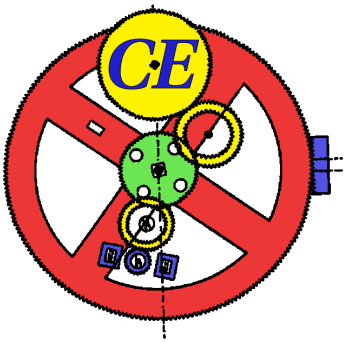

Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2011

MSc THESIS

Fingerprint Verification on the VEX Processor

Roël Seedorf

CE-MS-2010-27

Abstract

The speed gap between a processor realized in Semicustom ASIC technology and a processor realized in FPGA technology is narrowing. In processor design, the approach is to define the microarchitecture of the processor and to design and implement it for executing an application domain. In this thesis, we have investigated the approach to design a customized and parameterized VLIW processor, such that it becomes better suited for imaging applications. We analyzed the requirement of a fingerprint application. Porting the application to the VEX simulator required design and integration of an arithmetic module for emulating 64-bit arithmetic. Subsequently, the application was ported to the VEX simulator such that we could derive the set of architectural parameters for the VEX processor. Based on these parameters, we designed a pipelined and scalable VLIW processor according to the VEX ISA. Moreover, the design of the processor was implemented and realized in a Virtex-VI FPGA. Furthermore, to create the instruction memory for the VEX processor, the required toolchain was developed from the inherited development framework of the ρ-VEX project. For this purpose, we extended the back-end of the VEX compiler, re-designed the assembler, integrated the linker and the loader. The resulting toolchain allowed us to extract and execute the fingerprint application's bottleneck and three other benchmarks, on the VEX processor.

Compared to the inherited development framework, the re-designed toolchain is maturer and better to utilize, as it is able to create the instruction ROM for the VEX processor. The Fibonacci benchmark results showed that the VEX processor is 6.64 times faster than the original multi-cycle ρ-VEX processor. Moreover, the results of three other benchmark indicate that our designed processor is able to execute a range of applications.

TU Delft

Faculty of Electrical Engineering, Mathematics and Computer Science

Fingerprint Verification on the VEX Processor

Making a VLIW Processor and its Toolchain useful for real-life applications

---

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Roël Seedorf
born in Paramaribo, Suriname

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

Fingerprint Verification on the VEX Processor

by Roël Seedorf

Abstract

The speed gap between a processor realized in Semicustom ASIC technology and a processor realized in FPGA technology is narrowing. In processor design, the approach is to define the microarchitecture of the processor and to design and implement it for executing an application domain. In this thesis, we have investigated the approach to design a customized and parameterized VLIW processor, such that it becomes better suited for imaging applications. We analyzed the requirement of a fingerprint application. Porting the application to the VEX simulator required design and integration of an arithmetic module for emulating 64-bit arithmetic. Subsequently, the application was ported to the VEX simulator such that we could derive the set of architectural parameters for the VEX processor. Based on these parameters, we designed a pipelined and scalable VLIW processor according to the VEX ISA. Moreover, the design of the processor was implemented and realized in a Virtex-VI FPGA. Furthermore, to create the instruction memory for the VEX processor, the required toolchain was developed from the inherited development framework of the ρ-VEX project. For this purpose, we extended the back-end of the VEX compiler, re-designed the assembler, integrated the linker and the loader. The resulting toolchain allowed us to extract and execute the fingerprint application's bottleneck and three other benchmarks, on the VEX processor.

Compared to the inherited development framework, the re-designed toolchain is maturer and better to utilize, as it is able to create the instruction ROM for the VEX processor. The Fibonacci benchmark results showed that the VEX processor is 6.64 times faster than the original multi-cycle ρ-VEX processor. Moreover, the results of three other benchmark indicate that our designed processor is able to execute a range of applications.

Laboratory : Computer Engineering
Codenumber : CE-MS-2010-27

Committee Members :

Advisor:	dr.ir. Stephan Wong, CE, TU Delft
Advisor:	MSc. F. Anjam, CE, TU Delft
Chairperson:	dr.ir. K.L.M. Bertels, CE, TU Delft
Member:	dr.ir. T.G.R.M. van Leuken, CE, TU Delft
Member:	dr.ir. A. van Genderen, CE, TU Delft

Dedicated to the loving memories of my parents Emerenthia Starke Seedorf and Roël George Imro Seedorf

Contents

List of Figures

List of Tables

Acknowledgements

This thesis presents the results of my graduation project performed in the last year at the Computer Engineering Laboratory of the Electrical Engineering Department, Delft University of Technology.

