reading, computation process can start. Our core was tested in several versions:

- Version 1: Input was provided by a file. At this version, the time measurement includes the reading/writing to RAM Blocks.
- Version 2: Input was loaded from a.coe file into ROM memories: In that case, there was no need for RAM, since they are replaced from two single-port ROM. This allows measuring the actual time required for algorithm execution (main process computation).
- Version 3: Input was provided using the Ethernet driver. Input data were read by a file, with the use of a high level language like C, and being passed to our core through a controller, which was also responsible for the write back process. At next chapter we explain in more details these 3 modules (controller, driver and C program).

Chapter 4: Integration

As already been said, the goal of his diploma thesis was not only the implementation of core but 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. First of all, we need a driver that will be able to pass our data to our core. But because these data are raw, between the driver and the core we have inserted a controller who receives these raw data, modifies them and transmits them to the core. This controller is also responsible for the write back process, as it receives the output of the core and transmits it to the driver.

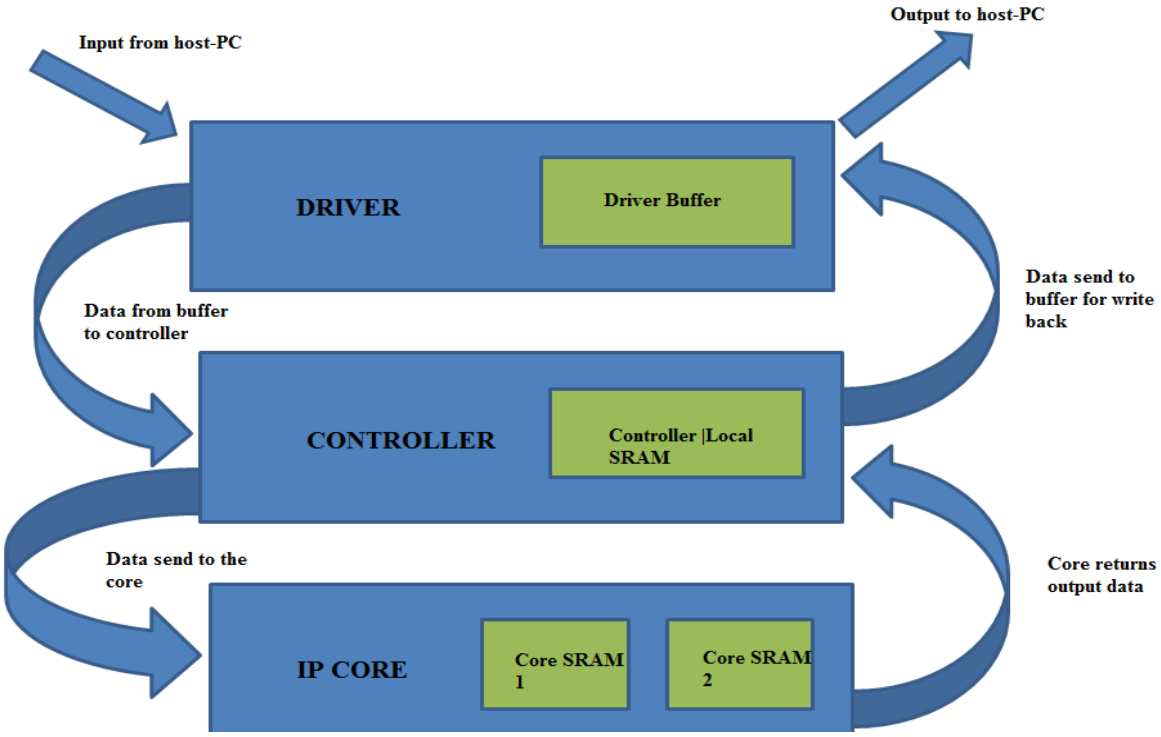


Figure 4.1 Communication diagram between modules

Controller Description

Component module as a black box has the following declaration.

```
entity controller is
        generic(
        data_length :integer := 8;
        address_length:integer:=7 ;
        comp_num:integer:=4;
        total_number:integer:=2);
        port (
        clk: in std_logic;
        rst: in std_logic;
        reading_state: in std_logic;
        din: in std_logic_vector (31 downto 0);
        dout_temp:out std_logic_vector (7 downto 0);
        core_address:out std_logic_vector(address_length downto 0);
        cntr_address: in std_logic_vector(7 downto 0);
        c_done: out std_logic;
        dout:out std_logic_vector (31 downto 0)
        );
end controller;
```

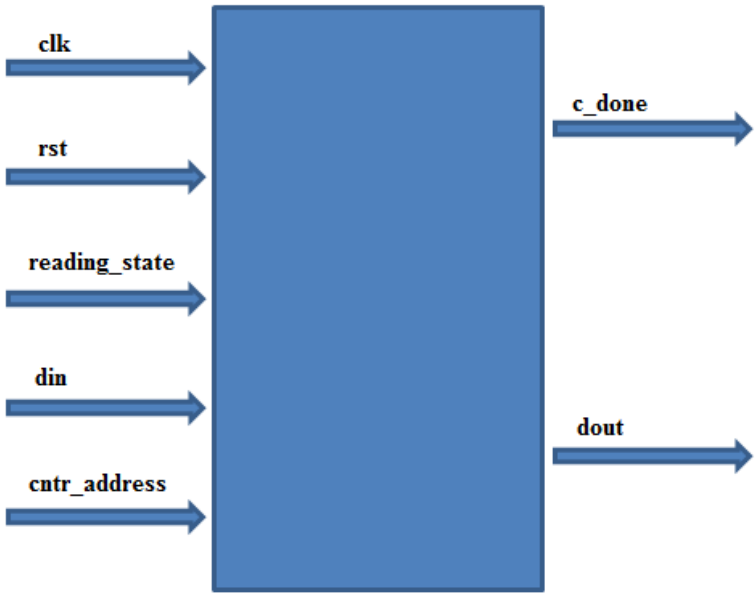


Figure 4.2 Controller module input/output.

Let's take a closer look at each of them.

As far as the generic constants are concerned:

- ✓ *Data_length, comp_num* as previous.
- ✓ *Address_length*: Natural number used for the representation of number of bits need for the 32 bits that controller reads from the Ethernet driver. Used for controller's local memory.
- ✓ *Total_number_A, total_number_B* as previous.

Inputs and output signals are the following:

- ➢ *Clk*: Clock of our design.
- ➢ *Rst*: reset signal, used for necessary initializations or for setting our design back to default.
- ➢ *Reading_state*: Std_logic signal. Signal stays L as long as controller receives data from the driver. When driver stops sending data, signal becomes H and controller will start next process: that of sending data to the core.
- ➢ *Din*: Input data received from a char buffer over the Ethernet driver. Data are of 32 bits length, as char buffer reads and transits 4 bytes.
- ➢ *Cntr_address*: Address of controller's local memory.
- ➢ *Core_address*: Address of our core memory.
- ➢ *C_done*: Output signal. Signal is set to H when controller has finished.
- ➢ *Dout*: used for sending data back to the driver during the write back process.

So our controller works like that: Controller receives data from a char buffer over the Ethernet driver. These data are saved in a controller's local single port RAM. When controller has received all the data, is ready for sending them to the core. So it fetches each 32bit element from the RAM, breaks it up to 4 separate 8 bit data, and send these

data to the core alongside with all the necessary information described previous (address of the current data, set of the data, and if there more data to be sent or not). For our controller design we used an FSM of 8 stages. In the first stage, the reading is performed while in stages 3 to 7 is the process of breaking the 32 bits into 4 unique. We decided to use a local memory for storing the data in order to be sure that synchronization problems will not arise. Since all data have been passed to our core, controller is in a wait stage and receives every output data of our core. This output also passes to the Ethernet driver in order to be written in a character buffer for transmition back to screen.

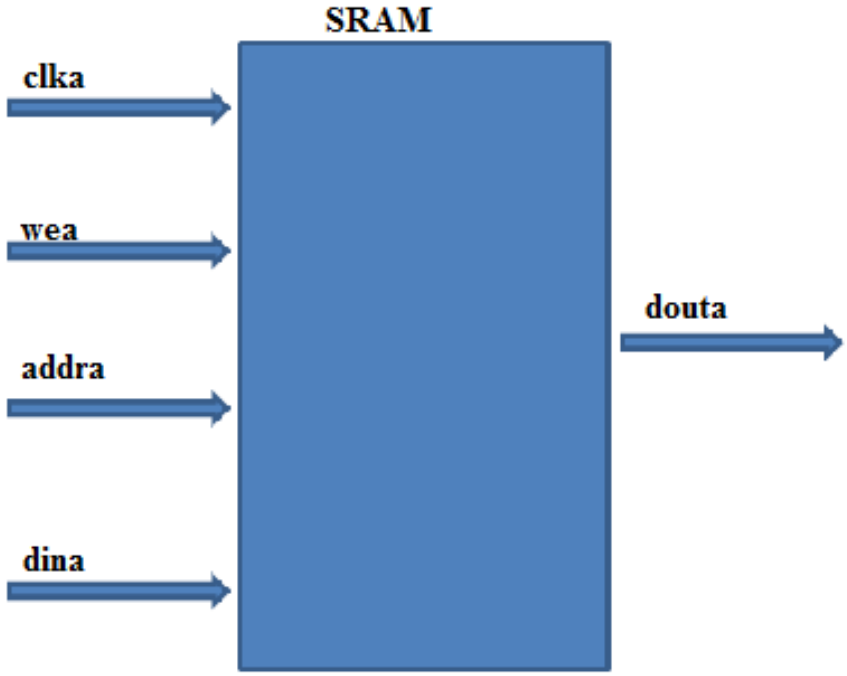


Figure 4.3 SRAM used in controller for storing 32 bits data

The FSM flow of the controller is shown in the next diagram:

