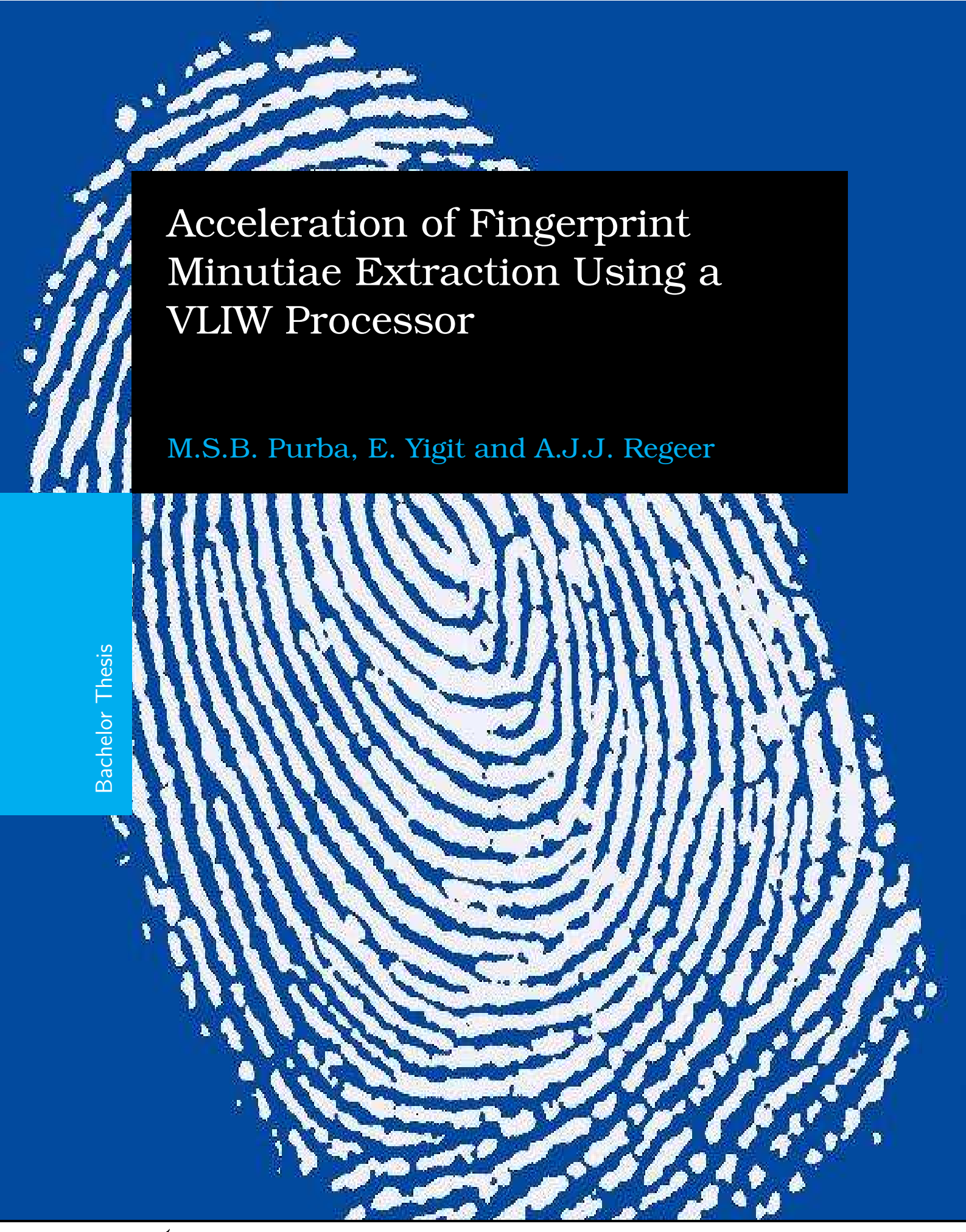


Acceleration of Fingerprint Minutiae Extraction Using a VLIW Processor

M.S.B. Purba, E. Yigit and A.J.J. Regeer

Bachelor Thesis

TU Delft Delft University of Technology

Bachelor of Electrical Engineering

Acceleration of Fingerprint Minutiae Extraction Using a VLIW Processor

Bachelor Thesis

M.S.B. Purba, E. Yigit and A.J.J. Regeer

August 23, 2011

Faculty of Electrical Engineering, Mathematics and Computer Science · Delft University of Technology

TU Delft Delft University of Technology

Preface

This thesis has been written in the context of the Electrical Engineering bachelor exam at the Delft University of Technology. The project its title is: 'Acceleration of Embedded Applications Using a VLIW Processor'. The project was conducted at the Computer Engineering group of the department of Electrical Engineering, Mathematics and Computer Science.

Chapter 1 gives the introduction of the project and specifies the exact problem. Chapter 2 presents some alternatives of applications using the ρ-VEX and explains why the fingerprint minutiae extraction application was chosen as vehicle to demonstrate the capabilities of the VEX implementation. In chapter 3 relevant theory is presented on embedded systems design, which is used in later chapters to explain our results. In chapter 4 the background of the chosen application is presented and the modified version of the software package is explained. In chapter 5 the setup of the experiment is described such as the FPGA-board, the VEX system, and the additional software tools that were written to extract the binary code for the VEX. In chapter 6 the results of the experiment are discussed, in particular the details of the partitioning of the application and the resulting system is described. In chapter 7 conclusions and recommendations are described, concluding this thesis.

We would like to thank our supervisor Fakhar Anjam for his valuable advice and support. Furthermore, we are indebted to Roël Seedorf and Anthony Brandon for their help on many occasions. A special thanks goes to Koen Bertels for his pragmatic solutions in unforeseen circumstances.

Delft, 24 August 2011,

Mandaren Purba, Erkut Yigit and Arjan Regeer

Table of Contents

List of Figures

List of Tables

Summary

The research group working on the ρ-VEX VLIW processor has expressed its desire that the performance of the ρ-VEX as accelerator in embedded applications is investigated further. This thesis presents a hardware-software co-design of a fingerprint minutiae extraction application in a Xilinx Virtex-6 FPGA using the ρ-VEX processor for acceleration.

Research on embedded systems design has shown that in order to meet contemporary constraints on embedded systems, systems have to be built as heterogeneous multiprocessor platforms, i.e. systems where the application is divided into several tasks, and where the hardware is tailored to its specific task, resulting in systems that contain several different processors and hardware specifically designed to perform some of the tasks.

VLIW processors have certain properties that make them in particular applicable in the design of embedded systems. Their design is less complex compared to other CPUs, which results in low cost, energy efficient, high performance processors.

At the computer engineering group of the Delft University of Technology research is being conducted on the implementation of a VLIW architecture processor based on the VEX instruction set architecture, called the ρ-VEX. In order to acquire data about the performance of this VEX implementation, embedded applications are built using the ρ-VEX as accelerator.

The goal of this project has been to present a hardware-software co-design of an application, demonstrating the specific capabilities of the ρ-VEX as accelerator. In particular a fingerprint minutiae extraction application has been partitioned in a Virtex-6 FPGA, using the MicroBlaze soft core processor and the ρ-VEX.

The first thing we did was to adapt the existing fingerprint extraction program such that we could run it on the MicroBlaze processor. For this we had to remove some troublesome functions and replace them by functions that posed no problem for the MicroBlaze system. Second, we had to rewrite the part of the program that reads in the compressed fingerprint image. Before we could run the program we had to create a MicroBlaze platform on which to run the program. A MicroBlaze platform was created on which the VEX processor was also already present. Once we did that we were able to run the program on the MicroBlaze platform. The next step was finding out the distribution of execution times over the different functions. The program was profiled in order to find out which parts of the program took the most execution time. These identified parts were then removed from the program and implemented in a program of its own, to be executed on the VEX processor. In order to run

this program on the VEX simulator we had to rewrite the original program to produce some blocks of data that the VEX program needs. Once this was done we were able to run the VEX program on the VEX simulator for several configurations of the VEX processor. Once the optimal configuration for the VEX processor was obtained from these simulations, we optimized the program for the VEX, using the different optimizing options for the VEX C compiler.

We obtained a speedup of 1.13 for the computation of one DFT block. This speedup was obtained by estimating the execution time of the kernel on the ρ-VEX because we were not able to get the ρ-VEX running. It is not possible to get an estimation for the whole application.

Chapter 1

Introduction

Embedded systems have to meet often conflicting requirements. They have to provide real-time computing power, but at the same time must be energy efficient, and development time has to be kept low because of time-to-market constraints. The area that the circuit occupies has to be kept small because costs increase exponentially with size. Designers have chosen to use parallelism to meet these different constraints. After careful analysis some of the functionality is implemented in software and some in hardware, where the hardware can be some processor running its own software or it can be some dedicated hardware specifically designed for just this function. A specially designed piece of hardware implementing some function is called hardware acceleration. Besides the obvious advantages that hardware acceleration offers, it has some disadvantages too. Hardware designs take a lot of time to develop, introducing additional labour costs compared to software design and risking violation of time-to-market constraints. The circuits are often designed for just one function, limiting their reuse in other designs. This short lifetime of these circuits make them expensive. Finally, most dedicated circuits cannot be programmed, so errors in design cannot be compensated for by reprogramming, as is the case in software design. A solution is needed that offers the speed advantage of hardware design and the flexibility of software design without the disadvantages that traditionally come with hardware design. Advances in technology offer new possibilities to solve these issues.

1-1 Project Statement

At the Delft University of Technology research is being done on a reconfigurable softcore processor, called ρ-VEX. With this processor it will be shown that designing with VLIW processors offers some substantial benefits compared to the more conventional design approaches. In order to show that the VLIW approach indeed meets the requirements of embedded system design, many existing applications are being rebuilt using the ρ-VEX. The objective of this project has been to present a hardware-software co-design of an application, in this case the fingerprint verification application, demonstrating the specific capabilities of the ρ-VEX as accelerator.

1-2 Workplan

This project starts with selecting an application that is going to be implement on the ρ-VEX processor. The next step is doing the literature research on the application that has been chosen and the architecture of the ρ-VEX processor. It is necessary to get an understanding of the application and the ρ-VEX processor. Subsequently, we need to study the fingerprint application source code before running them on the VEX simulator. It provides us the performance estimate of executing fingerprint application on the target processor. After this step is completed, the fingerprint application is run on the VEX simulator. This enables us to decide which issue-width should be used on the ρ-VEX processor: 1, 2, 4 of 8. The following step is to get the application to be able to run on MicroBlaze and then to analyze the application to find out for each part of the application how much execution-time and how many clock-cycle they have consumed. This will enable us to determine the bottlenecks in the application. After detecting the bottlenecks, these parts with insufficient speedup will be mapped to the ρ-VEX processor. In the next step, this mapping will be implemented to gain a higher speedup of these functions and the final speedup will be measured as the last step.

In the following the overall strategy to reach the project objectives are described:

- Select an application that is going to be implemented on the embedded system and find the available software application
- Research on software application and the ρ-VEX architecture
- Explore and study the source code of the application
- Run the application VEX-simulator
- Adapt and modify the application in order to be able to run on MicroBlaze and the ρ-VEX
- Run the application on MicroBlaze
- Analyze the bottlenecks(kernel) of application
- Mapping onto MicroBlaze and the ρ-VEX
- Implement the mapping
- Measure the speedup that the ρ-VEX has delivered in accelerating the application

1-3 Thesis Layout

The structure of this thesis is as follows. In chapter 2 some alternative designs of the application are presented and why the fingerprint minutiae extraction application was chosen as vehicle to demonstrate the capabilities of the ρ-VEX implementation are also explained. In chapter 3 relevant theory is presented on embedded systems design, which is used in later chapters to explain our results. In chapter 4, the background of chosen application is presented and the modified version of the software package is explained. In chapter 5 the setup of the experiment is described such as the FPGA-board, the ρ-VEX system, and the additional software tools that were written to extract the binary code for the VEX. In chapter 6 the results of the experiment are discussed, in particular the details of the partitioning of the application and the resulting system is described. In chapter 7 conclusions and recommendations are described, concluding this thesis.

Chapter 2

Alternative Product Designs using the ρ-VEX

In this chapter, a decision will be made about what kind of product the designers will make at the end of this project. This product is based on the accelerated fingerprint verification application that works with a higher rate of speedup by using the ρ-VEX processor. Today, there are many embedded systems or products requiring very high computing power and using the ρ-VEX in embedded applications is one of the solutions to achieve a high computation system. In order to get a decision, in section 2.1, some different forms that the hardware can take for fingerprint verification, within the security market, will be compared and a choice from these forms will be made. Section 2.2 deals with the requirements that the chosen design form will have to meet and in the last section, conclusion for this chapter is given.

2-1 Chosen Product Design

Fingerprint verification is one of the oldest forms of biometric identification and its use is varied. The FBI (Federal Bureau of Investigation) for example uses fingerprints to identify persons that were present at a crime scene and its use has increased in recent years for civilian purposes as well. Another use is in security issues and access control. Entrance to buildings is increasingly controlled by fingerprint verification, figure 2-1 shows an example of a door lock.

As has been mentioned earlier the task at hand is to design a hardware solution for minutiae extraction containing, in some form or another, the VEX processor. The hardware solution in this project can take on many forms, but the designers have chosen to narrow the design to the following three different design forms. These are listed as:

1. A PCI card: A Peripheral Component Interconnect (PCI) card that can be added to the motherboard of a personal computer. PCI fingerprint card allows computer users to identify the fingerprint of each personal data.

Figure 2-1: An example of a doorlock that uses fingerprint verification as a way to control access. Source: http://walyou.com/biometric-finger-print-door-lock/

2. A closed box that can be attached to the PC or other computer by means of a USB connector or other communication port. This can also be seen as an external PCI card but as it will follow below, it has a disadvantage compared to a PCI card.

3. An embedded piece of hardware, for example as part of a doorlock. Within these kind of doorlocks, the fingerprint verification application from this project is the key. In this kind of doorlock, the keyed locking mechanism is replaced with a fingerprint sensor that actually recognizes who is and who is not authorized to enter.

As it is apparent from the research on the security markets made for the Business Plan that is already established before this project, it is expected that the demand for PCI cards in the coming years will increase alongside other forms of products listed above. It might be considered that the safety measures in customs (access controls) or safety measures to be taken by secret or security services will be made more stringent to counteract the increasing terrorism. From this reasoning item 3 of the list above falls off. As it is already said, a closed box has a disadvantage compared to a PCI card. A PCI card works with a higher rate of speedup because it can make direct communication with the main computer, without using any cable or connector, which is in turn an important requirement at customs (access controls) and secret services. In this case, a Peripheral Component Interconnect (PCI) card has been chosen to be the design form of the hardware solution in this project.

2-2 Program of Requirements

There are some requirements and conditions that the PCI card produced has to meet. The requirements and conditions concern intended use, business strategy and marketing, the usability of the product etc. They are listed as follow:

Figure 2-2: Peripheral Component Interconnect card (PCI) of fingerprint. *Source: http://www.topproduct.nl/hardware/hubs/12249-belkin/78818-belkin-firewire-800-3-port-pci-card/*

Requirement on intended use of the fingerprint application

- The system has to work with higher speed compared to its software solution.
- The system has to operate as a PCI-express; a plug-card for different computers.
- The system should not save the fingerprints on its memory.
- Required drivers associated with the system need to be delivered to the users.

Precondition

- The system has to produce the required minutiae list by the user within 3 seconds.
- The system has to be able to run with 4 issue slots on ρ-VEXVEX processors.
- The product have to offer secure and powerful computer protection, so it must not cause any damage to the computer it is connected to.

Regulation

The design of the interface must not infringe patents or other intellectual property rights of competitors.

Liquidation properties

- The product users will receive an envelope including freepost to return the product back for recycling when it stops working.

Settings of the Application

User manual/software CD and software license code has to be included in the product package.

Usability of the Application

- The product have to be easily fixed on and removed from PC.
- The product has to make a back-up in PC in order not to lose the saved information.
- Advanced Password security has to be available to protect the users.
- The application has to include an installation guide that enables the users to install the application easily.
- No matter dry or wet of your finger skin, it can get clear fingerprint image easily.
- Multilingual support: English / Japanese / Traditional Chinese /Simplified Chinese / French / German

Business Strategy and Marketing

- The product must have an internal security code that protects the sellers against reverse engineering.
- The users must be offered quick service in case of a technical malfunction.
- Orders up to 3 products must be delivered to the users within one week without any shipping costs. For orders more than 3 products, 40% of the shipping costs will be paid by the users.
- The product has to able to support Windows 2000, Windows XP, Windows Vista and Windows7.

2-3 Conclusion

In this chapter the reader have seen what kind of possible forms the hardware solution of the fingerprint verification application can take. From the three possible forms, namely a PCI card, a closed box with USB connection or communication port and an embedded piece of hardware such as a doorlock system, the PCI card has been chosen to be the end product to be made in this project. It has been chosen because it is expected that the demand for this cards will be increase and it also has the advantage to operate with higher rate of speed up compared to a closed box which can be seen as an external PCI card. As all products have to meet some criteria, there are also some requirements and conditions that this PCI card have to meet such as that the product have to be easily fixed on and removed from PC , the system has to produce the required minutiae list within 3 seconds,the system should not save the fingerprints on its memory etc.

Chapter 3

Embedded Systems Design

3-1 Introduction

Embedded computing systems are part of a bigger system and are necessary for their operation. They enable part or most of the functionality of the device which it is part of. Without embedded systems technology some developments would not be economically feasible [1]. Take for example the developments in telecommunication such as mobile phones, or engine control in automobiles.

Embedded computing systems are found everywhere. Table 3-1 lists some areas of application, and gives some examples of systems in each application area.

APPLICATION AREA	EXAMPLES
Military	Rocket missile guidance systems, digital radio
Communication	Mobile phones, wireless systems, satellites
Consumer	Washing machines, televisions, microwaves
Industry	Control systems, safety systems

Table 3-1: Some application areas in which embedded systems can be found.

Embedded computing systems differ from the general-purpose computing systems that we know from the desktop. For desktop systems, such as the personal computer, and other similar computing systems the main task of these systems is computing, and energy consumption and other costs are of secondary concern. When designing embedded systems, energy consumption and design costs have to be considered as well.

The goal of embedded system design is to build embedded systems that offer enough (real-time) performance, are cost-effective, and operate at low energy levels [2]. These constraints make building embedded systems a challenge. Research on embedded system design has shown that in order to meet these constraints it is necessary to design hardware and software concurrently. The design methodology that takes this into consideration is called co-design. The goal of co-design is to make appropriate design decisions early in the design process such

that hardware and software can be implemented independently in later stages of the design process. One aspect of co-design is software/hardware partioning, where it is decided which parts are implemented in hardware and which parts are implemented in software. Apart from choosing the right design methodology, it has also become clear recently that using available parallelism is the only viable solution in meeting ever more challenging constraints.

3-2 Parallelism in Embedded Applications

As was said in the previous section, in order to meet the design constraints on embedded systems we have to exploit all available parallelism in the application. Parallelism can be distinguished at various levels of abstraction. We distinguish the following forms of parallelism:

1. *Instruction Level Parallelism (ILP).* Instruction level parallelism is a fine grained form of parallelism that becomes available when the application has been translated into a sequence of machine instructions. Instructions that do not depend on each other can be executed concurrently. Most modern processors utilize this form of parallelism by issuing more than one instruction per clock cycle.

2. *Data-level parallelism.* Some applications contain a lot of data on which the same operation has to be performed. Designers of processors have introduced a new type of instruction to take advantage of this type of parallelism, called an SIMD-instruction. Without such an instruction it would be necessary to retrieve the same instructions from memory for each data item, which would result in a very inefficient use of memory.