First, I would like to thank the most High Almighty God for guiding my steps through out my master studies and providing me with strength, determination and wisdom. During my MSc project I came across many people who supported and assited me. My unending gratitude, love and acknowledgements to you all.

I would like to thank my advisor dr. Stephan Wong for giving me the oppurtunity and advising me while performing my graduation project at the Computer Engineering Research Group. I want to express my gratitude for his patience, confidence and continuos support which have all contributed to my thesis work.

I want to thank Phd student Fakhar Anjam for advising me while I performed the project work. He was always ready to assist me during my graduation project.

My acknowledgements go out to my friend Gertjan Schoneveld for the many discussions and conversations we had about computer architecture and the future of embedded computing. Furthermore, I want to express my gratitude to my friend Marcel Helmer for offering me a second work place where I could concentrate and ponder on my graduation project. I would also like to thank Pastor Sunday Igbalaloju for his encouregments and prayers.

Finally, I want to thank my wife Nancy for her continued support and our daughter Chevenie for her inspiration.

Roël Seedorf
Delft, The Netherlands
June 15, 2010

1 Introduction

While in the field of general-purpose computing the competition between dynamic-issue superscalar and static-issue Very Long Instruction Word (VLIW) processors has many times been decided in favor of dynamic-issue superscalar processors, the story is different for the embedded computing field. Within the Embedded Systems (ES) market customers' increasing demand to run evermore applications on their devices, is driving processor technology to achieve higher performance at low-power consumption [1]. Although the requirements of performance and power consumption are competing from the perspective of a hardware designer, we argue that the VLIW architectural paradigm gives processors designers the ability to satisfy both requirements [2]. For application developers, the market developments mean that the complexity of their applications continues to increase. At the same time, the product development life cycle is becoming shorter. As a consequence, application developers have less time to design their applications. Furthermore, hardware experts are forced to keep in pace with evolving multimedia standards and customers demands for higher performance [1]. They have to design the processor in a shorter time that can deliver enough performance for running a range of complex applications. The embedded programmer mediates between the application developer and the hardware expert, as he needs to quickly adapt the underlying hardware such that it can execute the application, while he offers flexibility to the application developer.

This thesis describes the design and implementation of a softcore VLIW processor. The fingerprint application was utilized for evaluation, debug, and proof of concept purposes. In the light of above the discussion, we show the importance of developing the required toolchain for the designed processor.

The organization of this chapter is as follows: the motivation for the performed work is presented in Section 1.1. Subsequently, Section 1.2 describes the project and gives an overview of the conducted research. In Section 1.3, we describe the scope of the performed work. Section 1.4 concludes this chapter by presenting this thesis' framework.

1.1 Motivation

Our project originates from two projects and integrates their research in order to develop the processor such that a fingerprint application can be utilized for evaluation, debug and proof of concept purposes.

The application that we focused on originates from a project that was performed by Michel van der Net, about a System on Chip (SoC) solution to perform fingerprint minutiae extraction. That project's research was performed within the Circuits And System Group (CAS) of TU Delft. The performed work delivered a System on Chip (SoC) solution consisting of a modified version of the MINDTCT system that was executed on a LEON2 micro-controller [3]. The result showed that it was possible to accelerate the fingerprint application by using limited amounts of reconfigurable hardware resources. The SoC resources consisted of the LEON2 microcontroller with two attached accelerators. Although the computational kernels were being sped up by 40% with two accelerators, the total execution time was still 61.14 seconds. Furthermore, there were communication problems between the designed accelerators and the LEON2 microcontroller. It was clear that in practice when the system became operational, it would take too long to recognize a persons's fingerprint.

The second project was done by Thijs van As within the MOLEN research theme. The main goal was to design and implement a reconfigurable and extensible VLIW processor that conformed to the VEX Instruction Set Architecture (ISA) [4]. This project delivered a configurable and extensible VLIW processor called the ρ-VEX together with an applications development framework.[1] From the application's perspective, we investigated a different approach. The idea in this projet was to revisit the MINDTCT application and try to determine whether there was more potential Instruction Level Parallelism (ILP) present. This ILP can be extracted to keep a more powerful processor busy in order to decrease the total execution time. Therefore, we saw an opportunity to design such a processor and put it to the test by evaluating the fingerprint application on it. Consequently, from the perspective of a hardware designer we could utilize this application as a test case for the designed processor.