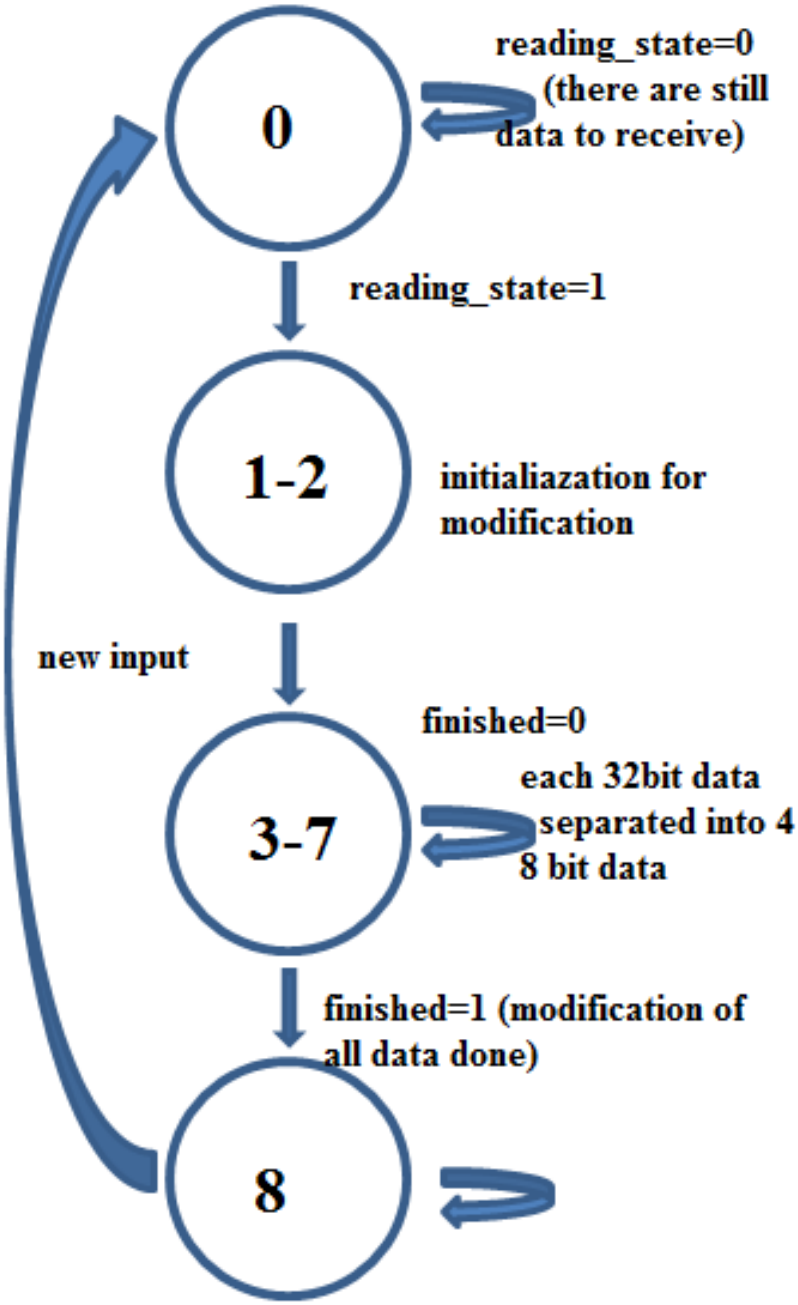


Figure 4.4 FSM flow of controller

Driver description:

The main goal of the driver is reading the data from the pc and sending them over Ethernet cable to our controller. Moreover, it is responsible for getting data from the controller and sends them to the pc 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 user only has to know a high level language. In this diploma thesis, program was written in C language. So, driver is responsible for reading data that are generated by the C program.

First of all, driver read 4 bytes containing the number of data that are going t be transferred. After that, driver read packages of 4 bytes until data are finished. Every 32bit data that is read is being transferred in the controller, for storing it in the local RAM as previously described. Driver uses 2 counter: an overall counter and an inner counter which is used like this: the driver uses a buffer to store data received over Ethernet. Driver's capacity is 1500 bytes. So, overall counter counts total bytes left, while inner counter counts how many bytes are left until the buffer of 1500 bytes is fully filled.

When the reading has 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.

Documentation of the driver is being presented right now.

Figure 4.5 Driver documentation

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) .

Chapter 5: Implementation results

In this chapter, we are going to demonstrate measurements taken for several inputs. In order to collect the inputs, we run several images obtained from link [1]. We run these images in matlab and we kept in a file the input traces for our function, which is landmark matching algorithm. After doing so, our inputs were refined. Since matlab has greatest accuracy with floats numbers we run several different instances of each input.

First of all, input was between -1 and 1. That means that it is safe for us to neglect the integer part, and only keep for input the fractional part. Because in VHDL the input must be declared and will be of standard length we had some limitations. When our input was set to 8 bits, our data should be between -128 and +127. That means that any number out of this range had to be adjusted in its closest limit. Similarly, when our input data was set to 16 bits length, the space has turned into -32767 and 32767.

Because of the above, we expect that there will be a slight deviation to our core results with the matlab results. We also expect that the longer the length will be, the less will be the deviation. In order to check that results of our core are correct, we also wrote a C program doing the exact same thing.

So we first run the project in matlab and we collected the results for the landmark. Then collected input was defined, and we run project for 8 bits, 16 bits both in C and VHDL. Demos for 8 bits were also tested with our integration project, and the use of controller and the Ethernet driver as described in the previous chapters.

So, at first, we are going to present measurements for our core.

Design properties of the devices used in our measurements are the following:

Family	Virtex5	Virtex6
Device	XC5VLX50T	XC6VLX240T
Package	FF1136	FF156
Speed	-2	-2

Core implementation results

Next we provide the experimental results about core implementation. For evaluation purposes, five different testcases are employed, whereas at the end there is a discussion about the overall power consumption.

TestCase 1:

At first we used for an input image, the following 4 frames.

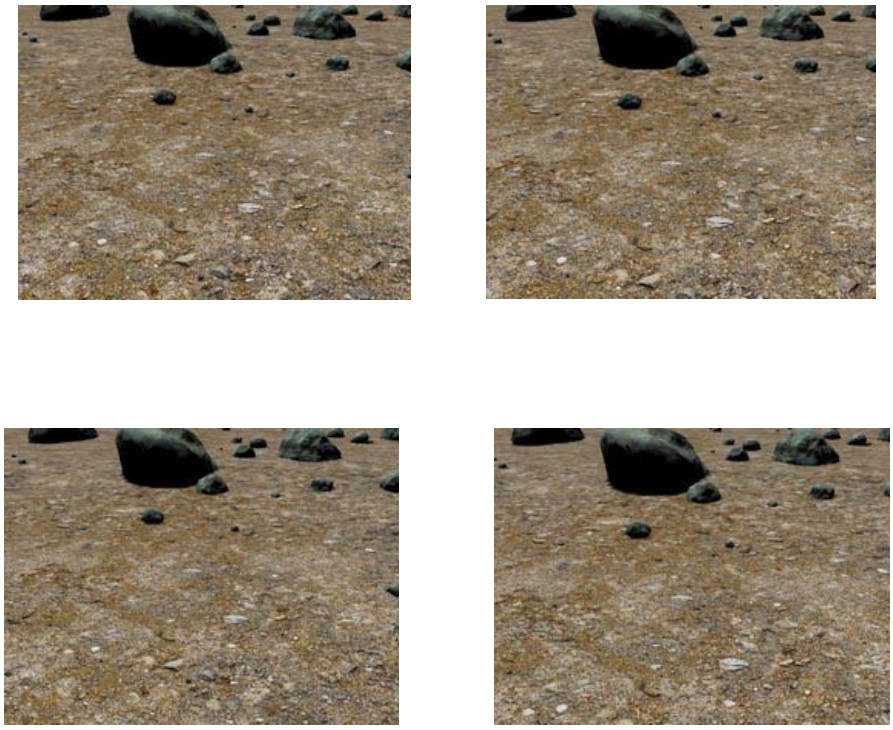


Figure 5.1: Input for the testcase 1 with image size 256x128.

Results for Image 1 from Matlab

Running the code in matlab gave us a result an array A with 13 vectors, each one consisting of 64 coordinates, while array B had 9 vectors, each one consisting of 64 coordinates. We had mentioned at the beginning, that the number of coordinates will always be 64 so in the next examples will not be mentioned at all. Matlab results are shown in the table below.

Table 5.1: Matlab results for testcase 1

Vector of Array A	result
Vector 1	0.151713
Vector 2	0.425982
Vector 3	0.156278
Vector 4	0.253199
Vector 5	0.286622
Vector 6	0.354446
Vector 7	0.009701
Vector 8	0.267147
Vector 9	0.197427
Vector 10	0.029481
Vector 11	0.011157
Vector 12	0.004622
Vector 13	0.006055

As it was expected, we got 13 outputs, and each one is the minimum Euclidian distance between specific vector of A and vectors of array B.

Afterwards, we readjusted the input data to 8 bits and we run the core in C and VHDL for the reformatted input.

VHDL Implementation with 8bit accuracy

Our core was run in VHDL, and screenshots of the execution are shown below.