3. *Thread-level parallelism.* When a processor executes a program it can not always find enough instructions to fill all execution units. To fill these execution units, it executes another thread of the program concurrently with the main thread, thereby utilising all available resources provided by the execution units.

4. *Task-level parallelism.* Most applications, and especially the embedded applications, can be divided into tasks that have no strong dependency on each other and could therefore be executed concurrently. It is this form of parallelism that provides the most opportunities to meet the design constraints in embedded applications.

Once it has been decided to add more processing elements to do more computations in parallel it is not to say that one and the same architectural solution is always the best solution for every type of application. For instance, multithreaded CPUs and multicore CPUs were compared and it was shown that applications that are typical in servers benefit more from multithreaded CPUs, while numeric applications benefit more from multicore CPUs [3]. Although this example comes from the general computing domain, the conclusions apply to the embedded systems domain as well. This shows that design decisions have to be taken carefully in deciding how the available chip real estate is used, and that the specific application's requirements can't be ignored. In the next section it is shown how parallelism is exploited in the design of embedded systems.

3-3 Distributed Embedded Systems

To cope with the demanding requirements on embedded systems, i.e. increase their performance while at the same time keeping them cost-efficient and keeping their energy consumption low, designers of embedded systems embrace some new design techniques such as (heterogeneous) multiprocessor architectures, distributed memory, and custom designed interconnection networks. These design techniques enable designers to come up with more energy-efficient and cost-efficient platforms.

3-3-1 Multiprocessor Platforms

Most embedded applications contain a lot of task-level parallelism. Take for example the modern cell phone.

1. It needs to communicate with the network, performing several network protocols.
2. It performs speech compression and decompression.
3. It needs to provide the interface to the user.
4. Modern phones provide cameras which need compression of the images taken.
5. A lot of phones provide audio support for most popular formats, such as MP3, AAC, and other formats. This requires a lot of computation as well.

A characteristic of these tasks is that they can be performed relatively independently of each other. This enables designers of these cell phones to divide the design over several processing elements.

Different tasks have different computing needs. Because not all tasks need the same computing power we are able to modify processing elements to the task at hand. The processor core can be adapted to the specific needs of the task. For example, when the task is very sequential in nature and hence there is not much ILP it would be a mistake to use a superscalar processor. It will not be able to issue a lot of instructions, but nevertheless it will consume a lot of needless power. A simpler core, without issuing logic would probably do the same job, requiring less power, and because it lacks a issuing logic unit it will have lower design costs.

Other tasks are best implemented in a hardwired fashion, resulting in a cost-effective and energy-efficient solution that will provide the expected computing performance. They do not require the flexibility that a programmable CPU provides. The designer has to make a decision as to implement tasks into a special hardware unit or using a CPU.

Multiprocessing offers better real-time performance at lower energy consumption and design costs. Using multiple processors in the design of embedded systems enlarges the design space. It enables designers to make more local modifications that only affect the local processing elements. For example, it is possible to use a memory system for just one processing element which allows the designer to tailor the memory system to the specific requirements of the processing element. Another example is that the designer can remove parts of a processing unit that are not needed for the task, thereby reducing costs and energy consumption by

the processing element. Multiprocessing leads to more design options, it allows the designer to make more design choices when designing the computing units for each task, and thereby enables him to adapt the system such that it better matches the specific needs of an application in terms of performance, energy efficiency and costs.

It is clear that multiprocessing provides more cost-effective and energy-efficient systems, but multiprocessor designs confront the designer with some new challenges as well. Interconnection between different computing units becomes an issue. Interconnection can take up an area that of three cores on a multicore processor and take the equivalent of one core on power consumption [4]. This shows that the design of multiprocessor systems cannot be seen independently of interconnection of these different processing elements. The different aspects of the design of a multiprocessor system such as memory issues, caches, number and type of processing elements, and interconnection cannot be considered in isolation.

3-3-2 Heterogeneous Multiprocessor Platforms

Embedded systems can be classified by their number and type of processing elements. In the early days of embedded computing most systems had one CPU, the so called uniprocessor designs. Embedded systems soon needed more computing performance and therefore more powerful CPUs were applied in these uniprocessor designs. So, more powerful CPUs were needed. But designers of processors are faced with increased power consumption and heat dissipation in their effort to improve processor performance. It is obvious that physical limitations prohibit further improvements in the design of CPUs. This was soon realized and a new class of embedded systems was created, that of the (symmetric) multiprocessor systems. As was discussed in the previous section this allowed designers to design systems that were at least as powerful as uniprocessor designs, but in a much more efficient form and at lower costs. Lately, designers of embedded systems have started to use different types of CPUs in one design, thereby creating a new class of embedded systems called the heterogeneous multiprocessor embedded systems.

In heterogeneous multiprocessor systems the processing cores are not all the same, as is the case with symmetric multiprocessor systems. This allows applications to be matched to the processing core that is best equiped for its computing demands. Diversification of the specifications of the cores ensures that a better match is obtained in terms of the workload, while at the same time gains can be accomplished with respect to power consumption and real estate, because less complex cores require less energy and area on the chip.

Heterogeneity offers more flexibility in the design choices, because every aspect of the multiprocessor, such as processing elements, memory system, and interconnection network, can be specifically tailored to the needs of the embedded application[2]. Specializing the hardware to the specific needs results in systems that need less hardware, which results in systems that are more power-efficient. A heterogeneously designed system also leads to better real-time performance because the designer can more precisely predict the timing behaviour of seperate parts of the system, compared to processes that all run on one and the same processor. Time-critical function-units can be moved to specially designed processing units. There are some pitfalls too in heterogeneous systems design, because if the embedded system is uncarefully designed the interconnection overhead between the different processing elements can result in a system that is slower and consumes more energy.

3-3-3 Hardware Accelerators

In terms of multiprocessing, hardware acceleration forms one of the simplest forms of multiprocessing. But it can be a very effective way of achieving improvements in the computing performance of an embedded system.

When a given embedded platform can't meet the required performance goals, it can be customized by using a specific circuit that implements part of the embedded application. When a small part of the application's code is responsible for a large part of the execution time then this application is a good candidtate for acceleration. This small part of code, which is called the kernel of the application, is then mapped onto some specific piece of hardware that communicates with the CPU over the bus. Such a circuit is called an accelerator.

From the point of view of the CPU, an accelerator can be considered an I/O device as far as communication is concerned. The accelerator connects to the CPU bus, and communicates with the CPU in the same way an I/O device would do. An accelerator can be implemented using one of several techniques, such as a custom designed circuit known as an ASIC, or an off-the-shelf component, or an FPGA that is programmed with the specific functionality, or a seperate CPU and software that together perform the required functionality.

There are two reasons why we would add an accelerator to the design of an embedded system. The first is that from a cost/performance perspective it often is beneficial to split the application over several processing elements [5]. The purchase price of microprocessors is a nonlinear function of performance. So, instead of increasing the performance of a microprocessor a bit, it is cheaper to use two microprocessors, or a microprocessor and accelerator, and dividing the application over these two processing elements.

The second reason is of course that the system without an accelerator is not fast enough to meet the real-time performance requirements, and that we need to boost the performance somehow. And in some cases adding an accelerator can do the job.

The total execution time of an accelerator that reads in the data, performs the required computation, and then writes the result back, is given by [5]

$$t_{accel} = t_{in} + t_x + t_{out}$$

where t_{accel} is the total execution time of the accelerated system, t_{in} the time needed by the accelerator to read in all data, t_x the computation time required by the accelerator, and finally t_{out} is the time it takes to write back the results.

We can improve upon this result if the accelerator can start computing on the data while it is still reading in the data, and the same is true for data that is written back. Then the values for t_{in} and t_{out} have to be adapted accordingly in the equation above.

The speedup is given by the following equation [5]:

$$S = n(t_{CPU} - t_{accel})$$

It can be interpreted as the time that is skipped or saved when an accelerator is used, compared to the time when the kernel would be executed in software. From this equation it follows that speedup is not exclusively dependent on execution time by the accelerator, but also on the

time it takes to move the data in and the results out of the accelerator. A kernel, where it is needed to move a lot of data, might not be a good candidate after all.

The design process for an accelerated system can be divided into several stages. The process begins by examining the application from a performance perspective, because the ultimate goal is speeding up the total design and we have to know in advance which part(s) of the system can best be accelerated. First the different tasks in the application are identified and then the execution time on the processor and the accelerator are determined. In order to be able to determine what the best partion is, we have to know what the times are to communicate the data between the processing elements. Once this is done, the next stage can be commenced.

An application can be thought of as being constructed out of functional units that together determine the overall functionality of the application which ultimately follows from the requirement specification of the application. When we design an accelerated system we have to identify the units that are best to be run on the processor and the units that are more suited to be run on the accelerator. This process is called partitioning. Partitioning is about determining the fitness of functional units for all processing elements, in this case the CPU and the accelerator. This must be done on the basis of a performance analysis. It is not the task of partitioning to come up with a definite assignment of units to the processing elements, that is the task for the next stage. Instead partitioning has to come up with all relevant assignments to processing elements based on their performance on that processing element.

The next stage is responsible for assignment of the functional units to the processing elements. Once it is known what the communication times and the execution times are on the different processing elements, we need to schedule the computations such that the different computations do not interfere with each other, due to dependencies and specific properties of the communication channel. Finally, an allocation is made based on the previous step and aiming at the shortest execution time possible considering communication between the functional units on different processing elements.

Finally, the last step in the design of an accelerated embedded system is to integrate the different parts of the system. Communication between the CPU and the accelerator needs to be tested and the interaction between the two processing elements needs to be tested as well.

Building an accelerated embedded system requires more design effort, and this has to be taken into account when deciding to split the application. From a cost perspective designing an accelerated system requires more design effort, which will result in a system that is more expensive.

3-3-4 Interconnection Networks and Memory Systems

Interconnection networks and memory systems become important aspects when designing distributed embedded systems, be it symmetric multiprocessor systems or heterogeneous multiprocessor systems. Results that have been computed in one processing element are needed in processing elements elsewhere in the system. For this we need connections between the processing elements, and these connections can take on many forms. The other aspect of distributed embedded systems is how the memory is organized. Questions have to be answered

whether all processing elements need access to this memory or that it suffices that we keep the memory local to some processing elements. We will next discuss aspects of networks in multiprocessor systems and finally the design of the memory system is discussed.
The collection of connections between the processing elements and memory blocks is called the interconnection network. The structure of these networks depends on the bandwidth requirements of the system. Familiar networks are the bus, crossbar and mesh networks. Each has its connection properties at the exepense of energy performance and area costs. The designer has to carefully examine the specific needs of the embedded system. Sometimes it is beneficial to design local networks that are exactly matched to the specific requirements at that location, thereby saving on area and power usage.

A special type of network is the so called network on chip (NoC). The NoC is completely integrated on the chip with the rest of the processing elements. Advances in technology have made it possible to integrate complete networks on chip. This makes it possible to acquire large improvements on latency, energy consumption, and production costs.
In their effort to meet the tight demands for real-time performance and energy consumption, designers have a number of options when it comes to the design of the memory system. Real-time performance behaviour of memory blocks depends in large part on the size of the blocks and the number of processing elements that have access to it. By minimizing the number of processing elements to the number of processing elements that need access to it, we can improve the real-time behaviour of the system. In the extreme case when a processing element must meet tight real-time constraints it can be decided to use a memory block exclusively for just this processing element and thereby guaranteeing the real-time performance of the processing element. So, the size and number of connections to the memory block are important design parameters in meeting the design constraints.

Energy consumption of memory blocks depends in large part on the size of the memory block. So, by choosing smaller blocks to improve real-time performance we automatically improve upon the energy consumption characteristics of the memory blocks.

A design methodology that enables the designer to determine an optimal memory configuration and accompanying network type for an embedded system is described in [6]. The size of the memory blocks is determined from the clock frequency of the system, which in turn determines together with the needs of the processing elements the number of memory blocks. Finally a network type is chosen that meets the connection requirements of the processing elements. The proposed methodology is iterative in nature, if the acquired results do not meet the requirements then decisions that were made in earlier steps have to be reconsidered.

3-4 VLIW Architecture Approach

In this section we describe a type of processor architecture that turns out to be a close match for embedded systems. In section 3-4-1 we explain the properties of VLIW processors and in section 3-4-2 we explain why VLIW processors are the preferred choice to exploit available ILP in an application.

3-4-1 VLIW CPUs

Very Long Instruction Word (VLIW) processors are widely used in embedded system design because of their properties. A VLIW processor executes several operations concurrently. These operations are contained in the instruction that is read from memory. All operations are executed simultaneously and the next instruction is only executed when all operations of the previous instruction have been executed. A VLIW processor executes the instructions in the order they appear in the program, it does no dynamical rescheduling and it does not issue more than one instruction at a time. A block diagram is shown in figure 3-1. After an

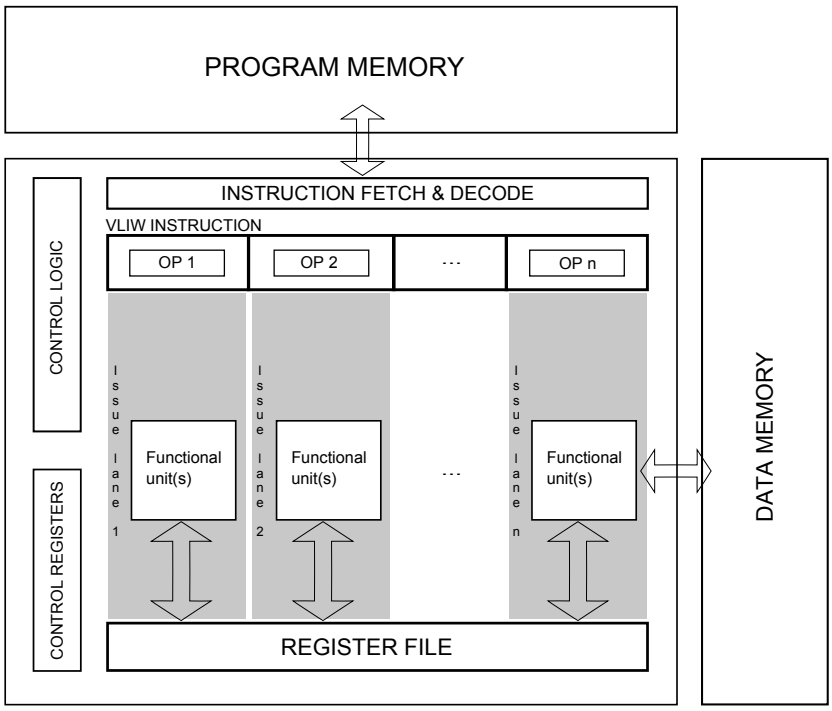


Figure 3-1: A block diagram of a VLIW architecture.

instruction is fetched and decoded, its operations are assigned to the proper functional units by putting them into the corresponding issue slots. In the figure these issue slots are the rectangular boxes containing the texts op1, op2, etc. The process of assigning the operations to the issue slots is called dispersal of operations. The operations that have been assigned to the issue slots are then executed by the proper functional unit that is attached to the issue lane. By changing the number of issue lanes and consequently the number of operations in the instruction the designer can vary the amount of ILP that is exposed by the processor.

The operations are not scheduled by the processor, it only executes the operations that are present in the current instruction. Hence, the job of scheduling the operations is completely handled by the compiler. It is the job of the compiler to extract all available parallelism in the application. The compiler is also responsible for another aspect of scheduling, namely it has to take into account the dependencies that exist between the operations and schedule them accordingly. Needless to say this requires a powerful state of the art compiler. Because the compiler does most of the work in scheduling the operations, the execution logic of the processor can be kept simple.

The advantage of VLIW processors is that they require not much hardware for issueing and scheduling of instructions, but there are some disadvanteges in the design of VLIW processors too [7]. First, by increasing the issue width of the processor it is necessary to increase the number of ports on the register file and to increase the memory bandwidth. This results in increase of silicon area for the register file and possibly a reduction in clock frequency. Second, when there are more operations in an instruction the compiler has a more difficult job of filling the issue slots of the instruction. It has to find more ILP, and it can do that by more agressive loop unrolling for example. Third, the code size for VLIW processors tends to increase because the compiler can not always fill the instruction and applies loop unrolling which increases the size of the code. Finally, there are issues of code compatibility. When it is decided to change the issue width, or to add execution units, the new processor is no longer binary compatible with the old code for the previous processor design, requiring recompiling of the code.

The next section discusses the relation between ILP and VLIW architectures more closely. It describes why VLIW processors are such a good candidate for embedded system design.

3-4-2 ILP and VLIW Processors

When instructions are carried out in a sequential fashion each instruction is completely carried out before the next instruction is started. Execution of an instruction can take up several cycles, so the average number of instructions per cycle is low. Sequential execution of the instructions that together form a program leads to an execution time of the program that can be reduced significantly by applying the principles of instruction level parallelism (or ILP).

The analogy often used is the assembly of cars. In the early days of car assembly, cars were typically built by one person. The task of building a car can be split up into many subtasks, and when one person is assembling a car he will carry out these subtasks one by one, taking days to complete one car. But the introduction of the assembly line changed the production of cars completely. An assembly line consists of many terminals, where each terminal is manned by one or more persons. The car travels along those terminals, where at each terminal some task is carried by the persons working at that terminal. Work is done simultaneously at these different terminals, making it possible to produce a car every 15 minutes.

The same principles apply to the execution of instructions of a program. The task of executing an instruction can be divided up into several sub tasks that can be carried out simultaneously just as with the assembly line. We call the organization of this type of execution a pipeline. A pipeline has several stages, each carrying out a sub task of the instruction. Using pipelining leads to a significant speedup, but it also exposes some problems in the organization of the processor's hardware that need to be taken care of.

Instructions that are executed in sequence have exclusive access to all resources that are needed for the execution of just this one instruction. But when there are several instructions in the pipeline, instructions can contend for the same resources, for example when two instructions need to access the memory at the same time. Another example of a situation that can produce problems is when an instruction is dependent on the result of an instruction that is still in the pipeline. So it is clear that the processor's hardware needs to be adapted to the new situation.

Pipelining and all the aspects of hardware organization is what ILP is all about. ILP is about identifying all the sub tasks in the execution of instructions and devising ways of executing these sub tasks simultaneously as much as possible. We distinguish the following aspects of a processor's hardware that enable ILP.

1. Hardware carrying out arithmetic and other operations needs to be reorganized such that it is able to carry out several sub tasks simultaneously. That way several instructions can be operated on at the same time without stalling the pipeline.

2. Operations are carried out in different parts of the hardware and this needs to be made possible. For example, an instruction is writing its results back to the register file while at the same time the next instruction is fetched from memory.

3. More than one instruction can be issued at the same time and requires careful design of the hardware. Architectures that issue more than one instruction per cycle are called superscalar, or muli-issue architectures.

4. When multiple instructions are issued, functional units need to be replicated. For example, when there are two independent floating point instructions they can each be issued to another floating point unit.

There are in general two ways ILP can be implemented. The first is letting the processor decide dynamically, i.e. during execution of the program, which instructions are issued at the same time and resolving all dependencies. This type of processor is what is used in general purpose computing and is known as the (superscalar) RISC design style. The other way of dealing with ILP is letting the compiler decide at compile-time what instructions are issued simultaneously and letting the compiler take care of dependencies between instructions by proper sequencing of operations. Letting the compiler do all the work is the VLIW design style. The VLIW design style exposes ILP explicitly in its architecture. It applies to many levels of the architecture, including the instruction-set architecture (ISA), the microarchitectural hardware, and the compiler.