1.2 Project Description

This project aims to design a parameterized VLIW processor based on the VEX ISA. The fingerprint minutiae extraction application is for evaluation, debug, and proof-of-concept purposes. Both the fingerprint application and the target processor are of primary importance in our research. The hardware that we started with was from the ρ-VEX project. This project delivered and equally named VLIW processor that was implemented on the Virtex 2-Pro FPGA. On the application side, we utilized a fingerprint minutiae extraction application that is called MINDTCT. It was originally designed by the National Institute of Standards and Technology (NIST) [6]. Our choice for this fingerprint application is motivated in Chapter 2.

[1] The term *reconfigurable* is reserved for a processor that is run-time configurable as the ρ-Trimedia processor [11].

1.2.1 Context

The context in which we performed the project work is described in the following. Personal identification based on fingerprint recognition offers a secure mean of accessing private information. The utilized recognition algorithms are very computationally intensive, yet their implementation in an Embedded System (ES) must be fast enough to allow for real-time execution. When the architecture of the host processor is more or less fixed, the standard approach in dealing with the application's bottlenecks is to accelerate them in (reconfigurable) hardware [3, 5]. When this is not the case, a more interesting approach is to extent or adjust the processor's architecture such that it becomes better suited for the target application. The ρ-VEX processor is a static-issue, design time configurable and parameterized VLIW processor. The implementation of the ρ-VEX processor has the ability to execute design time configurable operations by customizing it's functional units to the target application [4].

Besides a minutiae extractor, a practical fingerprint verification system also contains a matcher. The purpose of this matcher is to compare the extracted minutiae with a pre-stored template in the memory system. Consequently, the fingerprint verification system can decide whether the captured minutiae are the same as the pre-stored ones. In the verification system this is the slowest part, since the comparison is performed as a one to one search process. Hence, it can be executed fast. This justifies why the matcher is out of the scope of this project. A conceptual view of such a system is depicted in Figure 1.1.

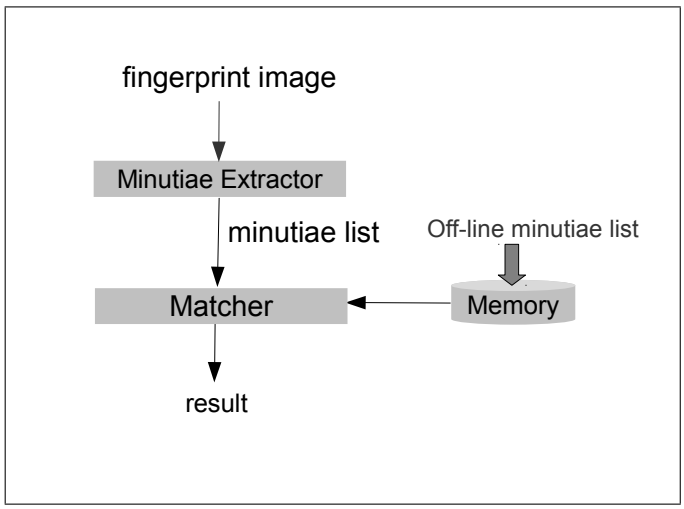

Figure 1.1: Overview of online matcher

1.2.2 Problem Definition

The overall problem statement of this project is defined as follows. We want to develop a VLIW processor that can deliver more performance than the ρ-VEX processor. In order to solve this problem three objectives are identified. First, we need to study and port the fingerprint application to the VEX simulator. This enables us to analyze and parallelize

the execution of the fingerprint application on the simulator. Furthermore, it allows us to derive a set of architectural parameters for the target VLIW processor. Second, to develop a VLIW processor to execute the parallelism that the VEX compiler is able to extract from the fingerprint application. Third, to develop the processor's toolchain from the inherited development framework of the ρ-VEX processor. The purpose of this objective is to devise a toolchain that can be utilized for generating the instruction memory of the VLIW processor.

1.3 Workplan

In the following we describe the overall strategy of the the workplan in order to reach our objective.

1. Convert the research idea into a project description.
2. Define the project work from the project description.
3. Devise the project planning from the project description.
4. Research the application and the ρ-VEX architecture.
5. Explore and study the source code of the application.
6. Port the target application to the VEX simulator.
7. Analyze, profile and optimize the target application.
8. Derive the architectural parameters for the required processor.
9. Develop the required processor from the architectural parameters.
10. Develop the required toolchain for the designed processor.
11. Verify and benchmark the designed VLIW processor.

The project starts with the initial phase, during which the research idea is converted to a project description [24]. This description enables a precise definition of the project's work. Subsequently, we devise a project planning. The definition phase requires extensive literature research on the application and the architecture of the target processor. The aim is to achieve understanding of both the application and the target processor. Subsequently, we proceed to explore and study the application's source code. This step serves as a preparation to port the target application to the VEX simulator. This simulator is a cycle accurate tool which is provided by Hewlett-Packard Laboratories [15]. Hereafter, we port the application to the VEX simulator, since this is necessary as it provides us the performance estimate of executing the application on the target processor. Moreover, this step provides us a set of architectural parameters of the target processor.