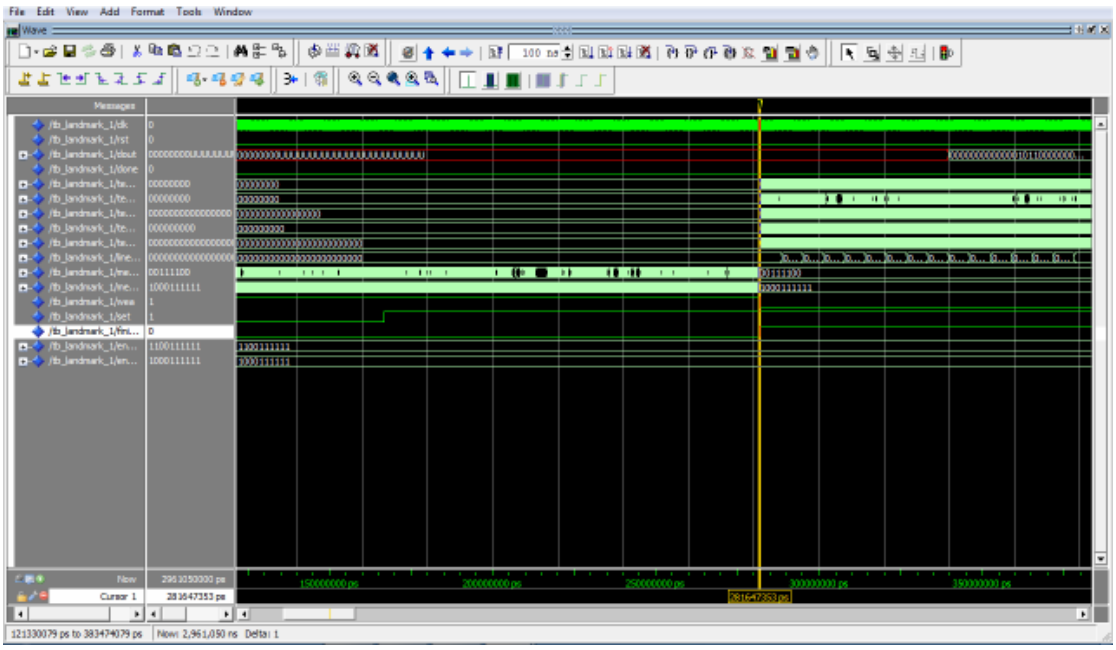


Figure 5.2 Reading ended in VHDL 8 bits

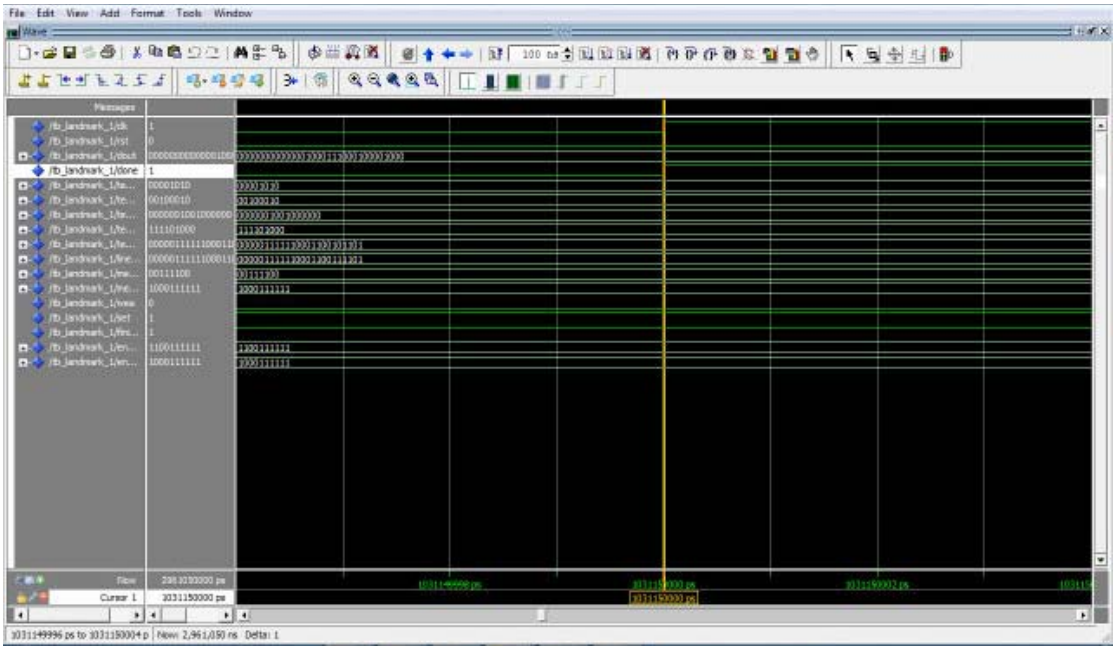


Figure 5.3 Output VHDL 8 bits.

We also run C program in order to check our results and we see that they are same. So, our core works perfectly. Results are being shown in the next table:

Table 5.2 Comparison results between C and VHDL

Vector	*Output from C*	*Output from VHDL*
Vector 1	360,667	360,667
Vector 2	423,350	423,350
Vector 3	282,327	282,327
Vector 4	403,470	403,470
Vector 5	501,065	501,065
Vector 6	449,097	449,097
Vector 7	365,093	365,093
Vector 8	310,860	310,860
Vector 9	449,698	449,698
Vector 10	462,896	462,896
Vector 11	366,795	366,795
Vector 12	352,639	352,639
Vector 13	291,080	291,080

VHDL Implementation with 16bit accuracy

As explained earlier, we run our core for inputs of length 16. Screenshots from Modelsim execution are shown below:

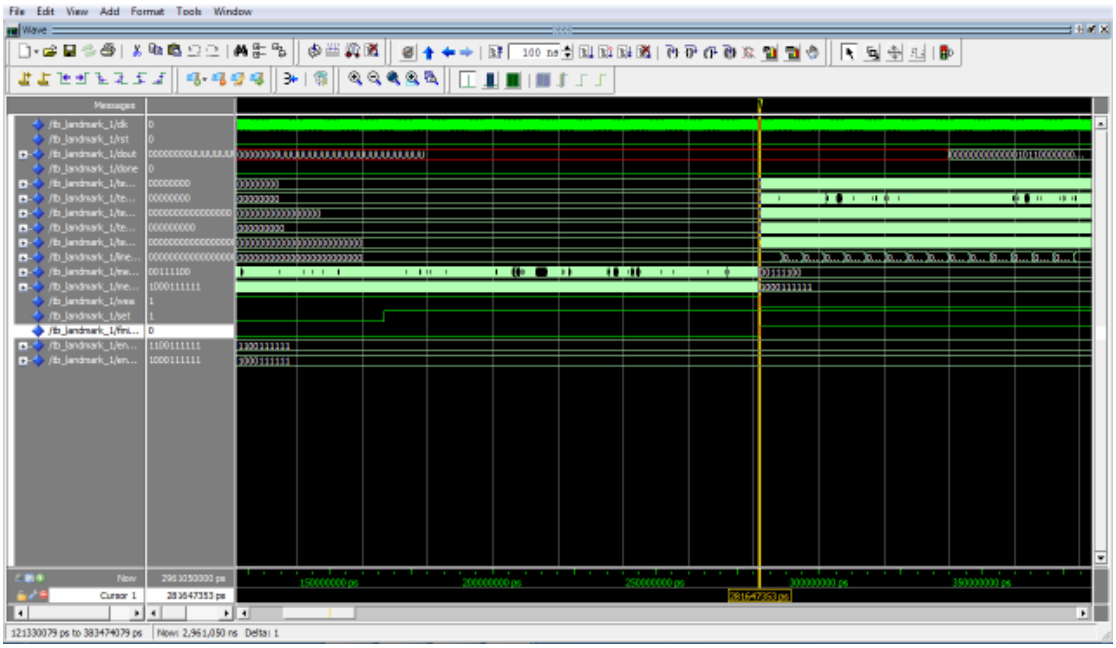


Figure 5.4 Reading 16 bits

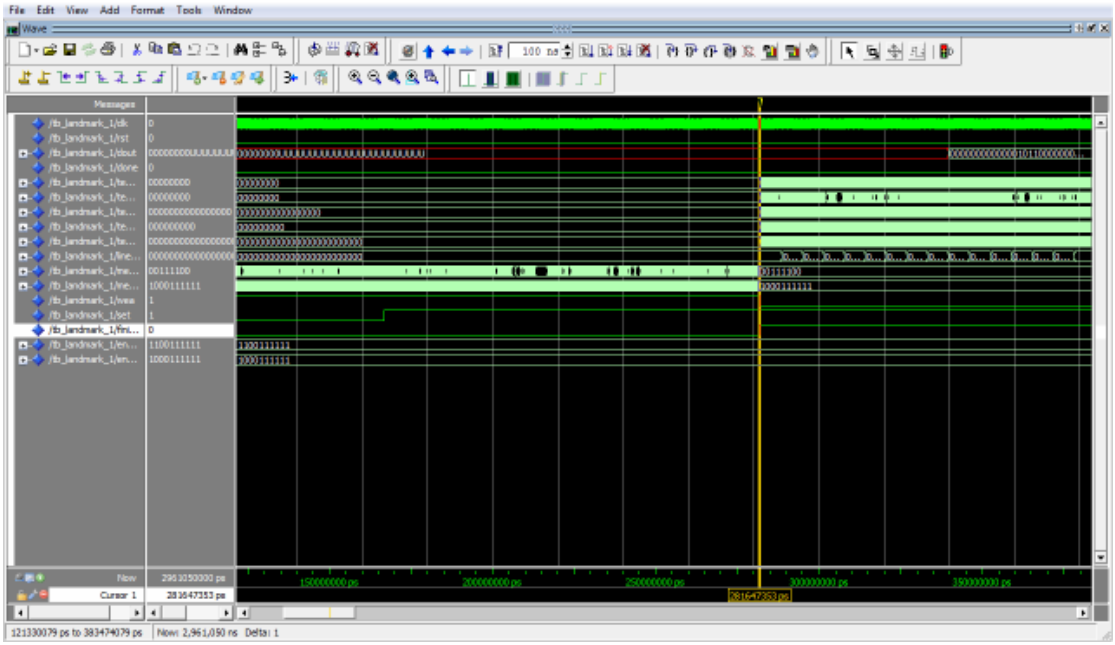


Figure 5.5 output 16 bits

As we can see from the above images, total time was the same in both bits. This is expected because they are not run with minimum clock cycle, but with a clock o period 10ns (or 100MHz). Moreover, we expect the number of cycles to be constant whatever the length is.

Results are shown in the table below.

Table 5.3: VHDL output for testcase 1, 32 bits

Vector	C output	VHDL output
Vector 1	42330408897	00111111000100111100010100000000001
Vector 2	6589236221	01100010001011111111000011111111101
Vector 3	5900855127	01010111111011011111100111010101111
Vector 4	6568030677	01100001110111110000110001110101010
Vector 5	14623368258	11011001111001111011100101010100000
Vector 6	12330778345	10110111101111100010111111011101001
Vector 7	4898325217	01001000111111011010000010111000001
Vector 8	7065353182	01101001010010000010111011111011110
Vector 9	15325579823	11100100010111100111001010001011111
Vector 10	5562675954	01010010111000111110110010111100010
Vector 11	4614745506	01000100110000111101101101101000010
Vector 12	5927199995	01011000010100100111100100111111011
Vector 13	5016801594	01001010110000011001010001001110010

As we can see, output in VHDL cannot be shown in decimal form, because it exceeds 32bits which is the integer range. But if we convert it to integer with a calculator we can verify that results are correct.