Improvements in transistor budgets and compiler technology have made it possible to use VLIW machines specifically in the embedded systems market where performance, cost, power and size constraints are of importance. Currently VLIW architectures are used in the high end of the embedded computing market but it is expected that in time this will become available for the low end embedded market as well.
There are good reasons why embedded systems designers use the VLIW approach in their designs [8]. First, VLIW processors require less control hardware as compared to superscalar processors. The area that control hardware occupies on the die can become very significant in superscalar processors, leading to high costs of production and energy consumption. These factors are important in embedded systems design, giving the VLIW approach an advantage over superscalar processors.

Another reason is that applications that are typically used on embedded systems have a program structure that is often very regular and contain a lot of ILP. In most cases it is very easy to extract this ILP at compile time and thus the ability of general purpose processors to extract ILP dynamically becomes less important [9].

Embedded system designs are often designed from scratch and code written for previous designs is often not reused. So, object code compatibility is not as important as it is in general purpose computing where new processors need to be able to execute legacy code.

Finally, VLIW processors are much easier to scale and customize than general purpose processors. A property of VLIW processors is that they have a very regular architecture, their functional units and register bank components can be easily modified and replicated.

3-5 Hardware/Software Co-design

Standard general-purpose architectures are hardly ever sufficient in meeting the design constraints of embedded systems, such as real-time performance, low energy consumption, and costs. Almost always the embedded system designer needs to make modifications to the hardware and consequently the software of the embedded application. It has been recognized early in the history of embedded system design that it is necessary to design both hardware and software concurrently. The methodology that has been developed for this is called hardware-software codesign.

Hardware-software codesign is a collection of design techniques that aid the designer in making the right design decisions to arrive at an optimized embedded system with regard to the above mentioned design constraints, real-time performance, low power consumption, and costs. Hardware-software codesign almost always leads to heterogeneous systems because this gives the designer the most freedom in choosing the right components for the different tasks of the application.

In order for the heterogeneous design to be effective it is absolutely necessary for the designer to know the application very well. Only then is he in the position to make the most effective modifications to hardware and software. But knowing the application is not enough.

Although enlarging the design space gives more freedom to choose from different configurations, it poses some challenging problems as well. Because the set of potential designs is so large, searching for the right design configuration can become a challenging task. It is necessary to strike a balance between efficiency of finding the right candidate and the accuracy of searching for the right candidate. The time the embedded system designer can take for the design of the embedded system is bounded by time-to-market constraints and other factors that have an influence on the design time. The designer needs a method that gives him quick answers to such questions as: 'should a certain task be implemented in hardware or software', 'what are the costs of a certain decision', 'what does it mean for the real-time performance'. Several techniques and methods have been developed that give quick answers to these questions, but a discussion is beyond the scope of this thesis.

It has been criticized that current hardware-software codesign approaches lack an abstract formal design methodology in which the design can be described independently of the implementation details. One of the proposals that address this problem uses a modified form of finite state machines (FSM), called CFSMs, as formal descriptions of the tasks of the application [10]. The applications's tasks are mapped to these CFSMs and necessary connections between these CFSMs are made, forming a CFSM-network. This network completely models the embedded application. The synthesis system of this design approach can produce several hardware and software representations for each task that are directly derived from the CFSM

that represents this task. Cost and speed metrics are described in terms of these representations, making a cost and/or speed driven design possible at this abstract level without having to translate these representations to some implementation. The designer can choose configurations of software and hardware representations for the different tasks of the application and determine what this will mean in terms of costs and performance. One of the big advantages of this formal methodology, and which was an explicit desire of the designers of the methodology, is that this methodology makes formal verification of the designs possible. A desirable feature that is made possible with this methodology, but is not yet implemented, is that of automated constrained-driven partitioning of a description of a system into a hardware-software system.

There is a widespread agreement among members of the embedded systems design community that hardware-software codesign is the best design methodology for embedded systems design. This methodology is the answer to the question on how to design embedded systems that can meet the severe design constraints that apply to embedded systems.

However, a characteristic of hardware-software codesign is that the design configurations that the designer has to consider becomes overwhelmingly large very rapidly, even for moderately sized design problems. This limits the practical use of this methodology and the only way a designer can handle this manually is by pruning the design space to manageable proportions thereby reducing the search to suboptimal design solutions. But there is another approach to solving this design problem.

Several research groups are working on projects that have as goal to automate the process of hardware-software codesign. One of those groups is working on the hArtes (Holistic Approach to Reconfigurable Real-Time Embedded Systems) project. The aim of this project is to provide designers with a toolset that automates or semi-automates the hardware-software codesign process [11].

The toolchain allows the designer to take an application written in the C language and use this as input to the toolchain. The toolchain then automatically finds a partitioning and mapping of the application to the available hardware. The designer does not need to know anything about mapping parts of the application to hardware, this is all being taken care of by the toolchain. The designer can, however, use the toolchain in a semi-automatic fashion, if he so wishes, by indicating which functions need to be mapped to hardware, or which functions can be performed in parallel. This is being done by adding pragmas in the source code of the application. The benefits for the designer are that he does not need to know anything about hardware mapping, and time-to-market is reduced substantially.

3-6 Programs

Programs form an essential part of embedded systems. Several techniques exist to improve real-time performance, as well as other constraints such as cost and energy consumption. It is therefore important for the designer to pay attention to the design and implementation of the programs. Compilers play an important role in software optimization and this will be discussed after we have explained what the relation is between software and real-time performance, cost and energy consumption.

3-6-1 Relation between software, real-time performance and cost

The size of memory in embedded systems is often limited and it is important that the software fits in the available memory. It is therefore important that the size of programs be kept as small as possible.

The execution time of programs is directly related to real-time performance, which is an important design parameter in embedded system design. The execution time of a program is determined, among other things, by the CPU and related hardware, the quality of the source code, and the compiler. The characteristics of the CPU and the way the hardware platform is designed determine for the large part the execution time. The CPU can be a simple microcontroller, or a sophisticated superscalar CPU that issues multiple instructions each clock cycle. When a superscalar CPU is used, it is important that there is enough available ILP. High level optimization by the compiler can produce compiled code from which the CPU can extract more ILP. In the next section it will be discussed which compiler optimization techniques are available.

3-6-2 Compiler Optimization Techniques

There are several performance improving techniques that are used by compilers to optimize the code. Procedure inlining is such a technique, and another technique that is used by compilers is transformation of loops.

When a procedure is called in a program there is some call-related overhead such as:

1. Creating space on the stack, called an activation record, that contains the arguments for the procedure that is going to be called, and some other information that is used to keep track of the activation record and to know where to return to when the procedure returns to its caller.

2. Pushing the return address on the stack and jumping to the body of the procedure, i.e. the actual call.

When such a procedure does not contain a lot of code the overhead of calling the procedure can become quite substantial. The compiler can then substitute the code of the procedure body at the place where the procedure is called in the code, thereby saving the overhead of calling the procedure. This optimization method is called procedure inlining.

Other important optimization techniques concern loops. Loops are a very common programming construct in which a program can spend a large fraction of its execution time. Several loop transformation techniques exist. One is loop unrolling. The body of the loop is executed several times and when the body of loop is small, the overhead of branching to the beginning of the body to execute the next iteration of the loop can be quite substantial. By copying the body of the loop several times, which is called unrolling the loop, the overhead of branching to the start of the body can be reduced substantially. Other loop transformation techniques aim at a better usage of the cache. These techniques modify the body of the loop such that the data fits the cache-lines better.

3-6-3 Improving code performance

Code performance depends on many factors:

- Performance depends very much on the efficiency of the code written in the high level language.
- Compilers are able to optimize the code and often the level at which the compiler should optimize can be chosen through an option at the command line.
- Performance depends very much on the CPU and memory system.

The programmer can do much on improving the code so that it performs better. It starts by choosing the right algorithm for some problems. Often different algorithms exist for the same problem and they do not all have the same computational costs. Hence the importance of spending time searching for the best algorithms. The programmer should find the places in the code where the processor spends most of its time, that are the places where most can be gained by optimization of the code. Loops are perfect candidates for code optimalization, and there are several techniques that can be used by the programmer to optimize loops.

When the programmer has done his best to optimize the code at the high language level it is up to the compiler to even further optimize the code. The strength of compilers has increased at a steady level over the past years when it comes to performance improvement of code. The compiler can optimize the code at increasing levels of performance optimization. Code optimization can have side effects. Some optimizations increase the code size a lot and this can be a significant penalty in some systems, especially in embedded systems where memory constraints are often severe.

CPUs vary from simple microcontrollers to sophisticated superscalar processors, issueing multiple instructions per clock cycle. By reordering the code at the instruction level it is possible for the CPU to find more ILP. Another technique often used in compilers is that of loop unrolling. That way the CPU is able to find more ILP, because of the larger loop body. But, the memory system can severely influence code performance as well. If the code is written in a way that causes a lot of cache misses, the cache lines have to be retreived from memory every time and main memory is much slower than cache memory causing the performance of the program to decrease significantly.

Optimizing code is not as straight forward as one might want. CPUs and memory systems have become increasingly complex and hence the relation between the code and it its performance is not always obvious. Compilers have become very powerful at optimization and they are in fact the only way large programs can be optimized in a practical way.

Chapter 4

Minutiae Extraction Application

In this chapter, some basic information about the application to be accelerated in this project will be given, i.e. fingerprint minutiae extraction application. The application discussed in this chapter is originally coming from the National Institute for Standards and Technology (NIST). This chapter starts by giving some background information on the fingerprint minutiae extraction application within section 4.1. Section 4.2 deals with the minutiae extraction process and the steps in this process will be explained in short. Section 4.3 deals with the image file format used in this project and a brief explanation about it. The motivation for choosing the software system in this project is explained in section 4.4. Finally, the chapter will be finished with conclusions in section 4.5

4-1 Background

The fingerprint of an individual is unique. It means that each person has a different fingerprint and it remains unchanged over his/her lifetime. Therefore, fingerprint verification is one of the oldest form of biometric techniques to verify the identity of a person. Namely, fingerprints haven been used for over a century for identity verification. This fashion of identification is used in forensic science to support criminal investigations and in biometric systems such as civilian and commercial identification devices [12].

A human fingerprint is formed by a pattern of ridges and valleys. As it is also clear from figure 4-1, a ridge is a single curved segment and a valley is the region between two adjacent ridges. The identification verification is done by comparing two human fingerprints. This comparison is realized using discrete features which are called minutiae. The minutiae are defined as the local discontinuities in the ridge flow pattern. It provides the needed features to verify the identity of a person and these features are stored in the application for each detected minutiae. These features are mostly the position and the orientation of the minutiae but the quality of the fingerprint is also an important feature in order to get a reliable result from the comparison of the fingerprints. The position of a minutiae is stored as a pair of (x,y) coordinates. These are the coordinates where the end-points or start-points from a detected

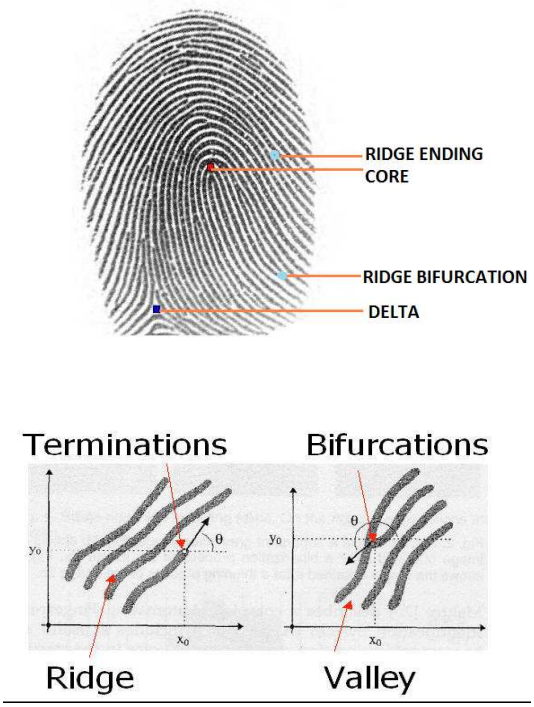


Figure 4-1: Minutiae in a Fingerprint image [13]

ridge are located in the fingerprint. The orientation is the angle between the tangent to the ridge line at the minutiae position and the horizontal axis, see figure 4-1. The quality of the fingerprints, thus quality of the fingerprint images, is also important because fingerprint images with poor quality results in false minutiae, so in the loss of true minutiae.

4-2 Minutiae Extraction Process

In [14], it is concluded that the minutiae extraction process can be seen as an image pipeline and the extraction process can be divided in three phases ; pre-process, minutiae detection and post-process. Pre-process consists of the following three steps ; image contrast enhancement, image quality analysis and binarization respectively and post process consists of the following two steps; false minutiae removal, minutiae quality assessment respectively. Now a brief explanation about each step of the minutiae extraction process, shown in Figure 4-2, will be given below. For more details about the process the reader is referred to [13].

4-2-1 Image (Contrast) Enhancement

Image contrast enhancement is the first step in pre-process phase and it is done in order to improve the contrast of a fingerprint image by executing an image enhancement algorithm. In doing this, a histogram of the fingerprint image is evaluated. This histogram is a plot of the pixels and it consists gray scale values or intensities. The enhancement is done in the case that the pixel values or intensities are not well-distributed over the histogram. Otherwise there

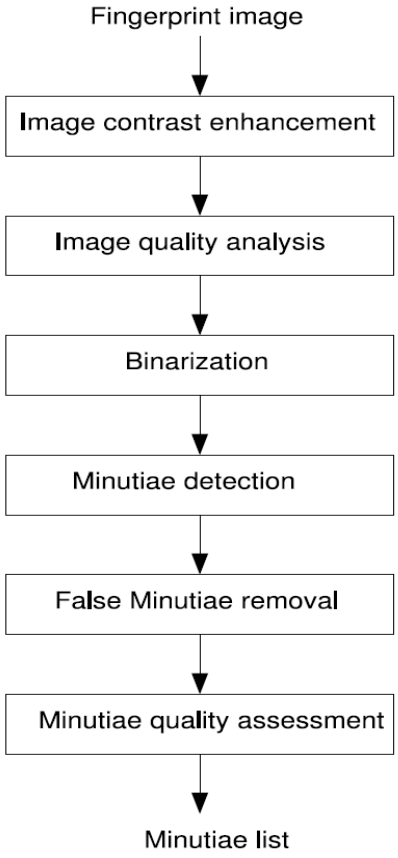


Figure 4-2: Image processing phases

is no enhancement done. Thus, enhancement is done by evaluating the pixels distribution intensity.

4-2-2 Image Quality Analysis

As it is already denoted above, the quality of a fingerprint image is important in order to get a reliable result. Mostly, fingerprint images are not of good quality, so it is needed to improve the quality of these images. In this step, the work is to evaluate and decide whether there are some regions of the image that are degraded or noisy and whether these regions are located on the image that will be used in the detection phase. This is important to prevent any problems in the detection phase. This step can be further divided in four sub-steps; low contrast analysis, direction analysis, direction map post-processing and high curvature analysis, respectively.

Low contrast analysis

It is too difficult or even impossible to detect minutiae in some parts of the image because of the unclear defined ridges. These unclear defined ridges are located at some parts such as image background or the smudge section. Regions with low contrast can also cause a problem in the detection phase. Therefore, these regions have to be improved properly, too. In this step, a low contrast map is generated by the low contrast analysis algorithm. The low contrast areas are marked in this map. This low contrast analysis algorithm also solves the problem with the background by segmentation. By segmentation, foreground is separated from the background and all other low contrast areas in the image are mapped out. This analysis is performed on image blocks and makes use of the reality that there is little dynamic range in the pixels intensity of the problematic regions described above. An image block is marked as low contrast if it is determined that there is little dynamic range in the pixel intensities in that image block.

Direction analysis

In order to detect minutiae reliable, it is essential to have well-formed clearly visible ridges. Hence, it is needed to know which areas of the image possess sufficient ridge structure. Therefore, a direction map is generated in this step that represents these areas. Also the general orientation of these areas are included in this map for every block. These are blocks of 8x8 pixel by default. More information about how to estimate the orientation and the implementation of the blocks can be found in [15] and [13], respectively.

Direction map post-processing

The task in this step is to improve the quality of the direction map that was generated in the previous step. This is achieved by removing the voids from the direction map. There are two basic types of techniques, called erosion and dilation, used in this step. These techniques are known as the most basic morphological operations. Dilation adds pixels to the boundaries of the objects in an image and erosion removes pixels on the object boundaries. Thus, by

dilation, the mapped regions of the blocks in the directional map become larger and by erosion these regions decrease.

High curvature analysis

Areas with high curvature such as the core and delta regions of the finger can also cause problems for detecting the minutiae reliably. There is also a map generated in this step like in the previous steps and it is called a high curvature map. Two different measures are made in this step; vorticity and curvature. Vorticity is defined as the cumulative change in the ridge flow direction between the block that is analyzed and its neighbor blocks and curvature is defined as the largest change in the direction of the ridge flow between the block that is analyzed and its neighbors. The algorithm used in this step determines whether a block in high curvature map is high curved or not, based on these two measures. Details of the algorithm used can be found in the source code.

4-2-3 Binarization

Binarization is the conversion of the gray-scale image to a binary image. This step is needed because the extraction algorithm can only work with a binary image as its input. The direction map generated in direction analysis step is used by this extraction algorithm in order to determine whether the pixels in a block are white or black. The pixels in a block are set to white when there is no direction of that block. Determining whether the pixels in a block have to be set to black is a much complex process as it requires some intensive mathematical computations. The reader is referred to [14] for more information about these computations. Furthermore, binarization is the last step of pre-process phase and will be followed by minutiae detection phase. Figure 4-3 depicts the binarization results for a typical fingerprint image.

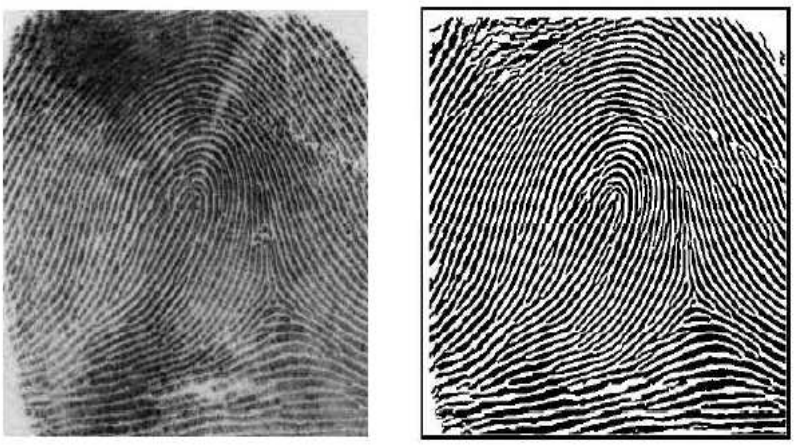


Figure 4-3: Typical fingerprint image (left) and its binarized version (right) [13]

4-2-4 Minutiae Detection

The image that is undergone the pre-process phase has now become suitable for minutiae detection. There is used a pattern matching algorithm in this phase as it is all about pattern matching in minutiae detection. The task of this algorithm is identifying the ridge endings and bifurcations. It performs a horizontal and vertical scan of the binary image from the binarization step of pre-process phase. During this scan, pixel pairs are matched with 10 other patterns that are already stored in the system. Ridge endings are identified by comparing the current pixel pair with only 2 patterns while the rest of the patterns are used to identify the bifurcations. Figure 4-4 depicts the 10 patterns used by the algorithm. The direction

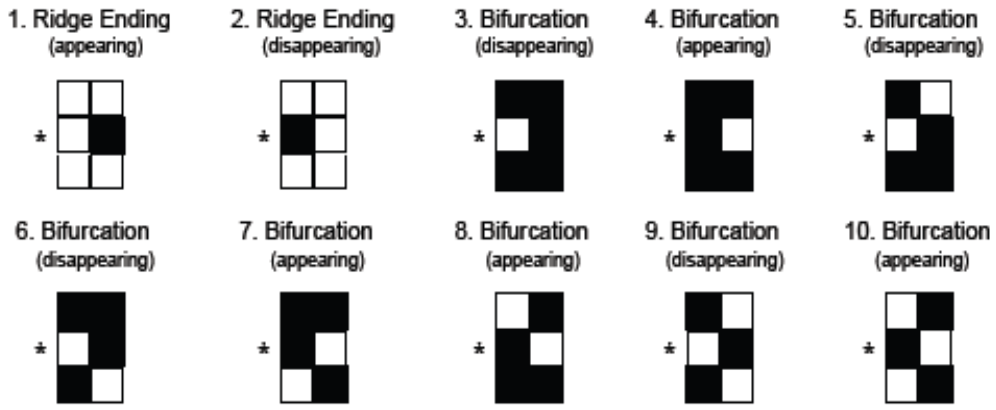


Figure 4-4: 10 patterns used by the algorithm to detect minutiae [13]

of a ridge or valley can also be specified for minutiae detection. This is called disappearing or appearing depending on the direction that the ridge or valley follows in the pattern. The horizontal scan is done to detect all vertical pixels and vice versa.

4-2-5 False Minutiae Removal

False minutiae removal is the first step in post process phase. In this step, the false minutiae produced by the minutiae detection algorithm have to be removed because this has undesired effect on the performance of the system. Actually there is a small chance of missing the true minutiae, however, many false minutiae are also produced. The fact that the pattern matching utilizes patterns of 6 pixels is the main cause for the production of these false minutiae [13]. The system performance decreases in the presence of false minutiae because minimizing these false minutiae costs much effort and code. There are used many algorithms in order to remove hooks, lakes, holes, islands, side minutiae, overlaps, too wide or to narrow minutiae and low quality minutiae. The description of how all these algorithms work is beyond the scope of this thesis, more information can be found in [13] or in the source code.

4-2-6 Minutiae Quality Assessment

Although the minimization of the false minutiae, not all of the false minutiae are completely removed from the system. There will still be some false minutiae present in the final minutiae list. These residual minutiae are assigned a quality factor ranging from high (4-5) to low (0) in order to identify them. Assigning the quality factor happens in a quality map. This quality map is generated based on the the information of the low contrast, low flow and high curvature maps. As it is logical , a false minutiae gets lower quality factor than the true one. Furthermore, the minutiae quality is determined using the following measures; the position of the minutiae in the quality map and intensity statistics such as the mean and the standard deviation within the direct neighborhood of the minutiae. A high quality factor is assigned to a minutiae if the mean is close to 127 and the standard deviation is larger or equal to 64 pixels.

4-3 The WSQ File Format

As it is denoted earlier in this chapter, the used application in this project is coming originally from NIST. There are several fingerprint databases distributed by NIST in order to use in fingerprint matching systems and the fingerprint images in these databases are formatted in different formats. The file format used as input image in this project is in WSQ file format which is also adopted by FBI. WSQ stands for Wavelet Scalar Quantization and it compresses gray-scale fingerprint images. An advantage of this file format is that it is optimized for fingerprints. WSQ keeps the very high resolution details of grayscale images and maintains high compression ratios. For other types of compressed file formats such as jpeg and tiff, both the quality of the image and important information would be lost and it would become unreadable by an Automated Fingerprint Identification System (AFIS).

4-4 Application Motivation

As it is denoted earlier in this chapter, the application used in this project is originally coming from the National Institute for Standards and Technology (NIST). It is a modified version of the MINDTCT software package by NIST and originally developed for FBI. However, the source code and execution time in this original version from NIST are too large and contains too much overhead. Therefore, a stripped down version of the fingerprint application has been chosen from The Circuits And System Research Group of TU Delft as the application to be accelerated using VEX processor.

The following four criteria was the main reason for choosing this application in this project :

- Fixed-point : As the VEX ISA uses fixed-point arithmetic for the computations, the original version of the MINDTC extraction algorithm from NIST can not be used in this project because it uses floating-point arithmetic for computations. It is determined that floating- point version cost much more execution time than the fixed-point version.
- Stripped : The fixed-point version of the fingerprint application was stripped down from some non essential codes in order to decrease the execution time.

- Optimized : The source code was optimized by utilizing look-up tables for the binarization and for computing the sine and cosine values required by the Discrete Fourier Transformation computation.
- C-program : The software package was written in the C language.

4-5 Conclusions

In this chapter, the minutiae extraction application and the extraction process is discussed. The chapter began with a section involving the background information on the application where it became clear that fingerprint verification is one of the oldest form of biometric techniques to verify the identity of a person. The position and the orientation of the minutiae and the quality of the fingerprint are the needed features in order to verify the identity of a person. The identification verification is done by comparing two human fingerprints.

As it is denoted in section 5-2, the minutiae extraction process can be seen as an image pipeline which is the most computationally intensive part during the verification. Because it can be seen as an image pipeline, the process can be divided in three phases ; pre-process, minutiae detection and post-process. Also these phases can be further divided in several steps which results in six image processing steps. The first two steps provides a fingerprint image with high quality by doing contrast enhancement and image quality analysis. The latter is to evaluate and decide whether there are some regions of the image that are degraded or noisy and whether these regions are located on the image in order to prevent any problems that can arise in the detection phase.

In the third step, binarization, the gray-scale image is converted to a binary image as the extraction algorithm can only work with a binary image as its input in the detection phase. In minutiae detection phase, it is all about pattern matching. The work to be done in this phase is identifying the ridge endings and bifurcations by performing a horizontal and vertical scan of the binary image. During this scan, current pixel pairs in the image are matched with 10 other pre-stored patterns. Because many false minutiae are produced in previous phases, these false minutiae are removed in the fourth step and in the last step the detected minutiae are assigned a quality factor before storing the final minutiae list.

Also the motivation for choosing the modified version of the fingerprint verification application has been given in this chapter. This application is chosen because it is a stripped, optimized, in C language written and fixed-point utilizing version which is quite suitable to use in this project. It is stripped down so that some non essential code are removed from the software. It is optimized by utilizing look-up tables for the binarization and for computing the sine and cosine values for DFT computation. It uses fixed-point arithmetic for the computations which is an important requirement in order to be able to work with VEX ISA. Furthermore, it is given that WSQ is the input image file format used in this project as it is well-suited and optimized for fingerprints. It is widely used by FBI and has the advantage of keeping the very high resolution details of gray-scale images and maintaining high compression ratios.

Chapter 5

The Embedded Computing Platform

As it is common there is no design or project to be realized without making use of some equipment, materials or tools. In this project, there are number of hardware components used which can be named as the building blocks to realize the aim of the project. This hardware is described in detail in the following sections. In section 5.1 The Virtex-6 FPGA board has been represented with some important features of it such as compatibility across its sub-families and flexible configuration options. Section 5.2 deals with the MicroBlaze soft core processor which is designed by Xilinx. Technical properties and some important features of this processor are discussed in detail in this section. Section 5.3 describes the VEX system which consists of three basic components. These components are explained further in this section. In section 5.4 the instruction set architecture is discussed. In section 5.5 the reader can find the implementation of The VEX. In this section a previous version of the VEX and the current implementation of it has been covered. Section 5.6 deals with the VEX tool chain which includes a C compiler and a simulation system. In this section more is given about the compiler; i.e its behaviour, its capabilities etc. This chapter ends with a section that describes the complete system, the so-called hardware platform, which describes the connection of the components described in the previous sections of this chapter.

5-1 Virtex-6 FPGA Board

An FPGA board from Virtex-6 FPGA families is used in this project which is produced by Xilinx. It is defined as the programmable silicon foundation for Targeted Design Platforms with integrated software and hardware components in these platforms. This board has some advantages above the previous generations of it. These can be listed as follow:

1. Up to 50% higher performance through hard memory controller and six-input look-up tables (LUT) architecture that can be configured as either 6-input LUT with one output or as two 5-input LUTs with separate outputs but common addresses or logic inputs,

2. Up to 50% lower cost and power due to advanced process technologies and integration of multiple dedicated IP blocks,
3. They deliver breakthrough I/O performance with the integration of high speed serial transceivers [16].

There are three sub-families of Virtex-6 FPGA boards that contain many built-in system-level blocks; Virtex-6 LXT FPGAs, Virtex-6 SXT FPGAs and Virtex-6 HXT FPGAs. The first one has the feature of high-performance logic with advanced serial connectivity, the second one has high signal processing capability with advanced serial connectivity and the last one delivers high bandwidth serial connectivity. Furthermore, Virtex-6 FPGA offers compatibility across sub-families and has flexible configuration options. This FPGA uses SRAM-type internal latches to store its (customized) configuration and the number of configuration bits is between 26 Mb and 160 Mb. Another advantage of Virtex-6 is that it supports partial reconfiguration which is considered to improve the versatility of the FPGA. The reader is referred to [17] to read more about this FPGA board.

5-2 MicroBlaze

The MicroBlaze is a soft core processor designed by Xilinx. It is a reduced instruction set computer (RISC) and its core is a Harvard architecture core. It is optimized for implemention in Field Programmable Gate Arrays (FPGAs) such as Spartan-6 and Virtex-6 which belong to Xilinx, too. High configurability of this processor allows the designer to select a specific set of features required by its design. Some important fixed features of the processor are given as follows:

- Thirty-two 32-bit general purpose registers
- 32-bit instruction word with three operands and two addressing modes
- 32-bit address bus
- Single issue pipeline [18]
- Harvard architecture

Furthermore, this processor is also parametrized so that the designer can select and enable the additional functionality that it needs.

In figure 5-1 below, an example of MicroBlaze system for Spartan-6 and Virtex-6 is given [19].

MicroBlaze has the following hardware supported data types ; word, half word and byte. This processor represents the data in Big-Endian bit-reversed format. In this format, the most significant value in the sequence of bytes is stored first, i.e. at the lowest storage address.

This processor has a many-sided interconnect system to support a various range of embedded applications. It has the CoreConnect Processor Local Bus (PLB) as its primary I/O bus and Local Memory Bus (LMB bus) to get access to its local memory. The latter has the advantage

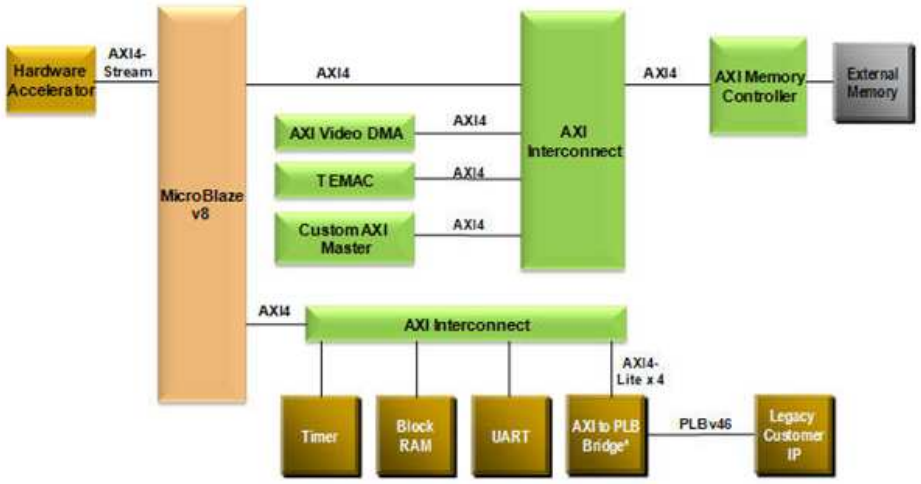


Figure 5-1: MicroBlaze system for Spartan-6 and Virtex-6 [19]

of reducing the loading on other buses. More information about PLB will be given in section 5.6.

Features like cache size, pipeline depth (3-stage or 5-stage), embedded peripherals, memory management unit and bus-interfaces can be customized within this processor. The performance of this processor depends on some factors such as the configuration of the processor, architecture of the target FPGA and speed grade. There are two versions of The MicroBlaze: area-optimized version using 3-stage pipeline and the performance optimized version with 5-stage pipeline. Another important property of this processor is that instructions such as multiplying, dividing and floating-point operations can be added and removed selectively. As it is known, these instructions are used seldom and they are expensive to implement in hardware.

5-3 The VEX System

The VEX system, which encompasses the ISA and toolset, is used as an educational instrument to support the book on embedded computing [9]. It was derived from the HP/ST Lx (ST200) architecture. The reason for choosing the Lx architecture is that Lx was a very clean new architecture, with no legacy to support, robust tools, and extensive scalability and customizability features, and therefore was the perfect candidate for educational purposes. Some adjustments had to be made to make it better match the tutorial requirements, and the result was called the VEX.

The VEX system is comprised of the following three basic components [9]:

1. *The VEX Instruction Set Architecture.* VEX defines a 32-bit clustered VLIW ISA that is scalable and customizable to individual application domains. VEX provides the user with the ability to change some aspects of the architecture, such as the number of clusters, the number of execution units, the number of registers, and latencies of the functional units. Customizability means that the user can add user defined instructions which can be convenient in applications that require specific operations.

2. *The VEX C Compiler.* The C compiler that came with the HP/ST Lx (ST200) architecture formed the basis for the VEX C compiler. It is an ISO/C89 compliant compiler that uses contemporary techniques for its global scheduling. The compiler allows the user to configure the target VEX architecture by specifying a configuration file, which enables the user to change the number of clusters, execution units, issue width and operation latencies.

3. *The VEX Simulation System.* The VEX simulator uses the VEX ISA specification to simulate execution of the instructions. Implementation details of a specific implementation are not considered by the VEX simulator, only what has been described in the VEX ISA is used in the execution of instructions. The execution of VEX instructions is translated into the host's native instructions, making the simulation as fast as possible on the host system. The simulator allows the use of most of the familiar C-libraries.

The processor is called the VEX which stands for 'VLIW Example', and is intended for experimental use. VEX is based on production tools used at Hewlett and Packard (HP) Labs and other laboratories. Its architecture is based on the Lx/ST200 ISA developed by HP and STMicroelectronics [8]. It is real technology, but is simplified for instructional use. The C compiler and the rest of the toolchain will be descibed in more detail in the section on the toolchain. The toolchain has been modified at the Delft University of Technology. The original toolchain does not produce an executable in native VEX binary code, and this and many other aspects have been modified in the new toolchain [20].

5-4 The VEX Instruction Set Architecture

The VEX ISA defines a flexible and scalable architecture for VLIW processors [9]. Aspects such as the issue width, the instruction set, and number of functional units can be modified. Because the compiler is responsible for the scheduling of operations, special attention has been payed to features of the VEX ISA that provide the compiler with greater flexibility in the scheduling of operations. Only structure and behaviour are specified with the VEX ISA, most implementation and microarchitecture details are excluded from this specification. Because VEX is a flexible architecture, two types of constraints are distinguished. The set of rules all implementations have to obey and the set of rules of a specific VEX instance. For the former we can think of the base ISA, register connectivity, memory coherency and architecture state. For the latter we can think of issue width, number of clusters, combination of functional units, latencies, and custom instructions. A VEX instance is obtained by implementing the basic architecture of the VEX ISA and by varying on the customizable parameters of the VEX ISA, such as the issue width, the number and composition of functional units, and so on.

VEX operations are very similar to RISC-style operations. They are called syllables in VEX parlance and are the most basic operation of VEX processors. A collection of these syllables together form an instruction in the VEX ISA. The number of syllables in an instruction depends on the issue width of the particular implementation. An example of an instruction is given in figure 5-2, which is the format of an instruction of a four-issue implementation, where one box represents a syllable. Instructions are treated as atomic units in the VEX

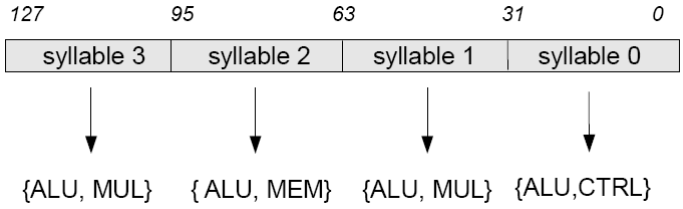


Figure 5-2: The instruction format of the ρ-VEX, containing four syllables. Each box contains one syllable which describes an operation. Every box is connected to functional units, in this case two functional units. *Source: R. Seedorf, "Fingerprint Verification on the VEX Processor," Master's thesis, Delft University of Technology, 2010*

ISA. An instruction of a VEX instance contains one or more syllables, i.e. operations, and these operations are executed as a single atomic action. All operands that are needed by the operations within an instruction are read first, before any operation commits its results to the registers or memory. The execution behaviour is that of an in-order machine, i.e. each instruction executes to completion before the start of the next one, and instructions are executed in program order. The exception behaviour follows this in-order execution model too. Latencies of functional units are visible in the instruction set architecture and the compiler has to schedule its code such that it obeys the latency constraints. Operations have latencies that depend on the functional unit and the compiler schedules the instruction in accordance with these latencies. If the hardware requires more cycles to execute the operation than was assumed by the compiler, the hardware stalls until the architectural requirements that are specified in the VEX ISA hold again. For implementations that do not stall, the behaviour of the execution of the instructions is undefined.

The VEX instruction set architecture has the usual set of instructions that can be expected of a typical RISC-processor. They can be divided into the following classes of instructions.

- Arithmetic and logic operations
- Memory operations
- Control operations
- Intercluster communication

The last category in this list is not a RISC-style instruction class, but is typical of the VEX architecture where multiple clusters can be present and data and other information needs to be exhanged between clusters. VEX supports the usual set of RISC-style integer operations, and some less traditional operations that make some often used operations more efficient. VEX focuses on integer operations, it does not support floating point operations.

Floating point operations have to be emulated if desired. VEX is a load/store-architecture, which means that only load and store operations can access memory. Memory accesses are restricted to native alignment, and misaligned accesses cause a nonrecoverable trap. The VEX is a conventional branch architecture in that it does not expose a branch delay slot and all instructions that come after the branch instruction are flushed from the pipeline when the branch is taken. The branch architecture uses branch registers to determine if branches have to be taken. These branch registers can be set using the usual logic and compare instructions. The VEX contains multiple branch registers, this is done so that the compiler can prepare several branches before they are actually executed. VEX specifies branch target addresses either through a relative displacement from the program counter or through the content of a special register. Call operations are supported by the VEX through a branch-and-link style operation, which saves the return pointer to the special link register. All other calling conventions are the responsibility of the software.

5-5 The VEX Implementation

5-5-1 Previous Versions of the ρ-VEX

At Delft University of Technology the VEX has been used in several research projects. Van As [21] describes an implementation of the VEX, called the ρ-VEX, which was used as a peripheral in the MOLEN project. It consists of four stages:

1. Fetch Stage
2. Decode Stage
3. Execute Stage
4. Write Back Stage

This implementation is a so called multi-cycle architecture where each stage is implemented by a finite state machine (FSM). Each stage takes several cycles to complete, and all stages are controlled by an inner control stage. This resulted in a reduced complexity and allowed the stages to share functional units during execution time. A disadvantage of this design choice is that the average clock cycles per instruction (CPI) is well above 1, hence making it a slow design.