After post-time synthesis (see complete steps and explanation at appendix A) cycles for each device are shown in the table below.

In the next diagram, we compare the time for our different executions for testcase1. As we can see we need more time when using 16 bits. In both cases, we see that VHDL implementation is much quicker that C implementation.

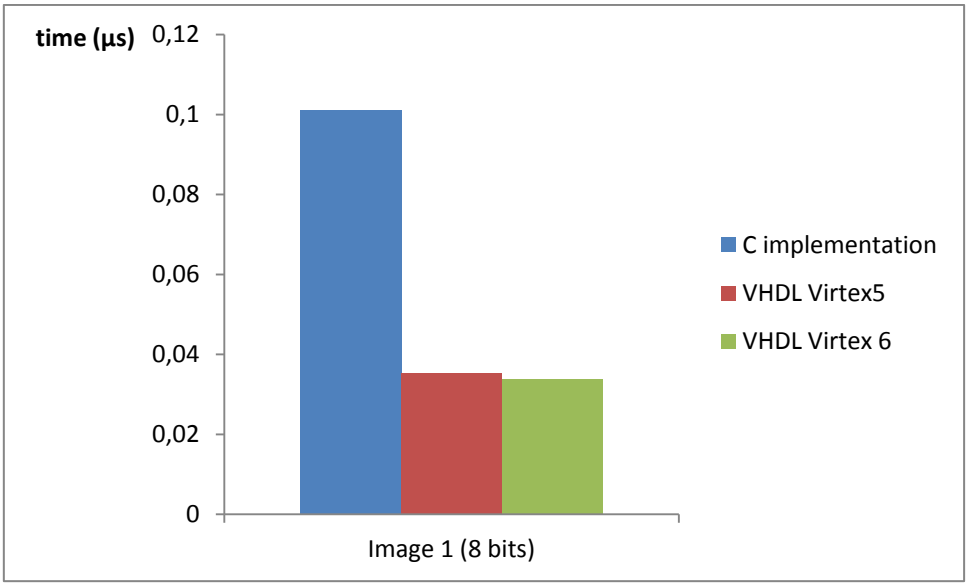


Figure 5.6 Time results for test case 1 (8 bits)

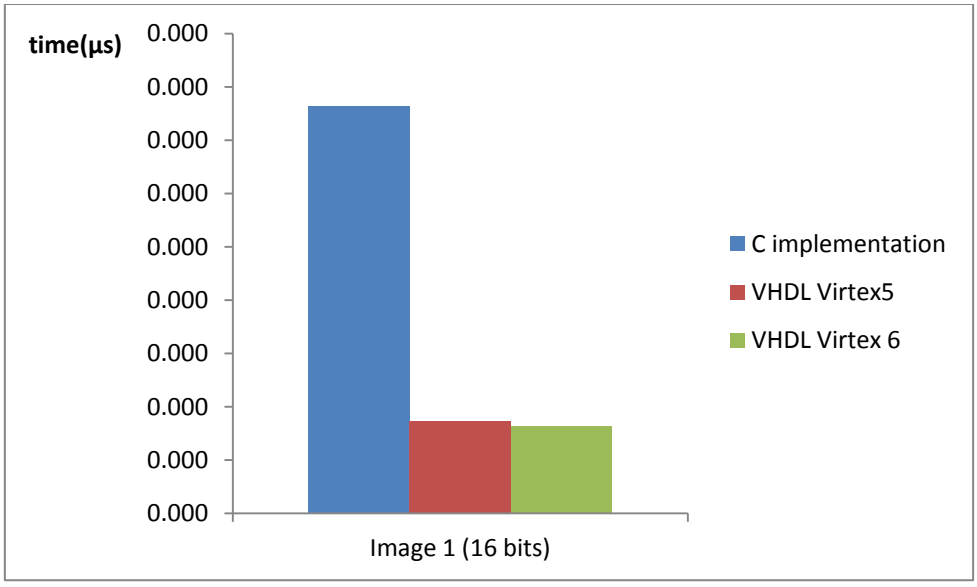


Figure 5.7 Time results for test case 1 (16 bits)

In the next table we preview the results of our 3 implementations (Matlab, VHDL/C).

TestCase 2.

In our next simulation we used the following frames.

Figure 5.8: Input for the testcase 2 with mage size 128x128

Running the simulation in Matlab the two arrays are of (89x64, 99x64) elements which means that our input is of about 12,000 elements.

Like previous, we run our core in Matlab, VHDL and C for both 8 and 16 bits. We can verify that our core works fine, as we get the correct results. As we can see from the above charts, C implementation is much more affected from the range of data, while VHDL does not.

Implementation results are given in the following diagrams:

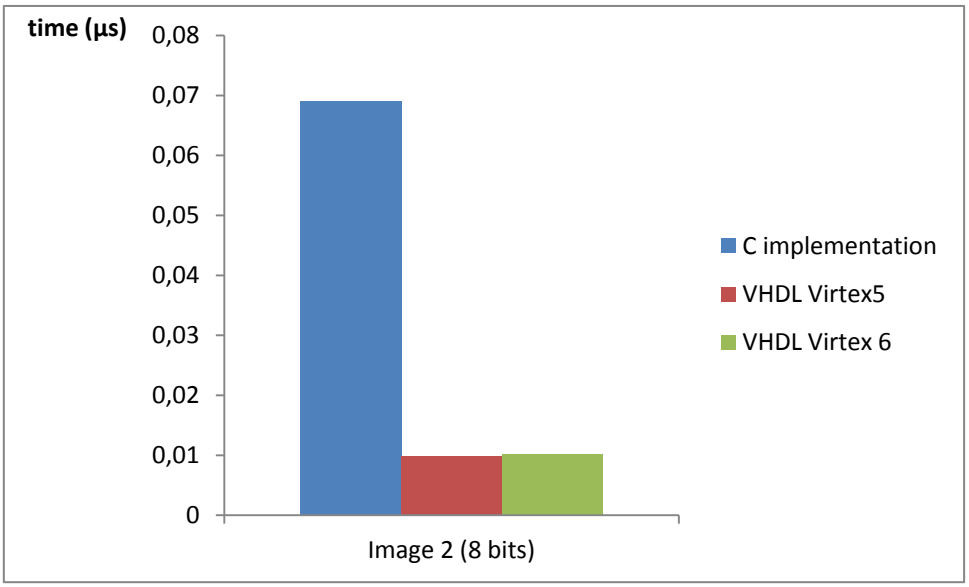


Figure 5.9 Time Results for test case 2 (8 bits)

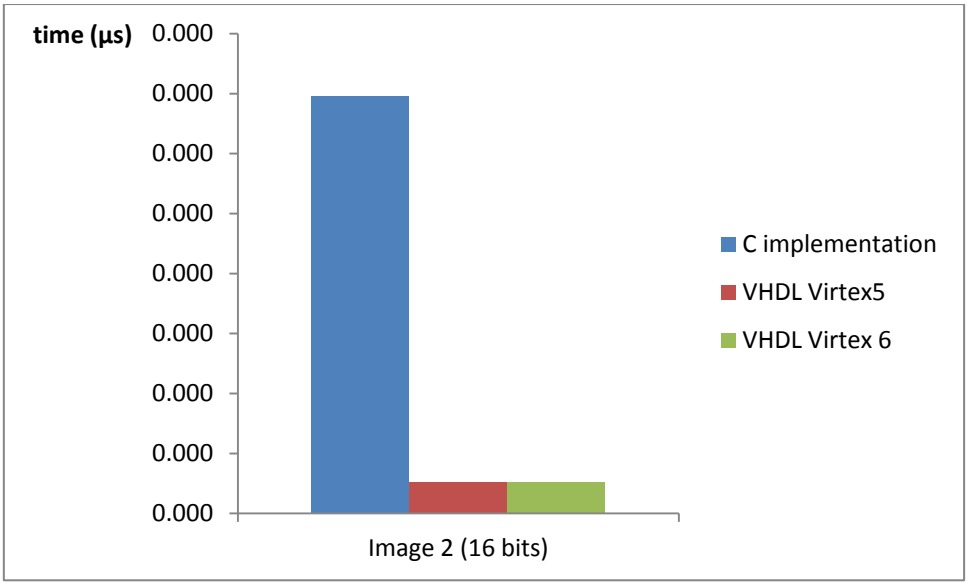


Figure 5.10 Time Results for image 2 (16bits)

TestCase 3

In our next simulation we used the following frames.

Figure 5.11 Input for the test case 3 with image size 64 x 50

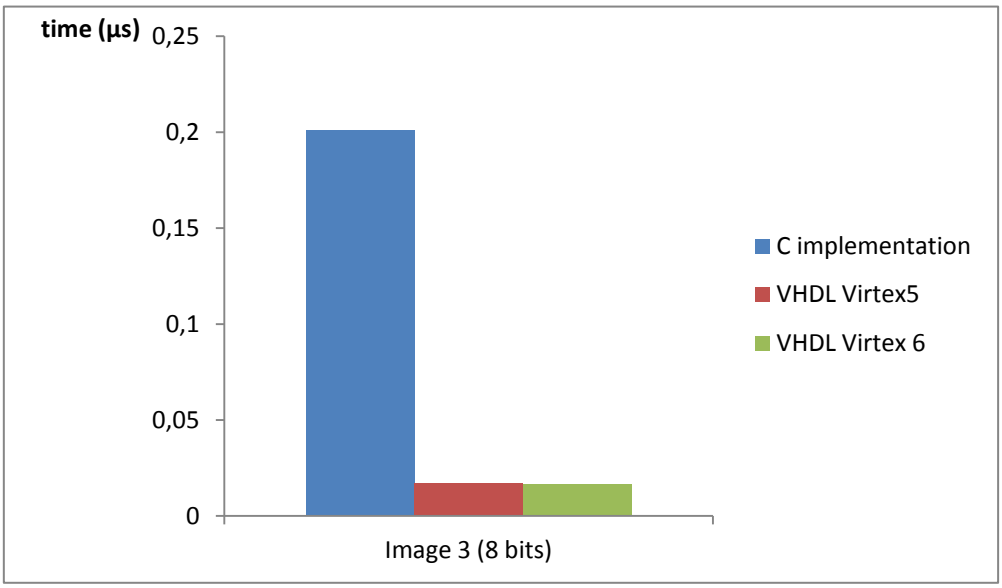


Figure 5.12 8 bits results

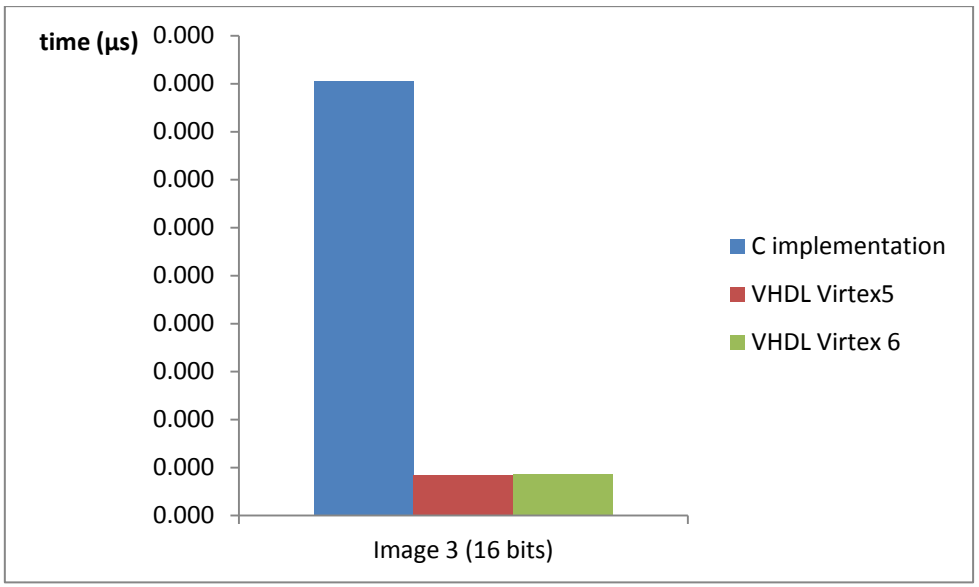


Figure 5.13 16 bits results

TestCase 4

In our next simulation we used the following frames.

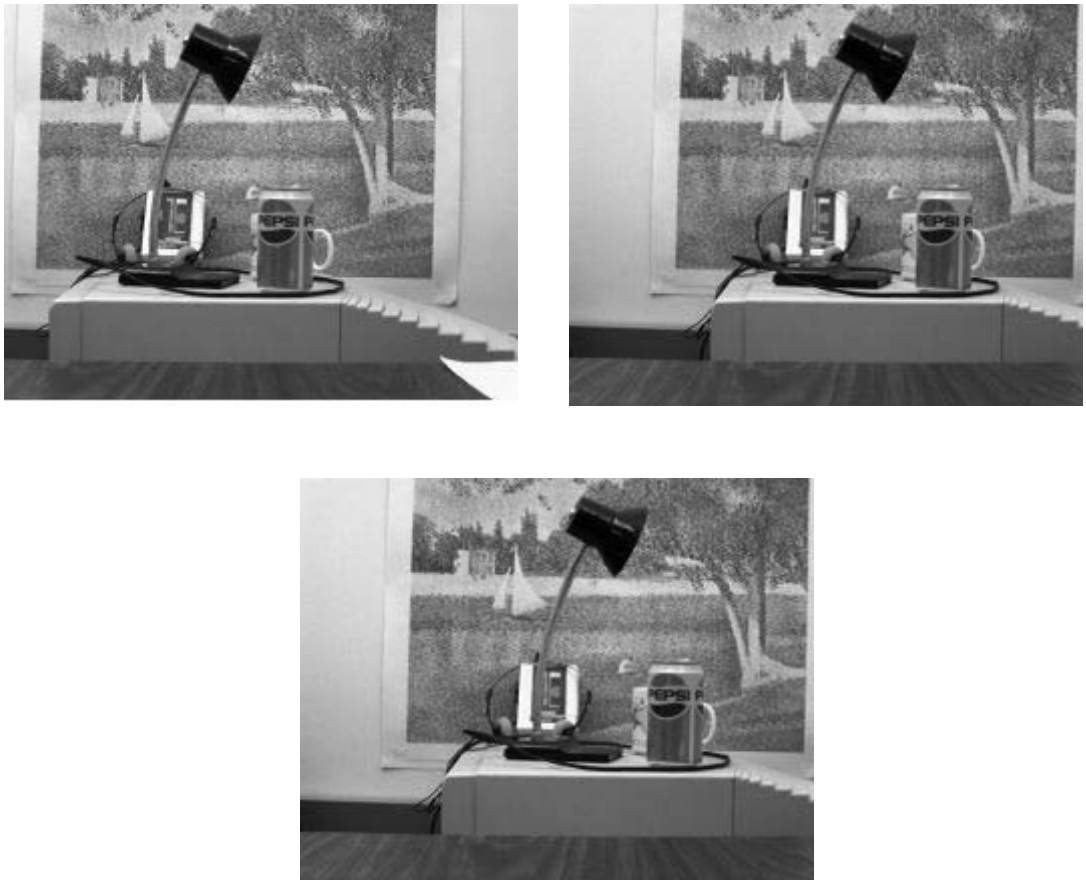


Figure 5.14: Input for the testcase 4 with image size 256x200

Running the simulation in Matlab the two arrays are of (89x64, 99x64) elements which means that our input is of about 12000 bytes.

Like previous, we run our core in Matlab, VHDL and C for both 8 and 16 bits. We can verify that our core works fine, as we get the correct results.

Implementation results in terms of time needed of our core and time needed for C program are given in the following diagrams:

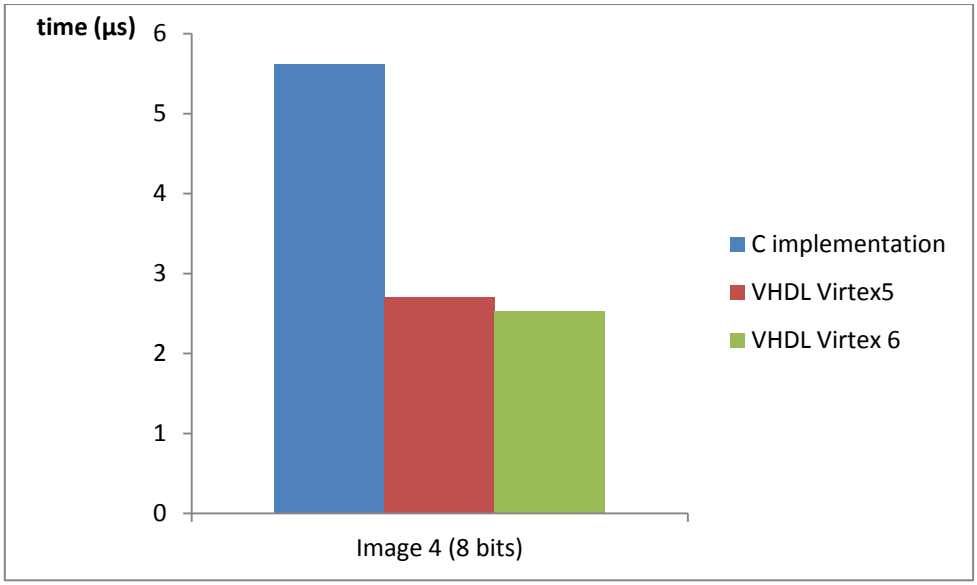


Figure 5.15 Time results for test case 4 (8 bits)

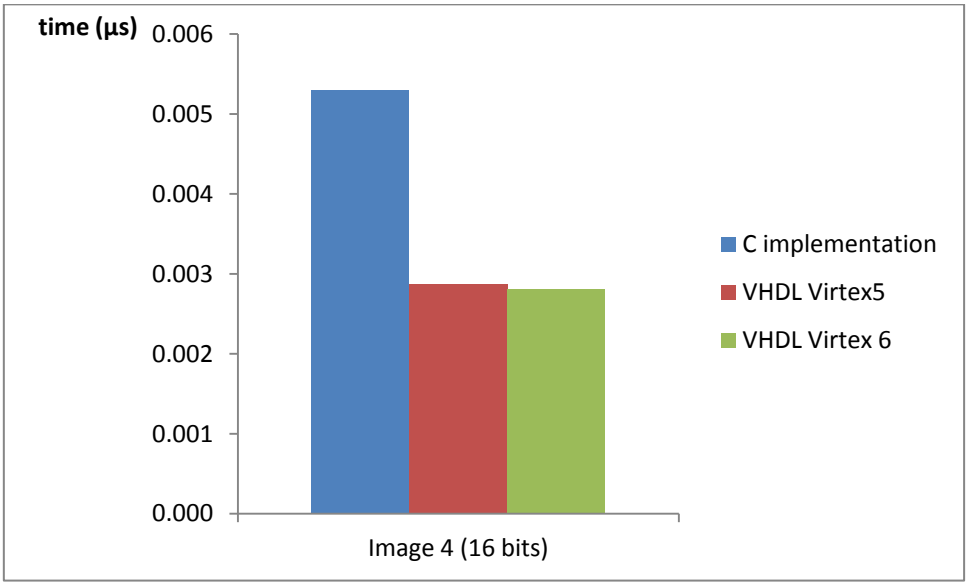


Figure 5.16 Time results for test case 4 (16 bits)

TestCase 5

In our next simulation we used the following frames.