Although the design discussed in the previous paragraph was satisfactory in terms of the project's requirements it was not suitable for another project which is discussed in [14]. Here a new design was proposed resulting in a five stage pipeline architecture. It's main advantage being that it reduces the CPI to 1, making it a faster implementation. This design will be discussed in the following section.

5-5-2 Organization of the ρ-VEX

The current implementation of the ρ-VEX can be described by the following characteristics [14],[20]:

- The ρ-VEX is a Harvard architecture. The organization of the memory system is divided into a memory for the instructions and a memory for the data. This way of organizing the memory system reduces memory access conflicts.
- It has four issue slots.
- It has a pipelined micro-architecture with five pipeline stages, see figure 5-3.
- The processor has four 32-bit ALUs, i.e. each issue slot has one ALU unit. It has two 16×32 bit multipliers, one Load/Store unit, and a branch unit.
- It has a 64×32 bit general purpose register file.
- It has a branch register file with 8×1-bit registers, specified by the ISA. It is used to store branch conditions, predicate values and it is used to store the carry of arithmetic functions.

In accordance with the VEX ISA specification the number and type of the functional units can be changed, except for the branch unit. There can be only one branch unit and it must be placed in the first issue slot of cluster 0. By being able to change the type and number of functional units the user is able to adapt the processor to the requirements of the application. The current VEX implementation uses a five stage pipeline. It consists of a fetch stage, a decode stage, two execution stages, and a write back stage. A detailed description of the

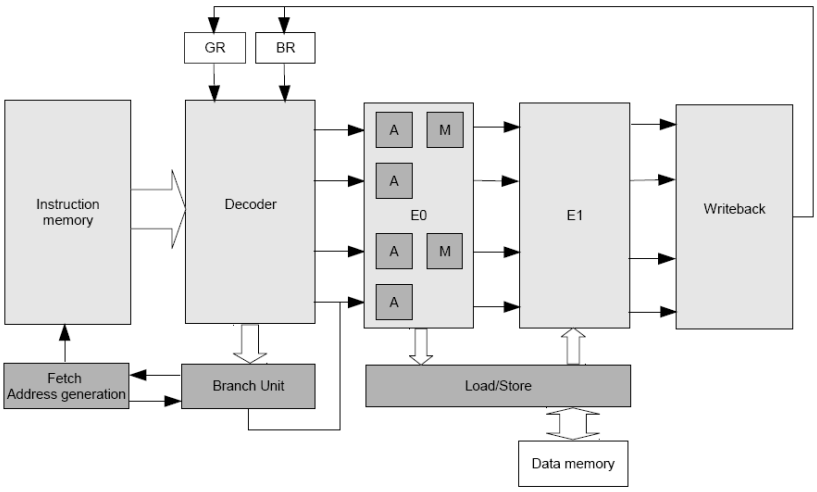


Figure 5-3: The 5-stage pipeline of the current ρ-VEX processor. *Source: R. Seedorf, "Fingerprint Verification on the VEX Processor," Master's thesis, Delft University of Technology, 2010*

micro-architecture of the current VEX implementation is beyond the scope of this thesis, but there are some observations that have to be mentioned.

Because the current implementation uses a pipeline architecture, typical issues related to pipeline design have to be considered, such as data dependencies and branch penalties. Forwarding logic is implemented to reduce data dependencies between the write back stage and the branch unit, and between the decode stage and the write back stage. Dependencies

have been reduced to a minimum, but some remained. The resulting latencies have to be taken into account by the compiler when scheduling the syllables into instructions. NOPs have to be inserted where necessary to resolve any dependencies that are the result of visible latencies. The branch syllable has a single-cycle branch penalty. All other latencies are hidden from the compiler and programmer.

5-6 The VEX Toolchain

The VEX toolchain that comes with VEX is the development system for VEX. It includes a C compiler and a simulation system. The C compiler is derived from the Lx compiler which is itself a descendant of the Multiflow compiler. It has the robustness of an industrial compiler and allows for experimenting with new architecture ideas, scalability and the introduction of custom instructions, by choosing parameter values in a configuration file that is read by the compiler. The compiler, which is only available in binary form, is limited in many ways. Currently, the compiler can only read C-source files, other languages such as for example Java and C++ are not supported. Furthermore, for its scheduling it only uses some basic compilation techniques.

The default behaviour of the compiler is to generate an executable that can be executed on the host system. The executable contains a simulation of the code produced for the VEX. See figure 5-4. The behaviour of the compiler can be changed by giving it the appropiate options on the command line. With proper options set it can produce all intermediate files, if necessary. The compiler is capable of optimizing the code by using the proper options. Default behaviour is to optimize as if option -O2 was chosen. The compiler is capable of loop unrolling but this should be used appropriately. Some other optimizations are inlining and compacting the object-files. It is possible to do profiling with the compiler. Finally, the compiler is able to support a limited amount of reconfigurability by presenting it with a configuration file.

As was explained earlier, the default behaviour of the compiler is to produce an executable for the host machine. There is no compiler option to produce an executable for an existing implementation of the VEX. For that purpose the toolchain was adapted, see [20], to produce VEX object code from the assembler code generated by the C compiler. This has been accomplished by using the GNU binutils package. An extra tool was added with which it is possible to extract VHDL code from the executable that is produced by the linker of the binutils package. It produces a VHDL-file containing the data of the program, and it produces a VHDL-file that contains the binary code of the program. These files form a part of the VHDL-description of the VEX, and so the code becomes part of the design, so to speak. Furthermore, it produces a `prog.h` file that can be included in a C project, and which contains the data and instructions of the program for the ρ-VEX. This data must be streamed to the ρ-VEX before it can be executed.

The VEX tools can produce graph information that is compatible with the VCG tool, which is a tool that converts graph information to graphical information. The compiler can emit control flow graphs, DAGs, and compacted schedules. Output produced for the gprof utility can be converted into a graphical representation with the help of the rgg utility that comes with the VEX toolset. Visualization helps to determine where to spend time in the source

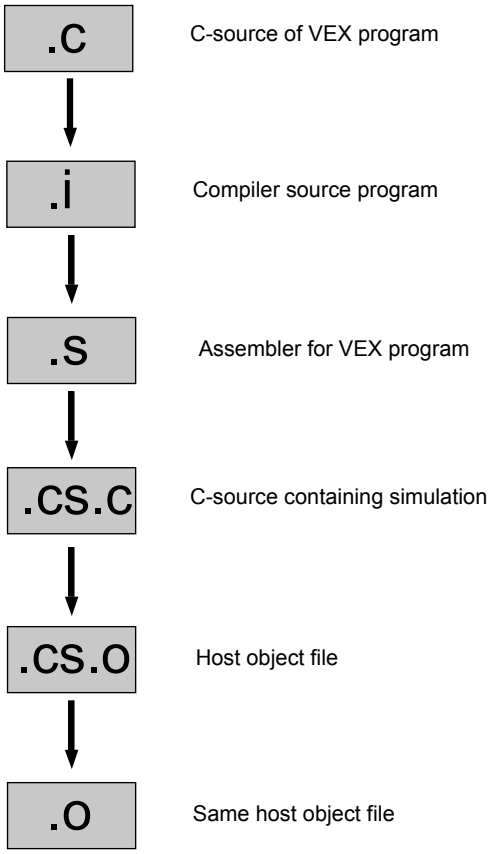


Figure 5-4: The intermediate steps that the compiler takes in producing the host executable

code looking for optimization possibilities. The VEX simulator includes 'simulated' support for gprof. It emits profiling data with and without cache simulation data.

5-7 The Complete System

There is a so-called hardware platform generated that enables some of the hardware's described in previous sections, i.e. MicroBlaze and ρ-VEX and other components such as timer and DDR memory to communicate with each other. The communication between the ρ-VEX processor and the Microblaze processor is established by using a Processor Local Bus (PLB). Through this bus, the MicroBlaze enable to read/write the ρ-VEX memory and exchange the input data and results between these two processors. This platform is generated by Xilinx Platform Studio. The schematic of the complete system can be seen in Figure 5-5. This complete system consists of MicroBlaze processor, `rvex_plb_wrapper` containing the ρ-VEX and its data and instruction memory, a timer and DDR memory mainly. Processor Local Bus connects the components to each other as it is represented in Figure 5-5. The wrapper links the instruction memory `i_mem` and data memory `d_mem` of the ρ-VEX to the PLB.

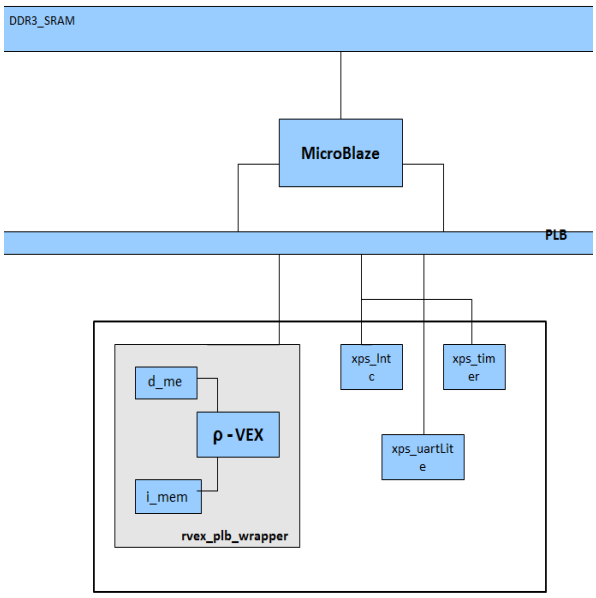


Figure 5-5: Communication between the components of the platform

More details of the communication between the MicroBlaze processor and ρ-VEX processor will be given in chapter 6 when discussing the partitioning of the application on ρ-VEX/Microblaze. Below follows a detailed information on the used PLB.

Processor Local Bus

The communication between the ρ-VEX Processor and the Microblaze Soft Core Processor is established by a Processor Local Bus (PLB). As it is already said in section 5.2, the MicroBlaze has the CoreConnect PLB as its I/O bus. The CoreConnect is a microprocessor bus architecture which is being used for system-on-a-chip (SoC) designs. The Processor Local Bus, namely PLB v4.6 protocol prescribed by Xilinx, is one of the elements it is including.

This PLB enables the designer to connect some number of PLB masters and slaves by making use of a bus infrastructure so that the designer can get an overall PLB system. So, it enables data transfer between devices defined as masters and slaves within that bus infrastructure [22]. The master ones make use of separate address, read-data and write-data buses and the slave ones make use of shared, but decoupled address, read-data and write-data buses in order to be fixed into the PLB.

Some features of this type of PLB are listed as :

- Arbitration support for a configurable number of PLB master devices
- PLB address and data steering support for all masters
- 128-bit, 64-bit, and 32-bit support for masters and slaves
- PLB address pipelining supported in shared bus mode or point-to-point configuration

- Three-cycle arbitration
- Configurable optimization for point-to-point topology [23].

5-8 Conclusion

This chapter has covered the most useful information on the used hardware's in this project. In the first section the reader has seen that there are three advantages of this type FPGA above its previous generations; up tp 50 percent higher performance, lower cost and power and the presence of breakthrough I/O performance. Because it also supports partial configuration, the versatility of this board is considered to be improved due to this feature.

In the second section MicroBlaze soft core processor has been represented to the reader which is a RISC and uses Harvard architecture core. The latter deals with separate storage and signal pathways for instructions and data which helps to improve the performance. One of the most important feature of this processor is that it makes use of Big-Endian format which always has to be kept in mind while working on the relevant code of the fingerprint munitiae extraction application in order to read the data properly in and out.

In section 5.3 the VEX system has been discussed in detail which was derived from HP/ST Lx architecture. The following three basic components that the VEX system consists of are mentioned ; The VEX Instruction Set Architecture (ISA), The VEX C Compiler and The VEX Simulation System. The VEX ISA offers a flexible and scalable architecture for VLIW processors so that some important aspects such as the issue width, the instruction set and number of functional units can be customized. The VEX operations are very similar to a RISC-style operations. These operations are called syllables and the collection of these forms an instruction in the VEX ISA. Another most important feature of the VEX is that it only supports integer (fixed-point) operations and no floating point operations as it is also denoted in section 4.3. Second component of the VEX system, The VEX C Compiler, allows the user to configure the target VEX architecture by specifying a configuration file which enables the user to change the number of clusters, execution units, issue width and operation latencies. The VEX Simulator allows the use of most of the familiar C-libraries and uses the VEX ISA specification to simulate execution of the instructions. Implementation details of a specific implementation are not considered by the VEX simulator.

In section 5.4 the implementation of the VEX, the ρ-VEX processor, has been represented which was used as a peripheral in the MOLEN project at Delft University of Technology. It is a multi-cycle architecture where each of the four stages; fetch, decode, execute and write back stages, is implemented by a finite state machine (FSM). It has been stated that the design for MOLEN project is a slow one as the average clock cycles per instruction (CPI) is above 1. The current rho-VEX with the six characteristics given in 5.4.2 can be seen as a better design because the older one was only suitable for the MOLEN project. The current VEX implementation uses a five stage pipeline; fetch stage, a decode stage, two execution stages, and a write back stage. As the current implementation uses a pipeline architecture, typical issues related to pipeline design have to be considered, such as data dependencies and branch penalties. Although dependencies have been reduced, they are not completely eliminated.

In section 5.5 of this chapter the VEX toolchain has been discussed. The toolchain includes a C compiler and a simulation system. The compiler is able to read C-source files only, other

languages such as Java and C++ are not supported. The compiler generates an executable which contains a simulation of the code produced for the VEX and that can be executed on the host system. There is no compiler option to produce an executable for an existing implementation of the VEX. Furthermore, the compiler is capable of performing optimization of the code such as loop unrolling, inlining and compacting the object files. The VEX tools is also able to produce graph information that is compatible with the VCG tool. This tool converts graph information in numbers to graphical information with the help of the rgg utility.

In the last section the hardware platform consisting of the components used in this project has been discussed. This forms a kind of communication area for these components to communicate with each other. It has been stated that the communication between the ρ-VEX Processor and the Microblaze Soft Core Processor is established by a Processor Local Bus. The MicroBlaze enable to read/write the ρ-VEX memory and exchange the input data and results between these two processors through this bus.

Chapter 6

Hardware/Software Mapping

In this chapter the partitioning process design and simulation results will be presented. Simulation is performed to verify the output results of the accelerator and to determine the performance. The simulation results are given in this chapter. Section 6.1 will present the profiling process and performance of the Minutiae Extraction application on the VEX simulator. Section 6.2 presents the description of experiments the application on MicroBlaze and gives the profiling result. Section 6.3 presents the process design and the performance of Minutiae Extraction on Microblaze. In section 6.4, we have conducted experiments on the simulator to find the right configuration for the VEX implementation which has to execute the kernel. In Section 6.5 discusses several optimization that were investigated on the VEX simulator. Finally, in the section 6.6 we discuss the execution times of the partitioned application together with the acquired speeding.

6-1 Profiling the Minutiae Extraction Application on the VEX Simulator

Profiling the application on the VEX simulator is the first step of the experiments in this project. In chapter 4, the fingerprint minutiae extraction application has been presented and this gave a good basic understanding of the theory that explains how the application works but profiling will help the designers to know more about the application practically. This involves some important information about the application such as the amount of total cycles, execution cycles, stall cycles and executed times that the program spent executing each function and its children. To analyze how much time fingerprint minutiae extraction application spent executing each function, we execute the application on the VEX simulator.

6-1-1 Description of Experiment

In order to profile the application, the `gprof` profiling tool which is also available within the VEX Simulator is used. The application has been executed on the VEX Simulator several

times for different machine configurations. Then the results are obtained by running the standard `gprof` utility. The different machine configurations deal mainly with different number of issue widths such as 1, 2, 4 and 8 issue widths in order to determine what kind of effect these different issue widths can have on the speed of the application. It is also a question whether multiple clusters would be required to execute the application with higher speedup, but an earlier research has shown that for the minutiae extraction application a single-cluster processor is more promising. It has been stated in [14] that the speedup improvement from a multi-cluster datapath is marginal and therefore it is decided to use a single cluster implementation in this step of the experiments. Thus, only the number of issue width of VEX datapath are varied, while keeping other datapath parameters such as the number and sort of functional units and the number of clusters constant.

Total issue width	1 Issue	2 Issue	4 Issue	8 Issue
Total Time	14210.9 msec	11320.4 msec	8744.34 msec	10783.8 msec
Total Cycles	7105455801 cycles	5660207983 cycles	4372171195 cycles	5391915615 cycles

Table 6-1: Simulation cycles for single cluster

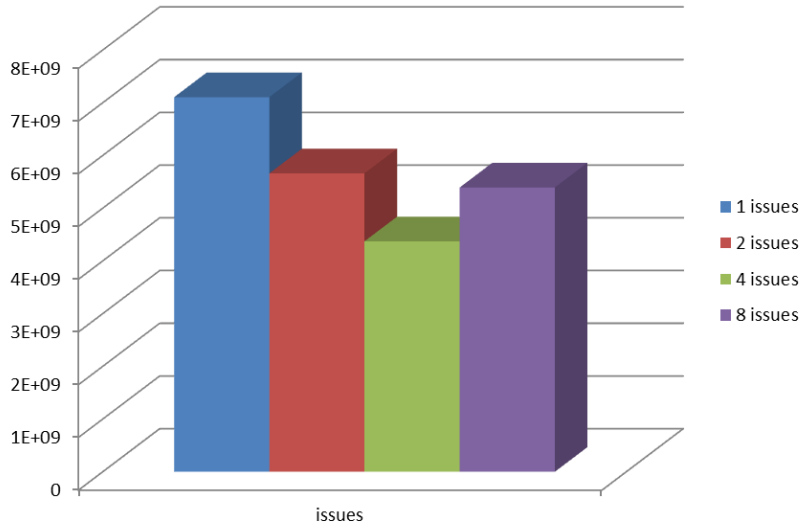
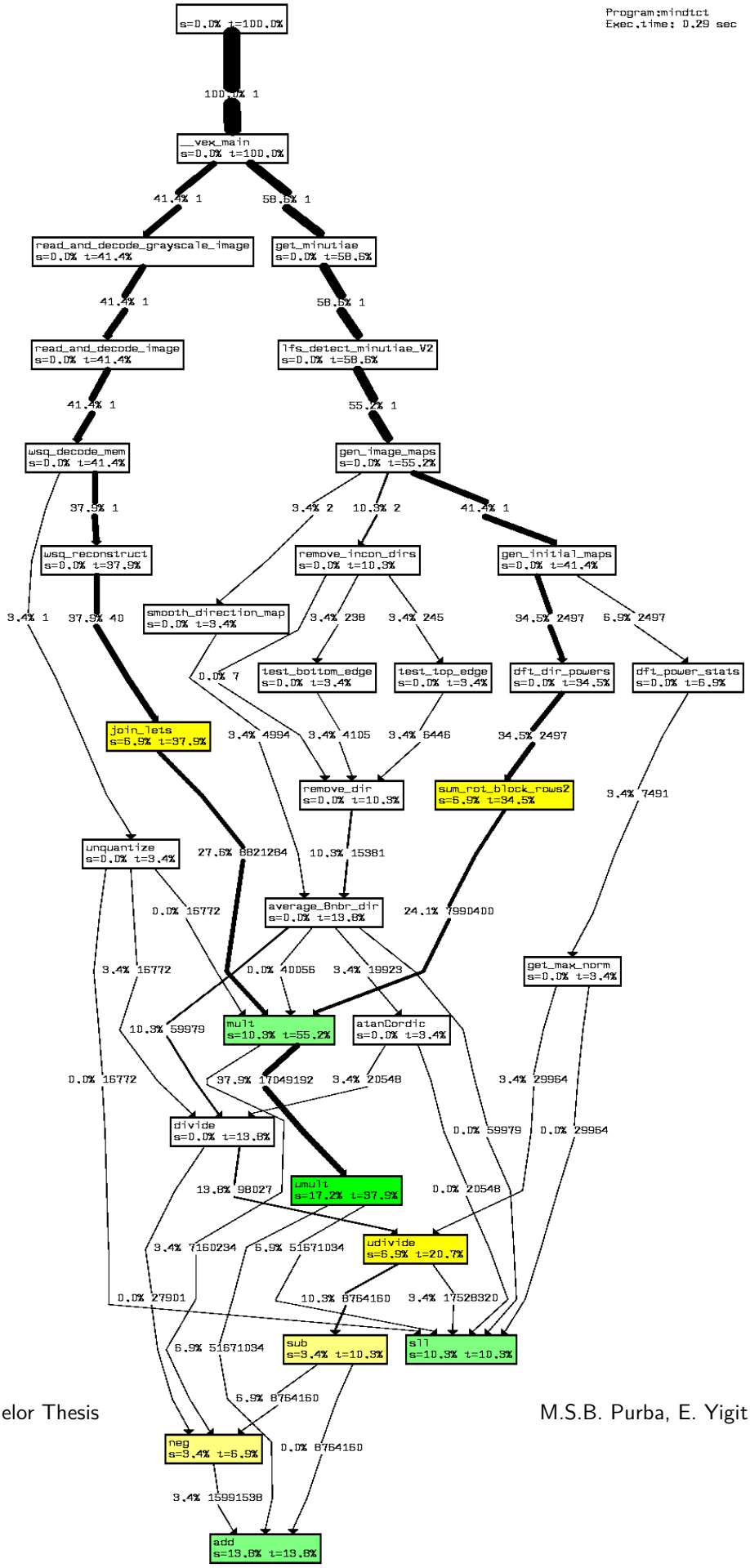


Figure 6-1: Execution time on simulator for different issue width

The application has been executed for each of these different number of issue width's on VEX Simulator. The number of clock-cycles it takes to run the application for each of these different number of issue width are presented in table 6-1 and a graphical illustration is depicted in figure 6-1. The graph shows the total execution time on the vertical axis and the issue-width on the horizontal axis. These profiling results and their significance will be discussed in the next subsection. Additionally, the profiling results can also be make graphically visual by using the `rgg` utility included in the VEX toolchain and the user can get a graph called call graph such as in figure 6-2. The functions taking less than 7% of the time are filtered out in this example graph. There follows no further explanation on this graph because it is just intended as an example to show how a call graph looks like.


Program:mindtct
Exec.time: 0.29 sec
s=0.0% t=100.0%
100.0% 1
__vex_main
s=0.0% t=100.0%
41.4% 1
58.5% 1
read_and_decode_grayscale_image
s=0.0% t=41.4%
get_minutiae
s=0.0% t=58.5%
41.4% 1
58.5% 1
read_and_decode_image
s=0.0% t=41.4%
lfs_detect_minutiae_V2
s=0.0% t=58.5%
41.4% 1
55.2% 1
wsq_decode_mem
s=0.0% t=41.4%
gen_image_maps
s=0.0% t=55.2%
37.9% 1
3.4% 2
10.3% 2
41.4% 1
wsq_reconstruct
s=0.0% t=37.9%
remove_incon_dirs
s=0.0% t=10.3%
gen_initial_maps
s=0.0% t=41.4%
3.4% 1
37.9% 40
smooth_direction_map
s=0.0% t=3.4%
3.4% 238
3.4% 245
34.5% 2497
6.9% 2497
0.0% 7
test_bottom_edge
s=0.0% t=3.4%
test_top_edge
s=0.0% t=3.4%
dft_dir_powers
s=0.0% t=34.5%
dft_power_stats
s=0.0% t=6.9%
join_lets
s=6.9% t=37.9%
3.4% 4994
3.4% 4105
3.4% 5446
34.5% 2497
remove_dir
s=0.0% t=10.3%
sum_rot_block_rows2
s=6.9% t=34.5%
3.4% 7491
unquantize
s=0.0% t=3.4%
27.6% 8821284
10.3% 15381
0.0% 16772
average_8nbr_dir
s=0.0% t=13.8%
24.1% 7990400
3.4% 16772
0.0% 40056
3.4% 19923
get_max_norm
s=0.0% t=3.4%
10.3% 59979
mult
s=10.3% t=55.2%
atanCordic
s=0.0% t=3.4%
0.0% 16772
37.9% 17049192
3.4% 20548
3.4% 29964
divide
s=0.0% t=13.8%
0.0% 59979
0.0% 29964
13.8% 98027
umult
s=17.2% t=37.9%
0.0% 20548
3.4% 7160234
6.9% 51671034
udivide
s=6.9% t=20.7%
0.0% 27901
10.3% 8764160
3.4% 1752832
6.9% 51671034
sub
s=3.4% t=10.3%
sll
s=10.3% t=10.3%
6.9% 8764160
neg
s=3.4% t=6.9%
0.0% 8764160
3.4% 15991536
add
s=13.8% t=13.8%

6-1-2 Results and Interpretation

As it is apparent from Table 6-1 and Figure 6-1 increasing the issue width decreases the number of clock-cycles and the execution time, resulting in marginally higher speedup. This holds up to 4 issue width, the number of clock cycles begins to increase again from 8 issue width on resulting in less speedup. As it has been proved in [24], a possible explanation for the increased execution time for 8 issue width is an increased code size and data cache conflict stalls. From the results, it looks temporarily more promising to concentrate the hardware design efforts on 4 issue width. It is said temporarily because this result from the VEX Simulator doesn't say that also the machine configuration with 4 issue width will have to be used on rho-VEX processor for the computational intensive kernel that will be determined after profiling the application on MicroBlaze processor, because the characteristics of the kernel may differ from that of the whole application so that it would be more sufficient to use, for example, 2 issue-width. Again, from these results it can be assumed that the proper number of issue-width in order to get the highest speed up is 4. If there will be enough time, the kernel will also be tested for the other numbers of issue-width to find a more accurate configuration of rho-VEX processor, otherwise 4 issue-width will be used as the right configuration of a VEX implementation running the kernel.

6-2 Profiling on MicroBlaze

The following step of the experiments in this project is profiling the application on MicroBlaze. As described in chapter 5, a platform has been generated containing the MicroBlaze processor. In this section, the description of the bottleneck of the application on MicroBlaze is presented. In section 6.2.1, the result of running the application on MicroBlaze is described. In subsection 6.2.2, the application running on MicroBlaze is described. In subsection 6.2.3, the result of profiling in MicroBlaze is interpreted.

6-2-1 Description of Experiment

Running and profiling the fingerprint minutiae extraction application on MicroBlaze can be seen as the first most important step in this project. The application has to be run on MicroBlaze in order to determine its execution speed before accelerating it using the ρ-VEXprocessor and it has to be profiled on MicroBlaze in order to know which part of the application has to be mapped to ρ-VEX processor. The part that has to be accelerated using the ρ-VEXprocessor is called the computational intensive kernel. The reader may ask why not the whole application is mapped to ρ-VEXso that the whole application is accelerated at once. The reason for this is that the fingerprint application is too big and if it is tried to accelerate the whole application at once, almost all the capacity of the FPGA will have to be used and this makes it difficult to run the whole application on the ρ-VEXprocessor. It also would cost the designers too much time to fix this problem. For these reasons it is decided to be a better solution accelerate the kernel of the application only that takes pretty much time.

6-2-2 Running the Application on MicroBlaze

Before profiling the application it has to be made adaptable for running on MicroBlaze processor and as well as for ρ-VEX processor. This is done by modifying and stripping the code down further than it was done by The Circuits And System Research Group of TU Delft. The microblaze and ρ-VEXdo not have an operating system, so there is no support for basic input/output. The original code contains many `fprintf` and `printf` calls for generating log files and debug information, however, these functions make use of the underlying file-IO system. So, because these functions cannot be used, they had to be removed and substituted for similar output routines that are still available to use within the two processors. The following library functions that have been adapted are explained in the following.

- `fprintf`: This function is used to print out a sequence of characters of fingerprint to a file.
- `fopen`: This function is used to read the input file; i.e. `wsq`-file, in the main `mindtct` file.
- `printf`: This function is used to print formatted output to the standard output stream which is usually the screen.

The problem how to print to the screen is solved by printing formatted output to a character buffer using the `sprintf` function and subsequently this buffer is printed to the screen by using the print function provided by Xilinx. The code had to be changed, otherwise all `fprintf` and `printf` functions would not work on the MicroBlaze because it lacks file I/O support. In short, there have been worked around the problem of reading the output data is solved by using the print statement and where necessary the `sprintf` function in combination with the print function. Because the MicroBlaze has also no file input support, there also had to be found another way to read the input data. The solution to this problem is solved by writing a little program that reads the required input file and converts the data within this file to a C-source code so that it became suited to the processors. The language chosen for this purpose is Python and the script is shown in appendix A. After modifying the code and stripping it down further, it is able to run on the MicroBlaze processor and to profile it. Next section deals with profiling the application and the profiling results.

6-2-3 Profiling the Application on MicroBlaze and Profiling Results

After the application has been made suitable to be run on the MicroBlaze, the application had to be profiled to find out where the program spends its time and which functions take more time than it is expected while it is being executed. This is needed as not the whole application can be accelerated using ρ-VEX because it is too big. Also the number of times the function actually has been called and the cumulative time spent in the functions can be determined by profiling the application. Profiling on MicroBlaze is based on the GNU `gprof` tool. This tool prints a flat profile and it also prints a call graph on standard output after having a profile data file named `gmon.out`. From the profiling results listed in Table 6-2, it is clear that the code in `long_int_math.c` and `dft.c` take the most of the execution time.

Codes	samples	Time[%]
long_int_math.c	12813	85.92
dft.c	700	4.69
util.c	581	3.90

Table 6-2: Profiling results on Microblaze

Functions of dft.c	Samples	Time[%]
dft_dir_powers	0	0.0
dft_power_stats	0	0.0
sort_dft_waves	2	0.01
get_max_norm	5	0.03
sum_rot_blocks_rows2	693	4.65

Table 6-3: Profiling results of code dft.c on Microblaze

Furthermore, `dft.c` is the c-code that contains routines responsible for conducting Discrete Fourier Transforms (DFT) analysis on a block of image data.

Table 6-3 represents the functions within `dft.c` code and their profiling results. From these results one can see that `sum_rot_blocks_rows2` function takes almost the whole time within `dft.c` code. This means that only this function from `dft.c` will be mapped to the accelerator beside the functions of `long_int_math.c`. So, these functions form together the kernel of the application.

6-3 Partitioning the Minutiae Extraction Application

In the previous section we found, by profiling the application on the MicroBlaze, that the function `sum_rot_block_rows2()` together with the functions of the `long_int_math.c` module consume the most execution time. So we decided to accelerate these functions on the ρ-VEX. In the first section the program for the ρ-VEX is discussed. In the second section the modifications to the main application are described that are necessary for the new partitioned application to function correctly.

6-3-1 The Kernel Program on the ρ-VEX

The modules that are used for the kernel program are described in table 6-4. A new module `computeDFT.c` was written that contains the `main()` function together with the `sum_rot_block_rows2()` function, which is called from the `main()` function. See appendix D for the source code of the `computeDFT.c` module. In the `main()` function we first initialize the memory of the powers matrix. The powers matrix is composed of four arrays of integers, representing the rows of the matrix, and powers is an array containing pointers to these arrays. Note that in the original application space in memory is assigned by calling the `malloc()` function. But in the ρ-VEX there is no such function, so we had to find an alternative for creating the necessary memory space. We created an array of 64 integers and considered that

Module	Description
`_start.s`	Assembler file that sets the stack pointer and calls the entry point of the program.
`computeDFT.c`	Module containing the `main()` function and `sum_rot_block_rows2()`.
`dft_lookup.h`	Contains the function values for the sine and cosine functions.
`fixed_point_math.h`	Macro definitions that are defined in terms of the macros of `fixed_point_mathcode.h`.
`fixed_point_mathcode.h`	Contains macros that emulate floating point operations.
`grid_lookup.h`	Used for DFT computations.
`lfs.h`	Contains all definitions of structures and constants that are used in the application.
`long_int_math.c`	Functions enabling 64-bit integer arithmetic.
`long_int_math.h`	Corresponding header file.

Table 6-4: The C-source modules that are used in the kernel program.

array to be composed of four arrays of sixteen integers each. We then stored the pointers to the beginnings of these arrays into powers, effectively creating the same result that is obtained in the original program.

After the memory is initialized, the function `sum_rot_block_rows2()` is called from `main()` and when it returns, the program on the ρ-VEX is finished and an interrupt is generated indicating that the ρ-VEX has stopped execution. This interrupt is then used by the main application to know that the kernel has computed the results.

The kernel program uses some additional module files that are necessary for its proper operation. The module `long_int_math.c` contains the functions that are necessary for 64-bit integer computations. Some header files contain arrays that are used in the DFT computations. In `dft_lookup`, for example, the values for cosine and sine functions are stored.

The kernel program can be compiled with the makefile in appendix C. For compilation of the kernel, the configuration file in appendix F has been used. It uses the tools in the binutils package to produce an object file which is used as input for the `elf2vhd` tool to extract the vhdl files that contain the instructions and data of the kernel. It also produces the `prog.h` file that is used in the main application to load the program to the ρ-VEX. To acquire the locations of the data where is going to be written en read from the `objdump` tool is used, which produces a list of global variables of the program together with their position and size.

6-3-2 Adaptations to the Main Application

In the original program the function `sum_rot_block_rows2()` is called by the function `dft_dir_powers()` to compute the directional information in a block of 34×34 pixels of

the fingerprint image. But in the partitioned application this computation is done by the ρ-VEX, so we had to change this call to `sum_rot_block_rows2()`. The partitioned application calls instead the function `sum_rot_block_rows2_onVEX()`. This function stops and resets the ρ-VEX and then writes the bytes of the 34×34 block contained in the array `data` to the memory of the ρ-VEX. The location for this array in the memory of the ρ-VEX was obtained with the `objdump` program. Then it starts the ρ-VEX and consequently waits for the ρ-VEX to give the signal that it has computed the results, see figure 6-3. When the results are computed, they are read from the memory of the ρ-VEX. The results are read from the `powersgeheugen` location. This location is also obtained with the `objdump` tool. The integers that are read in are then put into the powers matrix. The source code for this function can be found in appendix B.

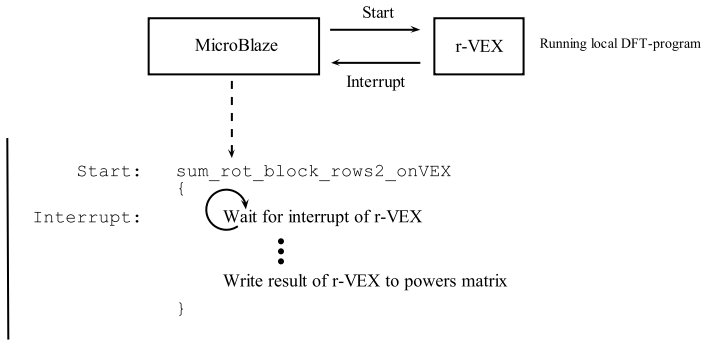


Figure 6-3: The main application loads the data into the memory of the ρ-VEX and then starts the ρ-VEX. It waits for the ρ-VEX to compute its results and then reads the results back into the powers matrix.

The ρ-VEX has to be initialized before it can be used. The kernel program has to be loaded into the ρ-VEX and the interrupt routine has to be initialized. This is done in the function `initialize_VEX()` and it is called in the `mindtct.c` module just before minutiae extraction is commenced. The program that is loaded into the ρ-VEX is obtained from the `prog.h` file. This file is produced by the `elf2vhd` tool in the binutils package.

6-4 Finding the Right Configuration for the ρ-VEX

In section 6.1 we described a profiling of the fingerprint minutiae extraction application on the VEX simulator in order to determine what kind of effect several different issue width such as 1,2,4 and 8 can have on the speed of the application. This profiling was done for the whole application and the results have shown that 4 issue width would be the best choice to use for the whole application. As it is already denoted in that section, that was a provisional result from which it could be assumed that it also holds for the kernel, but as there was enough time to test the kernel only, also a similar research has been done on the dft.c code found in the previous section. It was needed to determine the accurate configuration for the VEX implementation in order to execute the kernel with the highest possible speed. Thus, the computational intensive kernel, i.e. dft.c, has also been executed on VEX simulator for 1,2,4 and 8 issue-width. Also in this experiment other datapath parameters such as the number and sort of functional units and the number of clusters are kept constant while varying only

the issue-width of VEX datapath. From the profiling results represented in table 6-5, one can see that also for the kernel only, the ideal number of issue-width is 4. Total cycles and total execution time decrease by increasing the number of issue-width up to 4 and with 8 issue-width they increase again which is similar to the results found in section 6.1 for the whole application. These results are also represented graphically in Figure 6-4. The graph shows the total execution time on the vertical axis and the number of issue-width on the horizontal axis.

Total issue-width	1 Issue-width	2 Issue-width	4 Issue-width	8 Issue-width
Total Time	1.98389 msec	1.57864 msec	1.22525 msec	1.51372 msec
Total Cycles	991943 cycles	789320 cycles	612627 cycles	756858 cycles

Table 6-5: Execution times of the kernel for different issue-width obtained with the VEX simulator

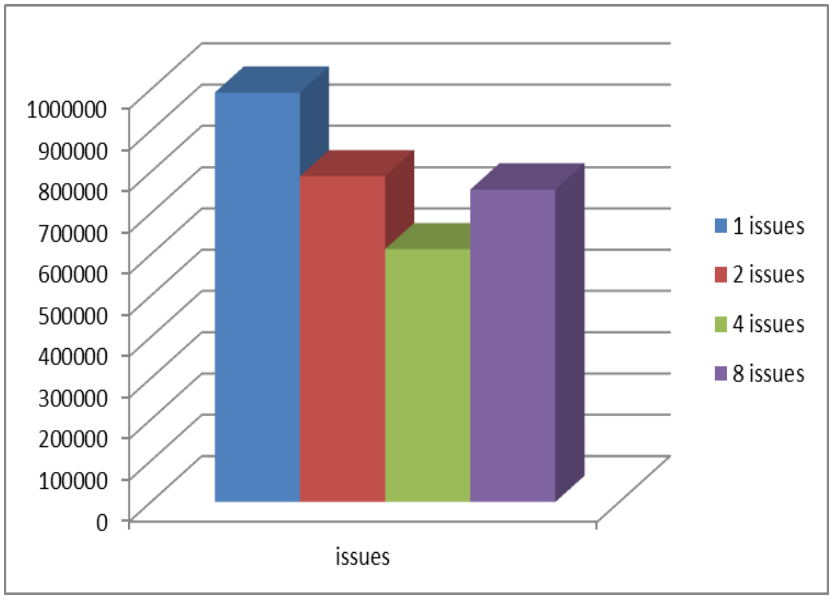


Figure 6-4: Kernel issues in VEX simulator

Apparently, the kernel has the same characteristics as the whole application. Thus, increasing the issue-width further than 4 does not help to reduce the execution time which, as desired, leads to higher speed up of the system. Again, the reasoning in subsection 6.1.2 holds also for this kernel. In short, the right configuration of the VEX in terms of the issue-width is 4 as it was assumed from the results in section 6.1.