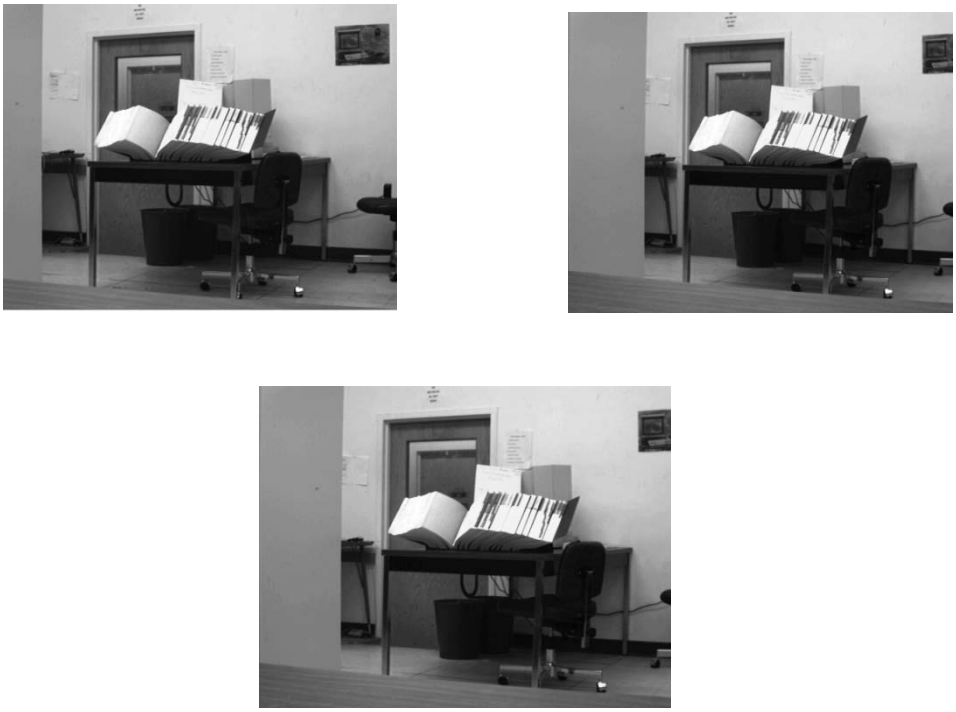


Figure 5.17: Input for the testcase 5 with image size 512x200

Running the simulation in Matlab the two arrays are of (125x64, 110x64) elements which means that our input is of about 15000 bytes.

Like previous, we run our core in Matlab, VHDL and C for both 8 and 16 bits. We can verify that our core works fine, as we get the correct results.

Implementation results are given in the following diagrams:

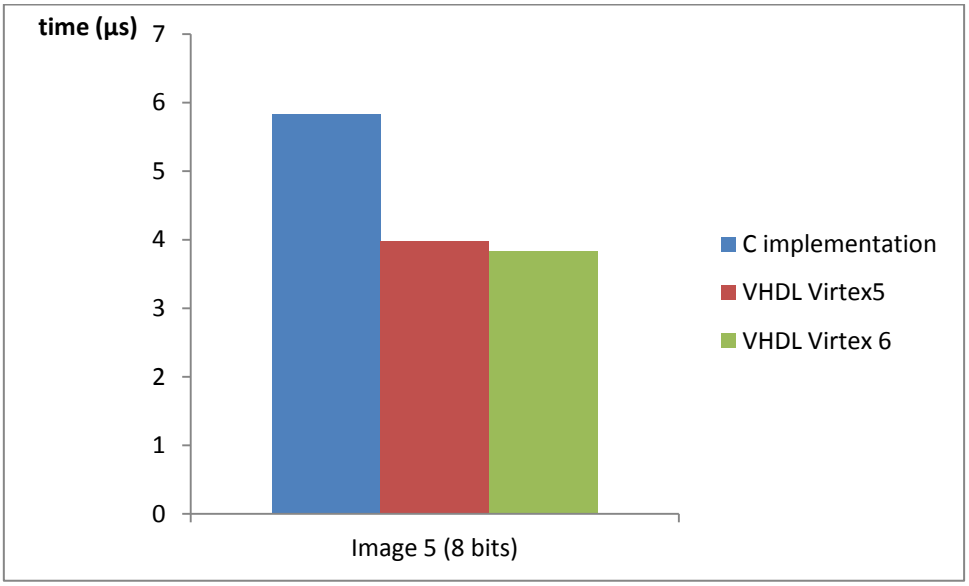


Figure 5.18 Time results for test case 5 (8 bits)

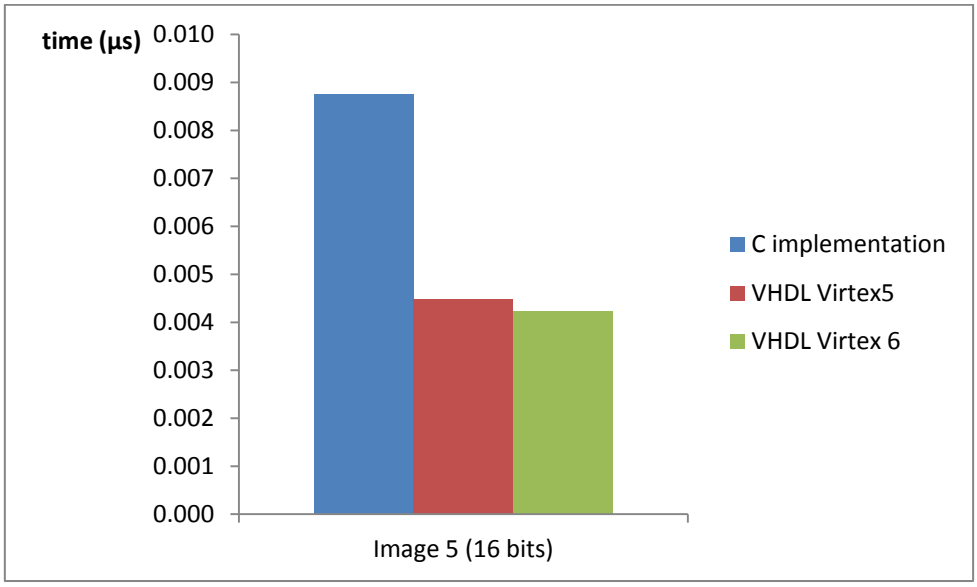


Figure 5.19 Time results for test case 5 (16 bits)

Overall Results

In this section we are going to present our overall result and how the numbre of data influences the time needed from our core in comparison to time needed from C program.

Moreover we have taken results, regarding power consumption and leakage power regarding the number of bytes, at a specific device (Virtex 6 device).

At the end of the section, we provide our overall results regarding cycles needed from our core and from C program in order to show the speedup we have gained which was the initial goal of the design.

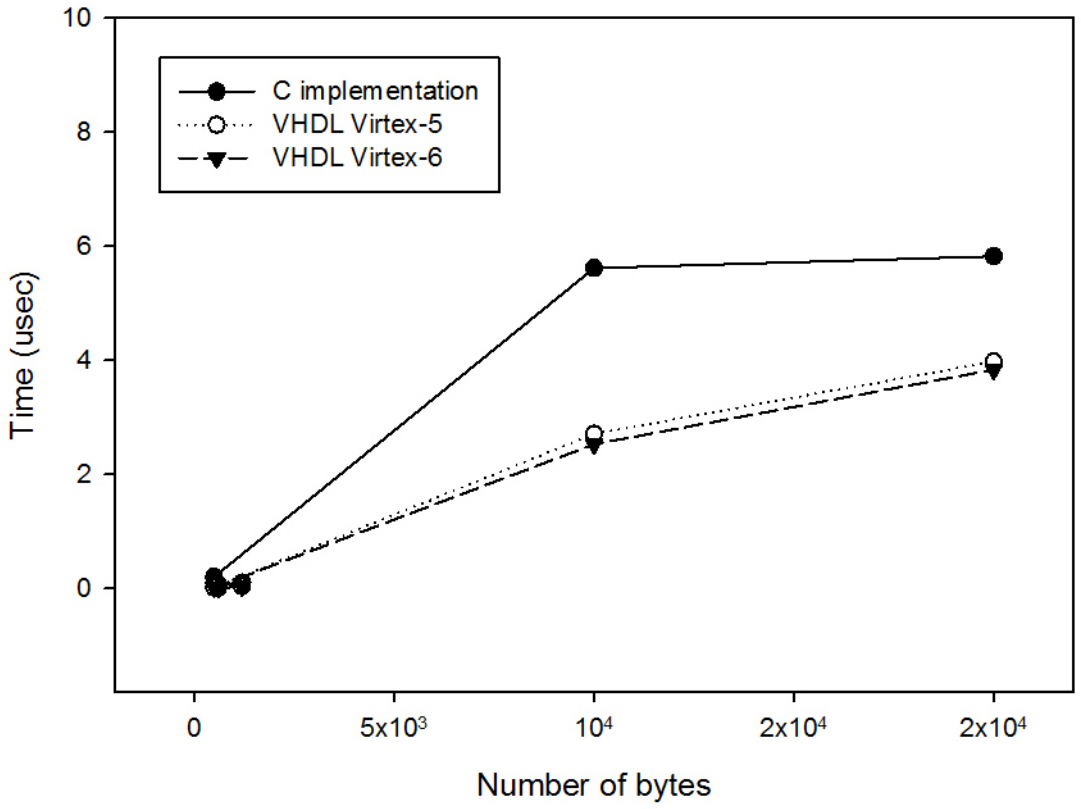


Figure 5.20 Time comparison for different inputs (8 bits)