6-5 Optimizing the Software Program on ρ-VEX

After profiling the application on the Microblaze, the kernel is known. The kernel will be optimize for the ρ-VEX. This can be done by applying different optimization techniques of the compiler. The benefits of these optimization techniques are that the program will run faster. As it is mentioned in section 6.4, 4 issue-width is the accurate configuration for the ρ-VEXprocessor in order to execute the kernel. Therefore, this configuration is used further in

optimization process. Today, most compilers support compiling techniques such as inlining, loop unrolling etc. There are many optimizations a compiler can be used to improve program performance, and particularly well-known optimization are loop unrolling and inlining. The several set of option can classify how the compiler optimize the application. Loop unrolling is a very useful technique for VLIW processors. Inlining is another technique that is also useful in optimization. In this section, the results of different optimization techniques are given. In subsection 6.5.1, the influence of different level of loop unrolling is investigated. In subsection 6.5.2, the result of different level of inlining are given. In subsection 6.5.3, the conclusions of optimization are given.

6-5-1 Loop Unrolling

To speed up the program, we can apply the loop unrolling technique. Loop unrolling is a method to improve the performance by reducing the repeating sequences in loop instructions. The main benefits of loop unrolling is reducing the loop overhead. Before optimizing the C code, we need to know what is the optimization limit of the code. In other words, what is the best cycle count we can achieve. Function `sum_rot_blocks_rows2 (..., ...)` contains such loops. There are also some limitation of loop unrolling. The loop unrolling can increase the memory overhead in every unrolls.

Compiler Options and Results of Experiments

In order to be able to compile the kernel, the kernel code need to be modified. Therefore, it is necessary to modified the code by hand to achieve optimization. Due to the code modification, the compiler can run the program efficiently. The modified kernel code can be found in appendix D. Several set option controls of loop unrolling has been applied to the kernel only in order to see wether the loop unrolling has some influence on the performance of the application. The flag `-Hn` and `-O2` for different optimization options can been used to perform loop unrolling. `-O2` is the default set in compiler. This will provide standard optimization. In [9], more details about different optimization options can be found. The compiler options of `-Hn` are given as followed:

- -H0 No unrolling
- -H1 Basic unrolling
- -H2 Aggressive loop unrolling
- -H3 Heavy loop unrolling
- -H4 Very heavy unrolling

In table 6-6, the clockcycle of different optimization options can be seen. From table 6-6, it is clear that the best result can be achieved by setting the flag to $-H3$. The relative speedup that has been achieved using loop unrolling is $816527/741059 = 1.10$. However, loop unrolling has some limitation. From the table 6-6, it is shown that the amount of instruction increased.

Compiler Option	-H0	-H1	-H2	-H3	-H4
Total Cycles	816527 cycles	749278 cycles	746331 cycles	741059 cycles	741195 cycles
Total Time	1.63305 msec	1.49856 msec	1.49266 msec	1.48212 msec	1.48239 msec
Total Instructions	1.52Kb	1.56Kb	1.64Kb	1.87Kb	1.87Kb

Table 6-6: Optimizing results of different loop unrolling compiler options

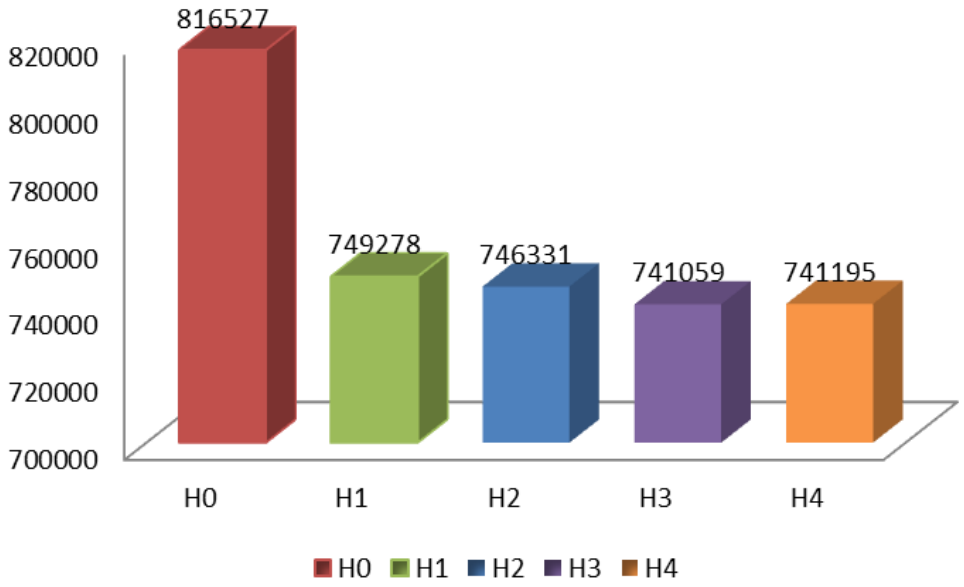


Figure 6-5: Optimizing results of different loop unrolling compiler options

6-5-2 Procedure Inlining

Other important technique to improve program performance is called inlining. This can be done by removing calling sequences in the program. When such a procedure does not contain a lot of code the overhead of calling the procedure can become quite substantial. The compiler can then substitute the code of the procedure body at the place where the procedure is called in the code, thereby saving the overhead of calling the procedure. This optimization method is called procedure inlining. The drawback of this technique is that it may increase the instruction size.

Compiler Option and Result of Experiments

The compiler offers two options to perform inlining. The first option is `-autoinline`. This is the most easy way to perform inlining. This can be done by setting the flag `-autoinline` in compiler. It enables automatic function inlining. The flag `-c99inline` is another option to perform inlining. Inlining compiler options perform some results. The experiment has been done by setting the -autoinline flag. In table 6-7, the total clockcycle of after and before inlining are given. From the table 6-6, it is obvious that inlining can speed up the kernel of the application. This experiment give us a total relative speedup of 754815/540313 = 1.39

Compiler Option	Before Inlining	After inlining
Total Cycles	754815	540313
Total Time	1.50963 msec	1.08063 msec
Total Instructions	1.54kB	1.56kB

Table 6-7: Comparison of results before and after inlining

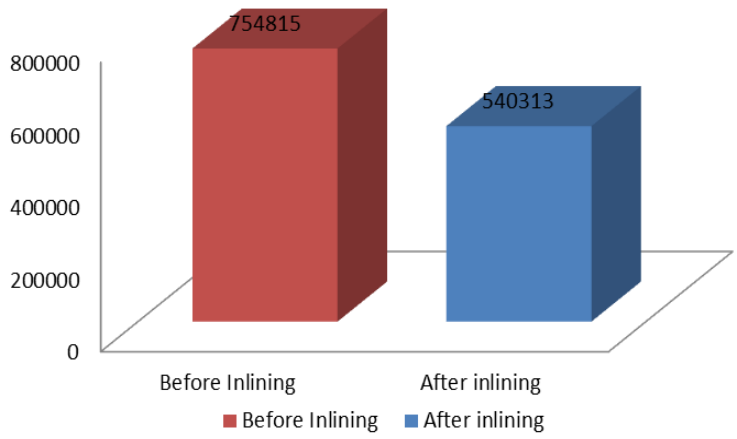


Figure 6-6: Comparison results before and after inlining compiler options

6-5-3 Conclusion of Optimizing

It is obvious that optimization methods can accelerate the kernel on ρ VEX. This has been proven in table 6-6 and table 6-7. Different compiler options have been evaluated in order to obtain the best acceleration performance. It appears in table 6-8 that loop unrolling combined with inlining gives the best acceleration. However, it uses a lot of memory of the ρ-VEXcompared with other optimization methods. Since the ρ-VEXprocessor has a maximum instruction memory of 8 kB, it is important to check the amount of instructions that the method has been gained. It is clear that this method increases the number of instructions.

In addition, loop unrolling and inlining optimization methods give also a substantial acceleration. The use of loop unrolling method indicated a factor of 1.1 faster than no loop unrolling. Inlining caused also a substantial acceleration. This provides a relative speedup with factor 1.39 faster than before inlining. The comparison of the acceleration between different optimization methods can be seen in 6-8. According to the optimization results, we can conclude that the acceleration of the kernel on the ρ-VEXcan be reached by applying different optimization methods.

Optimization Methods	Loop Unrolling	Inlining	Loop unrolling and Inlining
Total Cycles	741059	540313	521765
Total Instructions	1.8kB	1.5kB	3.1kB

Table 6-8: The comparison of different optimization methods

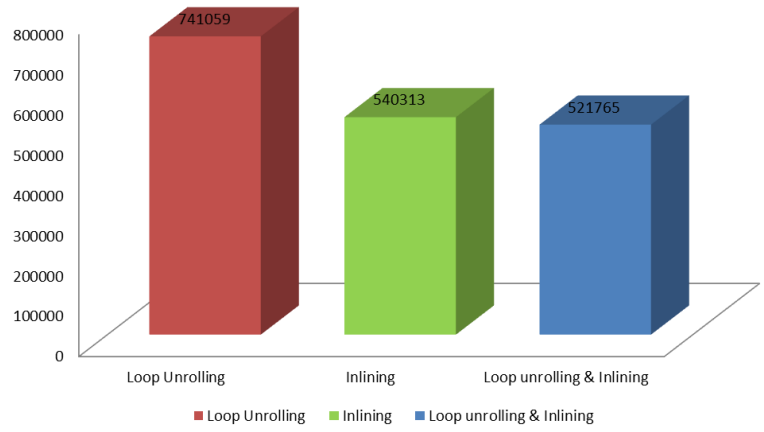


Figure 6-7: The comparison of the acceleration produced by different optimization option

6-6 Running the Partitioned Application

Once the application was partitioned and all modifications were made it was time to test the new application. But the application encountered problems almost immediately as is described in the following section. In the second section we give an estimation of the speedup

that can be obtained for the computation of one DFT block based on the results of the simulator. In the third section we present the measurements of the execution times of the application and the obtained speedup.

6-6-1 Encountered Problems

When all modifications to the application where made it was tried to run the application. We encountered several problems. First, we did not succeed in loading the program. It turned out that although we made modifications to the implementation of the ρ-VEX, and subsequentially generated a new bitstream, we were not aware of the fact that the file containing the bitstream has to be put manually into the SDK. So, for a while we were running an older implementation of the ρ-VEX. Once we put the bitstream into the right place, the loading of the program for the ρ-VEX went well.

The next problem we encountered, was that the kernel program on the ρ-VEX did not finish. It appeared, after some testing, that the call to `sum_rot_block_rows2()` did not return properly to main(). The last part of the code of this function is shown in assembler code below. It shows the code right before the return instruction and shows the code to restore some of the used registers. As can be seen register `$r0.63` is restored as well, but this is not supposed to happen because this register and the link register `$l0.0` are the same in the VEX ISA. Because the link register is saved seperately, see line `t90`, it was decided to remove the line that restores register `$r0.63`. It then did return properly.

```
.trace 13
L1?3:
c0    ldw $l0.0 = 0x70[$r0.1]  ## restore ## t90
;; ## 0
;; ## 1
;; ## 2
c0    ldw $r0.57 = 0x94[$r0.1]  ## restore ## t94
;; ## 3
c0    ldw $r0.58 = 0x98[$r0.1]  ## restore ## t95
;; ## 4
c0    ldw $r0.59 = 0x9c[$r0.1]  ## restore ## t96
;; ## 5
c0    ldw $r0.60 = 0xa0[$r0.1]  ## restore ## t97
;; ## 6
c0    ldw $r0.61 = 0xa4[$r0.1]  ## restore ## t98
;; ## 7
c0    ldw $r0.62 = 0xa8[$r0.1]  ## restore ## t99
;; ## 8
c0    ldw $r0.63 = 0xac[$r0.1]  ## restore ## t100
;; ## 9
;; ## 10
.return ret()
c0    return $r0.1 = $r0.1, (0xc0), $l0.0
## bblock 2, line 132,  t91,  ?2.1?2auto_size(I32),  t90
;; ## 11
```

Once this was fixed, the ρ-VEX gave an interrupt every time one block was calculated, indicating that the program finished. But the main application program produced unsigned cast errors for the 64-bit integers calculations when the results of the ρ-VEX were used, so we suspected that the problem was that the ρ-VEX did not produce the right values in the powers matrix.

In order to identify the problem we decided to focus on the computation of one block of 34×34 bytes, in particular we decided to take the 20^{th} block. In table 6-9 the values that should be returned, are shown. To rule out that we are reading from an incorrect place it was decided to change the kernel program such that it already contains the data and results in powers, and consequently we only loaded the program and then did not run it. We read the values in powers and they matched the ones in the table, so we are reading from the right place in the data memory of the ρ-VEX. When we do not let the ρ-VEX do the computation then the predefined powers matrix stays the same, and when this is used by the application on the MicroBlaze, the casting errors disappear. So, we narrowed the problem down to the ρ-VEX.

row 1	6557372	5018944	6064806	19806538	43170873	59228698	66913728	93578868
	128284170	121674782	86558608	59643831	43064712	25304216	8195291	4559378
row 2	4938792	675942	2452253	8324292	5690308	7102326	20697596	52448429
	101468128	88769774	27181030	6355585	11851906	7231731	9316530	7123757
row 3	3120531	761266	196744	1107178	1822803	997881	3783886	4541426
	4537683	8553284	2138247	3648197	218924	101250	1601640	3519395
row 4	365327	122468	129775	640140	500801	296067	1056788	3787023
	3904941	1344099	325773	3895776	1907983	116675	1028774	2049424

Table 6-9: The output of the matrix powers for block 20 of wsq file `103_1.wsq`.

It turns out that the ρ-VEX puts no values into `powersgeheugen`. But the program does return properly to the calling function in `_start.s`, and runs for about 577636 cycles. We checked this with results from the simulator and there the result was computed after 754815 cycles, so there is a discrepancy. It seems like the function `sum_rot_block_rows2()` is returning too early to the main-function, without writing its results. We suspect a bug in the ρ-VEX but were not able to find it. We did, however, find some peculiarities in the source files of the implementation of the ρ-VEX, as discussed below.

In `rvex_system.vhd` we found some errors in the dimensions of some vectors. Below are shown the declarations in the architecture body behavioural of `rvex_system`. The signal `imem_write_address` was wrongly dimensioned. For the top of the index it was stated `ADDR_WIDTH + 1`, but it should be `ADDR_WIDTH - 1`. The same was true in some other entries of the file. We corrected all of them.

```
signal reset_s : std_logic := '0';

signal clk_half : std_logic := '0';

signal address_dr_s  : std_logic_vector((DMEM_LOGDEP - 1) downto 0) := (others => '0');
signal address_dw_s  : std_logic_vector((DMEM_LOGDEP - 1) downto 0) := (others => '0');
signal write_en_dm_s : std_logic_vector(3 downto 0);
```

```
signal dm2rvex_data_s : std_logic_vector((DMEM_WIDTH - 1) downto 0);
signal rvex2dm_data_s : std_logic_vector((DMEM_WIDTH - 1) downto 0) := (others => '0');

signal done_s         : std_logic                                   := '0';
signal cycles_s       : std_logic_vector(31 downto 0)               := (others => '0');
signal address_uart_s : std_logic_vector((DMEM_LOGDEP - 1) downto 0) := (others => '0');
signal data_uart_s    : std_logic_vector(31 downto 0);

signal flush_s, clear_s : std_logic                       := '0';  -- flush for fetch stage
signal mpc_r   : std_logic_vector((ADDR_WIDTH - 1) downto 0);  -- pc to read instruction of i_mem
signal instr_s : std_logic_vector(127 downto 0) := (others => '0');  -- instruction from i_mem

signal dmem_write_enable : std_logic_vector(3 downto 0);
signal dmem_address      : std_logic_vector((DMEM_LOGDEP - 1) downto 0);
signal dmem_write_data   : std_logic_vector(31 downto 0);
signal dmem_read_data    : std_logic_vector(31 downto 0);

signal imem_write_address : std_logic_vector(ADDR_WIDTH - 1 downto 0);
signal imem_write_enable  : std_logic;
signal imem_write_data    : std_logic_vector(31 downto 0);
signal imem_read_data     : std_logic_vector(31 downto 0);
signal status_data_out_s  : std_logic_vector(31 downto 0);
```

The following variable was wrongfully defined:

```
imem_write_address <= mem_write_address(ADDR_WIDTH -1 downto 0);
```

It said `ADDR_WIDTH + 1` and should be changed to the value shown above.

6-6-2 Expected Speedup for the Computation of one DFT Block

Due to the fact that the ρ-VEX does not produce results, it was not possible to determine execution times for the partitioned application. However, we were able to determine the times it took to load the data to the ρ-VEX as well as the time it took to retrieve the data from the ρ-VEX. Together with the execution time obtained from simulating the execution of the kernel on the ρ-VEX which computes the DFT computation of one block of data, we can make an estimation of the speedup compared to the computation of the same block on the MicroBlaze. In the table, see table 6-10, we have shown the results of writing and reading of data from the ρ-VEX.

Direction	Time [cycles]
To ρ-VEX	7841
From ρ-VEX	1305

Table 6-10: Transportation times between the MicroBlaze and the ρ-VEX.

From this data, together with the execution time obtained from the simulator we can determine an execution time for the computation of one DFT-block on the ρ-VEX, that includes the transportation times of the data. The total execution time is:

$$\text{total\_execution\_time} = 7841 + 1305 + 754815 = 763961 \text{ cycles}$$

We have also determined the execution time for several blocks of data executed on the MicroBlaze. This gave an average execution time of 603493 cycles. In table 6-11 we have shown the execution times for one block on both the Microblaze and the ρ-VEX.

Processor	Time [cycles]
MicroBlaze	603493 (average)
ρ-VEX	763961 (estimation on simulator)
ρ-VEX	530911 (optimized)

Table 6-11: Execution times for computing one DFT block on both the MicroBlaze and ρ-VEX.

We are now able to determine a speedup for the computation of one block of data based on these results. Speedup is defined as [7]:

$$\text{Speedup} = \frac{\text{Execution\_time\_unaccelerated}}{\text{Execution\_time\_accelerated}}$$

With this definition and the execution times in table 6-11 we obtain a speedup of:

$$\text{Speedup} = \frac{603493}{763961} = 0.7899$$

So, the results from the simulator suggest that we obtain no speedup with the ρ-VEX. With the best optimization for the kernel we obtain an execution time of 521765 cycles. With the times for transportation of data we get a total execution time of 530911 cycles, which results in a speedup result of:

$$\text{Speedup} = \frac{603493}{530911} = 1.1367$$

So, in order to obtain a speedup we have to apply inlining and loop unrolling to the kernel on the ρ-VEX. Although this speedup is modest it can result in a significant speedup for the whole application because the DFT computations for the blocks of data is performed tens of times for one wsq file, but the exact amount depends on the specific wsq file.

6-6-3 Determination of Actual Speedup

Because the ρ-VEX did not work, we were not able to obtain actual execution times for the application. We wanted to measure the execution time of the minutiae extraction part of the partitioned application in order to determine a speedup value. Furthermore, we wanted to obtain the execution time of the complete application to compare this against the execution time of 11 minutes and 23 seconds for the original application, which we described earlier.