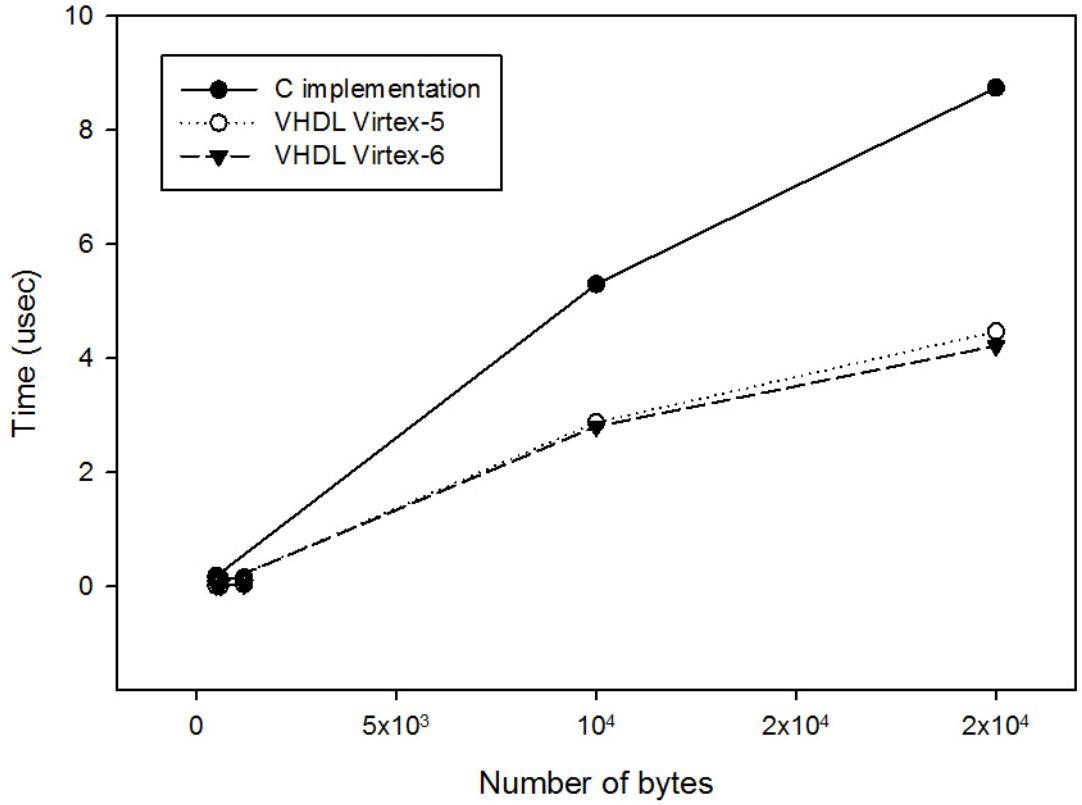


Figure 5.21 Time comparison for different inputs (16 bits)

As we can see, FPGA implementation is always faster than C implementation. In the next diagram we compare directly the cycles needed for each test case, where we can see clearly the speed up we have achieved with FPGA implementation.

For example, for different number of bytes input (used the test cases from above) the cycles needed for C implementation and FPGA implementation comparatively are shown in the following diagram:

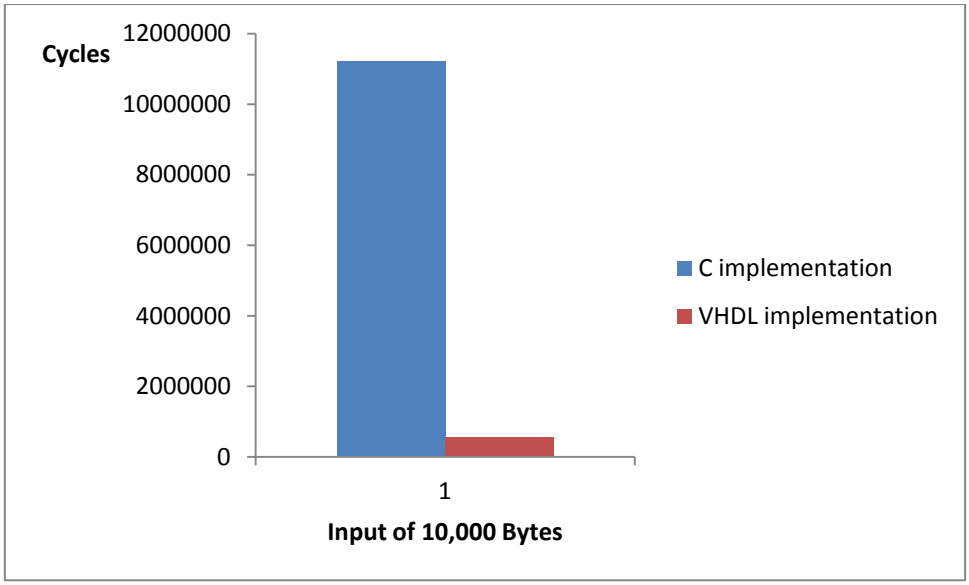


Figure 5.22 Cycles needed for input of 10,000 bytes

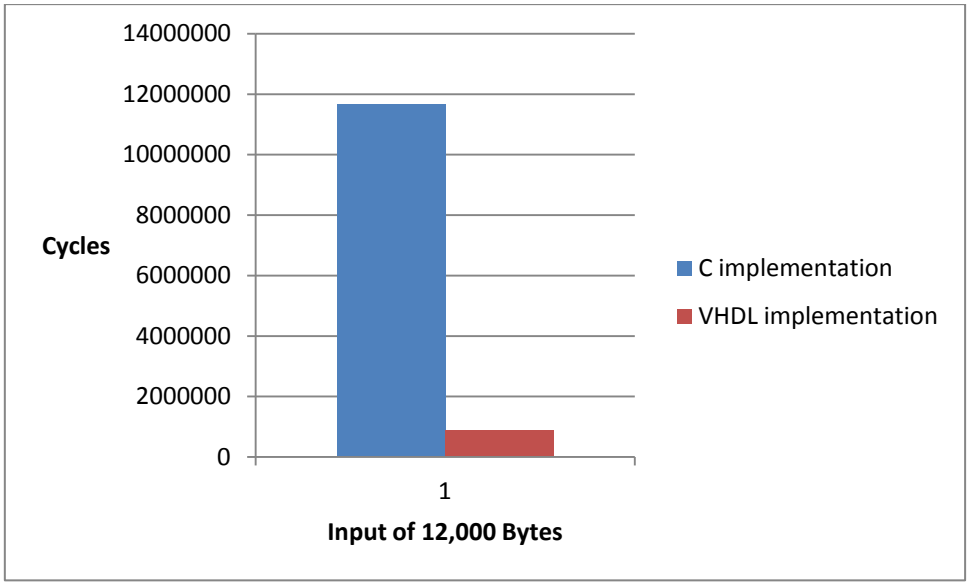


Figure 5.23 Cycles needed for input of 15,000 bytes

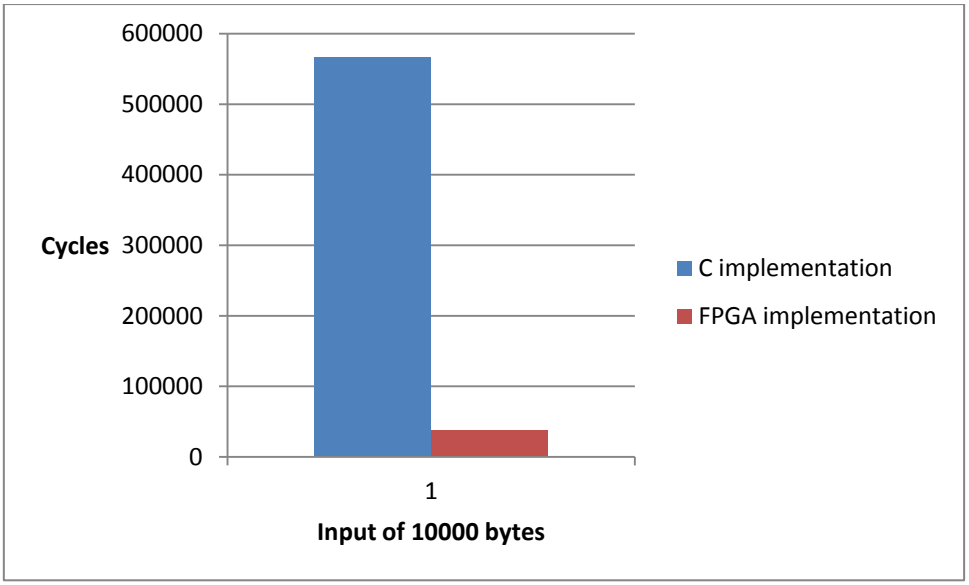


Figure 5.24 Cycles needed for input of 100,000 bytes

From the above diagram we can see that there is a huge difference in cycles needed from our core in comparison with the implementation of the program in a language like C. We expect this difference to be risen as lon as we use larger inputs.

In the next section we see results for power consumption for the previous test cases, taken from XPower tool of Xilinx ISE.

Power results:

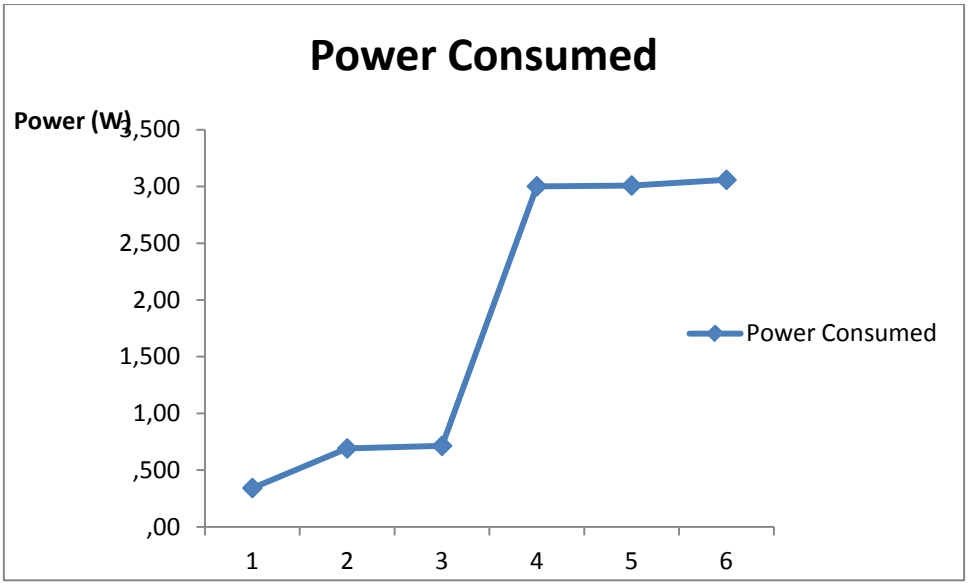


Figure 5.25 Power consumption on Virtex6 for different input

Leakage:

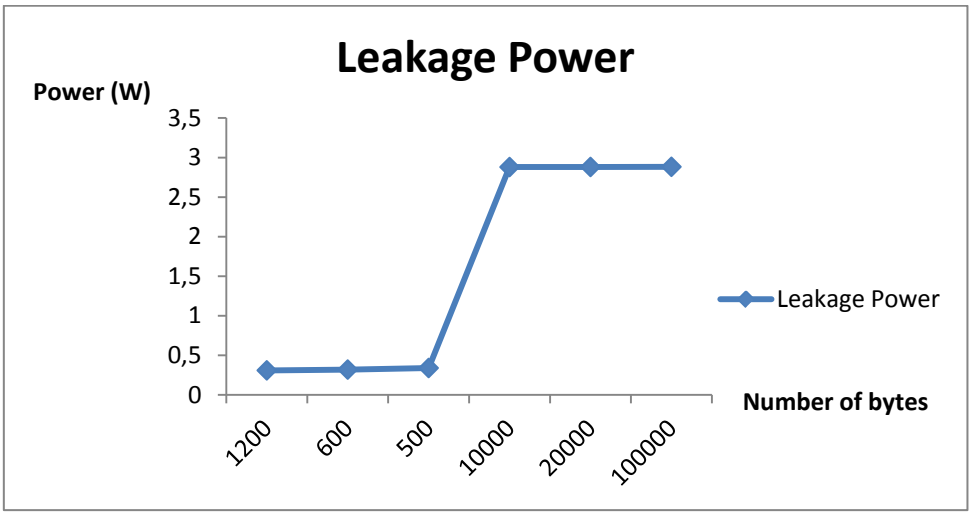


Figure 5.26 Leakage Power on Virtex6 for different inputs

As we can see from the above diagrams, there is a huge rise in power consumption between 500 bytes and 10000 bytes. This difference is expected because the amount of input data is being risen and as a result of this the utilization of components in FPGA increases.

Chapter 6: Conclusions

From the implementation results of the previous section we conclude that if we take advantage of parallelism of an FPGA we manage to speed up our implementation. Moreover, FPGA implementation seems to be unaffected from range of input data while languages like C are getting really slow in number of a wide range.

Especially when comparing cycles needed from the core, we can see that FPGA implementation is much quicker and it requires only the 10% of cycles needed from by C program. As a result of this, if we run our result in an embedded system, our implementation will be faster that running it in a high level language.

We also expect that when number of input increases, FPGA implementation will speed up our project at a greater percent.

From the results provided above, we can see that goals have been complete and we have provided a way that by using hardware for software processes we managed to speed up the algorithm of landmark matching.

Chapter 7: Appendix

Hardware Platform

The device we use for the FPGA implementation is the Xilinx Virtex 6.

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-theartcopper 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.

Synthesis and Simulation Process

The free version of the Xilinx design suite, Integrated Software Environment 13.4, 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, VHDL was the language used. 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. The programming file thus 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 7.1 translation documentation

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 7.2 mapping documentation

Command line tool	MAP
Tcl command	process run "Map"
Input files	NGD, NMC, NCD, NGM 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 7.3 place and route documentation

Command line tool	PAR
Tcl command	process run "Place & Route"
Input files	NCD, PCF 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 7.4 generating bitstream documentation

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

Software execution process

The goal of this appendix is to provide a detailed description on how to run the project, even for someone who knows nothing about VHDL. Basic prerequisite is that reader will be familiar with C language and the UNIX environment. Because driver is implemented for unix, we are going to use a machine running unix. Specifically we used LinuxMint Maya.

First of all, we have to install the driver. Suppose ht the driver is installed in a directory in Desktop with the name driver, we do the following.

- ✓ Open up a terminal and type:

```
Cd /Desktop/driver
```

In this folder if we type *ls* we must see our driver file, and the Makefile.

- ✓ Then we compile it by typing

```
Make
```

Next thing to do is to insert the module of the driver to our linux kernel. In order to do this, we must have root privileges so first we type *su* and make ourselves root.

Having done this, we are ready to insert the module.

- ✓ We execute the following command:

```
insmod fpga.ko
```

If there is no error, nothing must be written on the terminal. Now we are ready to run our program.

Suppose that our C programs for writing and reading to the driver are in a folder named reading, in a directory under desktop.

- ✓ We open a terminal and type:

```
Cd /Desktop/reading
```

Then we compile and run our programs by typing:

```
gcc –o tx_file_plus_len tx_file_plus_len.c
gcc –o rx_file rx_file.c
```

And we run them as follows:

```
./tx_file_plus_len <file>
```

Where file id the file from where we are about to read our input.

In order to receive data from the driver, we execute:

```
./rx_file <file> <number of bytes> <flag>
```

Where:

- File is the output file to where we are about to write our results.
- Number of bytes is the bytes of expected output.
- Flag is a single value: 0 for overwriting data and 0 for appending.

References

[1] Y. Yagi, Y. Nishizawa, and M. Yachida, "Map-based navigation for a mobile robot with omnidirectional image sensor COPIS," *IEEE Trans.Robot. Autom.*, vol. 11, pp. 634–647, Oct. 1995.

[2] Z. Zhu, S. Yang, G. Xu, X. Lin, and D. Shi, "Fast road classification and orientation estimation using omni-view images and neural networks," *IEEE Trans. Image Process.*, vol. 7, pp. 1182–1197, Aug. 1998.

[3] R. R. Murphy, *Introduction to AI Robotics*. Cambridge, MA: MIT Press, 2000, p. 415.

[4] J. Zhang, A. Knoll, and V. Schwert, "Situated neuro-fuzzy control for vision-based robot localization," *Robot. Auton. Syst.*, vol. 28, pp. 71–82, 1999.

[5] A. Rizzi and R. Cassinis, "A robot self-localization system based on omnidirectional color images," *Robot. Auton. Syst.*, vol. 30, pp. 23–38, 2001.

[6] H. Ishiguro and S. Tsuji, "Image-based memory of environment," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst.*, vol. 2, 1996, pp. 634–639.

[7] H. Ishiguro, K. Kato, and M. Barth, "Identifying and localizing robots with omnidirectional vision sensors," in *Panoramic Vision: Sensors, Theory, and Application*, R. Benosman and S. B. Kang, Eds. New York: Springer-Verlag, 2001, pp. 376–391.

[8] S. Atiya and G. D. Hager, "Real-time vision-based robot localization," *IEEE Trans. Robot. Autom.*, vol. 9, pp. 785–800, Dec. 1993.

[9] D. G. Lowe, "Object recognition from local scale-invariant features," in *Proc. Int. Conf. Computer Vision*, Corfu, Greece, 1999, pp. 1150–1157.

[10] S. Se, D. G. Lowe, and J. Little, "Global localization using distinctive visual features," in *Proc. Int. Conf. Intell. Robots Syst.*, Lausanne, Switzerland, 2002, pp. 226–231.

[11] A. Gruen and T. Huang, Eds., *Calibration and Orientation of Cameras in Computer Vision*. New York: Springer, 2001, pp. 63–94.

[12] M. A. Fischler and R. C. Bolles, "Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography," *Commun. ACM*, vol. 24, no. 6, pp. 381–395, 1981.

[13] F. Dellaert, S. M. Seitz, C. E. Thorpe, and S. Thrun, "EM, MCMC, and chain flipping for structure from motion with unknown correspondence,"

Machine Learning, vol. 50, pp. 45–71, 2003.

[14] C. C. Slama, *Manual of Photogrammetry*, 4th ed. Bethesda, MD: Amer. Soc. Photogrammetry, Remote Sensing, 1980.

[15] J. O'Rourke, *Art Gallery Theorems and Algorithms*. Cambridge, U.K.:Oxford Univ. Press, 1987, p. 126.

[16] K. T. Simsarian, T. J. Olson, and N. Nandhakumar, "View-invariant regions and mobile robot self-localization," *IEEE Trans. Robot. Autom.*, vol. 12, pp. 810–816, Oct. 1996.

[17] A. Okabe, B. Boots, K. Sugihara, and S. N. Chiu, *Spatial Tessellations:Concepts and Applications of Voronoi Diagrams*, 2nd ed. New York: Wiley, 2000, pp. 169–178.

[18] N. S. Rao, N. Stoltzfus, and S. Iyengar, "A "retraction" method for learned navigation in unknown terrains for a circular robot," *IEEE Trans.Robot. Autom.*, vol. 7, pp. 699–707, Oct. 1991.

[19] O. Takahashi and R. Schilling, "Motion planning in a plane using generalized Voronoi diagram," *IEEE Trans. Robot. Autom.*, vol. 5, pp. 143–150, Apr. 1989.

[20] S. Thrun, "Learning metric-topological maps for indoor mobile robot navigation," *Artif. Intell.*, vol. 99, pp. 21–71, 1998.

[21] H. Choset and K. Nagatani, "Topological simultaneous localization and mapping (SLAM): Toward exact localization without explicit localization," *IEEE Trans. Robot. Autom.*, vol. 17, pp. 125–137, Apr. 2001.

[22] I. Konukseven and H. Choset, "Mobile robot navigation: Implementing the GVG in the presence of sharp corners," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst.*, vol. 3, 1997, pp. 1218–1223.

[23] D. C. Yuen and B. A. MacDonald, "Natural landmark based localization system using panoramic images," in *Proc. IEEE Int. Conf. Robot.*

Autom., vol. 1, Washington, DC, May 2002, pp. 915–920.

[24] , David C. K. Yuen, *Member, IEEE*, and Bruce A. MacDonald, *Senior Member, IEEE*, Vision-Based Localization Algorithm Based on Landmark Matching, Triangulation, Reconstruction, and Comparison, IEEE TRANSACTIONS ON ROBOTICS, VOL. 21, NO. 2, APRIL 2005 217,

[25] Vytautas ŠTUIKYS, Design of Reusable VHDL Component Using External Functions, INFORMATICA, 1998, Vol. 9, No. 4, 491–506 491, Ó 1998 *Institute of Mathematics and Informatics, Vilnius*