Chapter 7

Conclusions

In this bachelor thesis, it is presented how to accelerate the minutiae extraction process within the big Fingerprint Verification Application using the ρ-VEX although this application also appears to be a good choice to prove the quality of ρ-VEX. There are several steps that have been taken to obtain the acceleration of the application. First, the application has been run on the VEX simulator to analyze the effects of different issue widths on the execution time. The results show that it is promising to concentrate the hardware design effort on 4 issue-width. Second, the application has been profiled on the MicroBlaze soft core processor. The main motive of this profiling was to determine which parts cost the most execution-time during the minutiae extraction process. These parts of the application are going to be accelerated further. The important reason of accelerating only the kernel part of the application is that the ρ-VEX has not enough memory to execute the whole application since the fingerprint application is a very big application. Also the FPGA board would be overload if the whole application is tried to be accelerated at once. In order to make the fingerprint application running on MicroBlaze and ρ-VEX, the application has been modified by removing the `fprintf` and `printf` calls (libc call). After this adjustment, the application is ready to be run on MicroBlaze. The profiling result shows that the function code `sum_rot_blocks_rows2(..., ...)` and `long_int_math.c` take the most execution time. The third step, this code which is included in the folder `mindtct` has been implemented on the ρ-VEX processor and the result has been used in the main application, running on MicroBlaze. When optimizing the kernel, it is obvious that the application is accelerated using ρ-VEX. This has been proven in section 6.5. The different compiler options have been evaluated in order to obtain the best acceleration. Loop unrolling and inlining optimization methods give a substantial acceleration. Using loop unrolling method indicates a factor of 1.1 faster than no loop unrolling. Inlining caused also a substantial acceleration. This provide a relative speedup with factor 1.39 faster than before inlining. The combination of loop unrolling with inlining gives the best acceleration (see Table 6-8) However, it increased the amount of instructions. With the ρ-VEX, only minimal speedup can be obtained. With all optimizations applied we only obtained a speedup of 1.13 for the computation of one DFT block. Speedup can still be obtained because in the application several blocks are computed resulting in a cumulative reduction of execution cycles.

Bibliography

[1] S. Heng, "Embedded systems–an (inconspicuous) key technology in the ascendancy," Deutsche Bank Research, Tech. Rep. 11, Feb. 2001, www.dbresearch.com.

[2] W. Wolf, *High Performance Embedded Computing*. Morgan Kaufmann, 2007.

[3] A. C. Sodan *et al.*, "Parallelism via multithreaded and multicore CPUs," *Computer*, vol. 43, no. 3, pp. 24–32, Mar. 2010.

[4] R. Kumar *et al.*, "Heterogeneous chip multiprocessors," *Computer*, vol. 38, no. 11, pp. 32–38, Nov. 2005.

[5] W. Wolf, *Computers as Components: Principles of Embedded Computing System Design*. Morgan Kaufmann, 2001.

[6] S. Dutta *et al.*, "A methodology to evaluate memory architecture design tradeoffs for video signal processors," *IEEE Transactions on Circuit and Systems for Video Technology*, vol. 8, no. 1, pp. 36–53, Feb. 1998.

[7] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1996.

[8] S. Wong and F. Anjam, "The Delft reconfigurable VLIW processor," in *17th International Conference on Advanced Computing and Communications (ADCOM 2009)*, December 2009, pp. 244–251.

[9] J. A. Fisher, P. Faraboschi, and C. Young, *Embedded Computing: A VLIW Approach to Architecture, Compilers, and Tools*. Morgan Kaufmann, 2005.

[10] M. Chiodo *et al.*, "Hardware-software codesign of embedded systems," *IEEE Micro*, vol. 14, no. 4, pp. 26–36, Aug. 1994.

[11] K. Bertels *et al.*, "HArtes: Hardware-software codesign for heterogeneous multicore platforms," *IEEE Micro*, vol. 30, no. 5, pp. 88–97, Sep. 2010.

[12] D. Maltoni, D. Maio, A. Jain, and S. Prabhakar, *Handbook of Fingerprint Recognition*. Springer-Verlag, 2003.

[13] NIST, "User's guide to nist fingerprint image software (nfis)."

[14] R. Seedorf, "Fingerprint verification on the VEX processor," Master's thesis, Delft University of Technology, 2010.

[15] M. van der Net, "A SoC solution for fingerprint minutiae extraction," Master's thesis, Delft University of Technology, 2008.

[16] "Configurable embedded system design with xilinx fpgas," http://www.xilinx.com/publications/prod_mktg/TDP_Embedded_Product_Brief.pdf, 2011.

[17] "Virtex-6 fpga board," http://www.xilinx.com/support/documentation/data_sheets/ds150.pdf, 2011.

[18] "Microblaze processor reference guide, embedded development kit edk 10.1," http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_1/mb_ref_guide.pdf, 2011.

[19] "Example microblaze system for Spartan-6 and Virtex-6," http://www.xilinx.com/images/ise/microblaze_v6s6fpgas.jpg, 2011.

[20] R. Seedorf, F. Anjam, A. Brandon, and S. Wong, "ρ-VEX: A parameterized VLIW processor," http://ce.et.tudelft.nl.

[21] T. van As, "ρ-VEX: A reconfigurable and extensible VLIW processor," Master's thesis, Delft University of Technology, September 2008.

[22] "Coreconnect architecture - processor local bus (plb)," http://www.xilinx.com/ipcenter/processor_central/coreconnect/coreconnect_plb.htm, 2011.

[23] "Ds531 processor local bus (plb) v4.6," http://www.xilinx.com/support/documentation/ip_documentat
2011.

[24] P. Faraboschi, G. Brown, J. Fisher, G. Desoll, and F. Homewood, "Lx: a technology platform for customizable vliw embedded processing," in *Computer Architecture, 2000. Proceedings of the 27th International Symposium on*, june 2000, pp. 203 –213.

Appendix A: Input converter

The original application opens a wsq-file and reads the contents, but with when the application has to run on the Microblaze and ρ-VEX it cannot read the file because there is no file I/O support on those platforms. A program in Python has been written that takes care of that. It reads in a wsq file and outputs a C-source file called `imagestub.c`, containing the data in the wsq file. This C-source file has to be included into the project and the data is put into a memory block which is created for this purpose. The application has been modified accordingly so that it no longer reads in the wsq file but instead uses the content of the filled memory block.

```
#Programma om wsq-files in te lezen en om te zetten
#naar een C-source file waar de data in een
#array wordt gezet.

import sys
import os
from datetime import date

if len(sys.argv) != 2:
    print 'Usage: python wsqconvert.py test.wsq'
    sys.exit()

wsqpath = sys.argv[1]

try:
    wsqfile = open(wsqpath, 'rb')
except IOError:
    print 'Could not find %s. Check spelling of name and path.' % wsqpath
    sys.exit()

print 'Reading:', wsqpath
wsqdata = wsqfile.read()
wsqfile.close()
N = len(wsqdata)

(wsqpathhead, wsqpathtail) = os.path.split(wsqpath)
cfilepath = os.path.join(wsqpathhead, 'imagestub.c')
cfile = open(cfilepath, 'w')

now = date.today()

#Hier maken we de C-source file aan de hand van de data in wsqdata.
print 'Writing: imagestub.c...',
```

```
35  textblock = """/*************************************************************************
    LIBRARY: IMAGE - Image Manipulation and Processing Routines
    FILE:  IMAGESTUB.C
    AUTHOR:  Automatically generated\n"""

40  cfile.write(textblock)

    cfile.write('WSQ FILE: ' + wsqpathtail + '\n')
    cfile.write('DATE: ' + now.strftime("%d %b %Y") + '\n')

45  textblock = """*************************************************************************
    Contains routines responsible for emulating reading a wsq compressed image from
    a file.
    *************************************************************************/

50  static int fillArray(unsigned char *memptr);

    /*************************************************************************
    * Creates a block of memory and puts the content of a wsq-compressed
    * image file into this block of memory.
55  * Emulates reading the corresponding file.
    *************************************************************************/
    """

    cfile.write(textblock)
60
    textblock = """int fill_memory_with_raw(unsigned char **odata, int *ofsize)
    {
       int fsize;
       unsigned char *idata;
65  """

    cfile.write(textblock)

    cfile.write('   fsize = ')
70  cfile.write(str(N))
    cfile.write(';\n')

    textblock = """
       /* allocate bytes to store bytes of file*/
75     idata = malloc(fsize * sizeof(unsigned char));
       if(idata == 0){
          print("ERORR BUFFER NOT ALLOCATED fill_memory_with_raw : malloc : idata==0.");\n
          return(-3);
       }
80
       fillArray(idata);

       *odata = idata;
       *ofsize = fsize;
85
       return(0);
    }\n\n"""

    cfile.write(textblock)
90
    textblock = """static void fillArray(unsigned char *memptr)\n"""

    cfile.write(textblock)

95  cfile.write('{\n')

    for i in xrange(N):
        cfile.write('\t*memptr = 0x%x;\n' % ord(wsqdata[i]))
        cfile.write('\tmemptr++;\n')
100
```

```
    cfile.write('}')
    print 'Done.'
105 cfile.close()
```

Appendix B: VEX Accelerator

```
/*
 * vexaccelerator.c
 *
 *  Created on: Jun 20, 2011
 *      Author: A.J.J. Regeer
 */

#define PTR_TO_DATA_ARRAY (0x00009400)
#define PTR_TO_POWERSGEHEUGEN (0x00009300)

#include "vexaccelerator.h"
#include "fixedpointmath.h"
#include "xparameters.h"
#include "rvex.h"
#include "compute_dft_prog.h"
#include "platform.h"

extern int debug;
static int kernel_done;


/*
     De nieuwe vervangende functie voor sum_rot_block_rows2.
     Start uiteindelijk een programma op de VEX waar dezelfde
     berekeningen worden uitgevoerd. De functie schrijft de data
     naar het geheugen van de VEX en wacht daarna totdat de VEX
     klaar is. Daarna leest het de resultaten uit het geheugen van de
     VEX en schrijft ze in powers. De definitie van powers is te vinden
     in init.c in de folder mindtct.
*/
int sum_rot_block_rows2_onVEX(const unsigned char *blkptr, ufp27p5_t **powers)
{
        Xuint32 baseaddr, timeout;
        RVexCtr *RVex;
        int i,j;
        unsigned int dword;

        RVex = &RVexInstances[0];
        baseaddr = RVex->BaseAddress;

        /* Stop en Reset de VEX */
        rvex_stop(RVex->BaseAddress);
        rvex_reset(RVex->BaseAddress);

        // Schrijf het data-blok naar de VEX
        for (i = 0; i < 1156; i+=4){
                // Maak het word dat naar het VEX-geheugen wordt geschreven.
                // LET OP: Dit gaat er vanuit dat de VEX BIG ENDIAN is.
```

```
                //
                dword = ((unsigned int)(blkptr[i]))<<24 |
                                ((unsigned int)(blkptr[i+1]))<<16 |
                                ((unsigned int)(blkptr[i+2]))<<8 |
                                ((unsigned int)(blkptr[i+3]));
                // schrijf naar datageheugen van rvex.
                rvex_write_mem(RVex->DataMemAddress, (PTR_TO_DATA_ARRAY + i)/4, dword);
        }

        // Voer de kernel-task uit op de VEX
        rvex_start(RVex->BaseAddress);

        // Wacht voor resultaat in een loop
        kernel_done = 0;
        for (timeout = 0; timeout < RVEX_BREAK_TIMEOUT; timeout++)
        {
                if (RVex->Status & RVEX_DONE)
                {
                        kernel_done = 1;
                        break;    // Finished
                }
        }

        // Indien fout dan terug met foutcode
        if (kernel_done == 0)
        {
                print("Timeout bij Kernel!!!!\n");
                return 1; // Return on Error
        }

        /* Stop de VEX */
        rvex_stop(RVex->BaseAddress);

        /* Als succesvol dan het resultaat uit geheugen van VEX schrijven naar powers.   */
        for (i = 0; i < 4; i++){
                for (j = 0; j < 16; j++){
                        /* Read resonance coefficients from DMEM on VEX */
                        powers[i][j] = rvex_read_mem(RVex->DataMemAddress, PTR_TO_POWERSGEHEUGEN/4 + i*16 + j);
                }
        }

        return 0; // success.
}

XStatus initialize_VEX(void)
{
        Xuint32 baseaddr, result;
        RVexCtr *RVex;

        /* Initialiseer het platform: UART etc. */
        init_platform();

        /* Initialiseer het DFT-programma. */
        RVexPrograms[0] = RVEX_PROG_computeDFT;

        /* Initialiseer rvex-instantie: 0 */
        RVexInstances[0].BaseAddress            = XPAR_RVEX_PLB_WRAPPER_0_BASEADDR;
        RVexInstances[0].DataMemAddress         = XPAR_RVEX_PLB_WRAPPER_0_MEM0_BASEADDR;
        RVexInstances[0].InstrMemAddress        = XPAR_RVEX_PLB_WRAPPER_0_MEM1_BASEADDR;
        RVexInstances[0].StatusMemAddress       = XPAR_RVEX_PLB_WRAPPER_0_MEM2_BASEADDR;
        RVexInstances[0].Status                         = 0;


        RVex = &RVexInstances[0];
        baseaddr = RVex->BaseAddress;
```

```
115         /* Initialiseer de interrupts */
            initialize_intc();
            result = rvex_init_interrupts(RVex, XPAR_XPS_INTC_0_RVEX_PLB_WRAPPER_0_IP2INTC_IRPT_INTR);

            if (result == XST_FAILURE) {
120                 xil_printf("Interrupts initialization failed!\r\n");
                    return result;
            }
            else {
                    xil_printf("Interrupts initialization OK!\r\n");
125         }

            /* Zet het programma in het geheugen van de VEX. */
            if (rvex_setup(RVex, 0) == XST_FAILURE) {
                    xil_printf("\n\r VEX setup niet gelukt.\r\n");
130                 return XST_FAILURE;
            }

            rvex_print_status(RVex);

135         if (debug > 0){
                    print("Initialisering van VEX gedaan.");
            }

            return XST_SUCCESS;
140 }
```

Appendix C: Makefile for the VEX program

```
CFLAGS = -S -fno-xnop -fexpand-div -fmm=pipe_1_4.mm
CC = ~/VEX-Tools/vex-3.43/bin/cc

DFTHEADERS = dft_lookup.h fixedpointmath.h fixedpointmathcode.h grid_lookup.h lfs.h long_int_math.h
MATHHEADERS = fixedpointmath.h

computeDFT : computeDFT.o long_int_math.o _start.o
	~/BinTools/ld/ld-new -o computeDFT _start.o computeDFT.o long_int_math.o

computeDFT.s : computeDFT.c $(DFTHEADERS)
	$(CC) $(CFLAGS) computeDFT.c

long_int_math.s : long_int_math.c $(MATHHEADERS)
	$(CC) $(CFLAGS) long_int_math.c

computeDFT.o : computeDFT.s
	~/BinTools/gas/as-new -o computeDFT.o computeDFT.s

long_int_math.o : long_int_math.s
	~/BinTools/gas/as-new -o long_int_math.o long_int_math.s

_start.o : _start.s
	~/BinTools/gas/as-new -o _start.o _start.s

vhdl: computeDFT
	~/BinTools/binutils/elf2vhd computeDFT

clean :
	rm computeDFT computeDFT.s long_int_math.s computeDFT.o
	long_int_math.o _start.o d_mem.vhd i_mem.vhd prog.h system_tb.vhd
```

Appendix D: C source file computeDFT.C

```
#include "lfs.h"
#include "dft_lookup.h"
#include "grid_lookup.h"
#include "fixedpointmath.h"

unsigned char data[1156];
ufp27p5_t *powersarrays[4];
ufp27p5_t powersgeheugen[64];

void sum_rot_block_rows2(const unsigned char *blkptr, ufp27p5_t **powers)
{
   int i, ix, iy, gi, w, dir;
   int rowsums[24];
   fp15p17_t fp_cospart, fp_sinpart; /* FIXED-POINT */


   /* Initialize rotation offset index */
   gi = 0;

   /* Foreach direction ... */
   for(dir = 0; dir < 16; dir++)
   {
          /* Re-initialize rotation offset index. */
          gi = 0;

          /* For each row in block ... */
          for(iy = 0; iy < 24; iy++)
          {
                /* The sums are accumlated along the rotated rows of the grid, */
                /* so initialize row sum to 0.                                  */
                rowsums[iy] = 0;

                /* Foreach column in block ... */
                for(ix = 0; ix < 24; ix++) {
                    /* Accumulate pixel value at rotated grid position in image */
                    rowsums[iy] += *(blkptr + gridoffsets[gi + (dir * 576)]);
                    gi++;
                }
          }

          /* Foreach DFT wave ... */
          for(w = 0; w < 4; w++)
          {   /* Initialize accumulators */
                fp_cospart = 0; /* FIXED-POINT */
```

```
                fp_sinpart = 0; /* FIXED-POINT */

                /* Accumulate cos and sin components of DFT. */
                for(i = 0; i < 24; i++) {
                    /* Each rotated row sum is multiplied by its corresponding cos and sin point in DFT wave */
                    fp_cospart += mulfp15p17(itofp15p17(rowsums[i]),
                        dft_waves_cos[i + (w * 24)]); /* FIXED-POINT */
                    fp_sinpart += mulfp15p17(itofp15p17(rowsums[i]),
                        dft_waves_sin[i + (w * 24)]); /* FIXED-POINT */
                }

                /* Power is sum of squared cos and sin components */
                powers[w][dir] = mulfp27p5(fp_cospart>>12, fp_cospart>>12) +
                mulfp27p5(fp_sinpart>>12, fp_sinpart>>12); /* FIXED-POINT */
            }
    }
}

int main(void)
{
        powersarrays[0] = &powersgeheugen[0];
        powersarrays[1] = &powersgeheugen[16];
        powersarrays[2] = &powersgeheugen[32];
        powersarrays[3] = &powersgeheugen[48];

        sum_rot_block_rows2(data, powersarrays);

        return 0;
}
```

Appendix E: Startup Assembler Code

```
1           .section .text
            .proc
    _start::
            c0      add $r0.1 = $r0.0, 0xFF00
5   ;;
            c0      call $l0.0 = main
    ;;
            c0      stop
    ;;
10          c0      nop
    ;;
            .endp
```

Appendix F: Configuration File

```
# 4-issue vex default cluster
CFG: Debug 0
RES: IssueWidth 4
RES: MemLoad 1
RES: MemStore 1
RES: MemPft 1
RES: IssueWidth.0 4
RES: Alu.0 4
RES: Mpy.0 2
RES: CopySrc.0 2
RES: CopyDst.0 2
RES: Memory.0 1
DEL: AluR.0 2
DEL: Alu.0 2
DEL: CmpBr.0 2
DEL: CmpGr.0 2
DEL: Select.0 2
DEL: Multiply.0 2
DEL: Load.0 2
DEL: LoadLr.0 2
DEL: Store.0 2
DEL: Pft.0 2
DEL: Asm1L.0 2          ## user defined assembly intrinsics
DEL: Asm2L.0 2
DEL: Asm3L.0 2
DEL: Asm4L.0 2
DEL: Asm1H.0 2
DEL: Asm2H.0 2
DEL: Asm3H.0 2
DEL: Asm4H.0 2
DEL: CpGrBr.0 2
DEL: CpBrGr.0 2
DEL: CpGrLr.0 2
DEL: CpLrGr.0 2
DEL: Spill.0 2
DEL: Restore.0 2
DEL: RestoreLr.0 2
REG: $r0 62
REG: $b0 8

```