After this phase of the project is completed, the design phase can be entered. In this phase, we analyse the application that runs on the simulator and we discuss the analysis results. Subsequently, we choose the most promising architecture for the target

processor. This provides us the architectural parameters of the processor that has to be developed. After this step, we move on to develop the required processor. Furthermore, it is required to develop the processor's toolchain. Finally, we verify the target processor by benchmarking it, and by utilizing the target application as a test case.

1.4 Thesis Overview

The remainder of this dissertation has the following structure. Chapter 2 discusses related background information for this project. Chapter 3 presents the requirements and the specifications of the System design. Chapter 4 explains developing the fingerprint application and porting it to the VEX simulator. In Chapter 5, we critically analyze the design of the ρ-VEX processor, discuss its limitations and proposes a novel processor design called the VEX processor to deal with these limitations. Subsequently, Chapter 6 discusses how we implemented the VEX processor and fit it into the system architecture. After the implementation phase it became clear that in order to make the development of the fingerprint application less cumbersome, we had to come up with a better toolchain. Chapter 7 describes how we accomplished this. Chapter 8 presents the experiments that we performed and discusses the results in order to demonstrate the benefits of the proposed solution. Chapter 9 concludes this dissertation, reflects on the main contributions and suggests future research work.

Minutiae Extraction Application

2

In this chapter, we discuss related background information of the fingerprint minutiae extraction application. The knowledge that we obtain provides us understanding about the workings and requirements of this application. We investigate a modified version of the MINDTCT software package from the National Institute for Standards and Technology (NIST). This software package is programmed in the C language and describes the biometric technique to perform the extraction of minutiae from an fingerprint image. The original target processing platform was a SPARC general-purpose processor.

This chapter has the following organization. Section 2.1 presents an general overview of fingerprint minutiae extraction. Section 2.2 discusses in a top-down fashion the inner workings and the important phases which this applications follows, in order to arrive at a list of extracted minutiae. Section 2.3 discusses the first processing phase of the application. Section 2.4 explains how the application detects the minutiae. Subsequently, Section 2.5 discusses how the fingerprint application recovers from the false minutiae that are introduced during the first two phases. Finally, Section 2.6 presents this chapter's conclusion.

2.1 Background

Fingerprint verification is one of the oldest biometric techniques that may be utilized to verify a persons identity [6]. There has been lots of research performed on this technique in the scientific community and publications on it date back to 1684 [8]. When performing fingerprint verification one tries to solve a very complex biometric problem. In essence, the problem is to compare a questioned impression of the fingertip skin to a known fingerprint impression, of which certain characteristics are pre-stored in a biometric system. The question to answer is whether this comparison indicates a resemblance. Before discussing some details of this problem, we present an informal discussion about some of the utilized definitions in this scientific field. Furthermore, these definitions are utilized in the rest of this dissertation.

A (*friction*) *ridge* is defined as a raised portion of the epidermis skin from a fingertip. This portion is made up of one or more connected ridge units. Together these units form a *ridge* [25]. The topology of the skin structure implies that there are parts with lower skin running parallel to the ridges. This phenomenon is known as the **ridge-valley duality**. The dual skin region is called a *valley*.

It is a very complex and computationally intensive task to determine a resemblance between the fingerprint in question and the pre-stored characteristics of a known fingerprint. In general, researchers have solved this problem by applying a divide and conquer strategy that reduces the original verification problem into a subset of smaller less complex problems. Subsequently, each smaller problem is solved. When this is not possible, this problem is further divided until one arrives at a solvable situation.

The MINDTCT application uses local ridge features known as the *minutiae* characteristics [7]. Among the many different minutiae characteristics found in the literature the MINDTC application utilizes only two [3]. These minutiae characteristics are:

1. The ridge *endings*
2. The ridge *bifurcations*

We illustrate these characteristics in Figure 2.1. The left picture shows two bifurcations and the right picture shows the ridge endings. The point at which a ridge diverges into two is called the bifurcation and the point where the reverse happens is called the ridge ending [6].

For each detected minutiae, the application stores three characteristics in order to position and quantify the reliability of a minutu.

1. **Position**
2. **Orientation**
3. **Quality**

The position of a minutu is stored as a pair of (x,y) coordinates. These are located where the end-points or the start-points from a detected ridge are found in the fingerprint

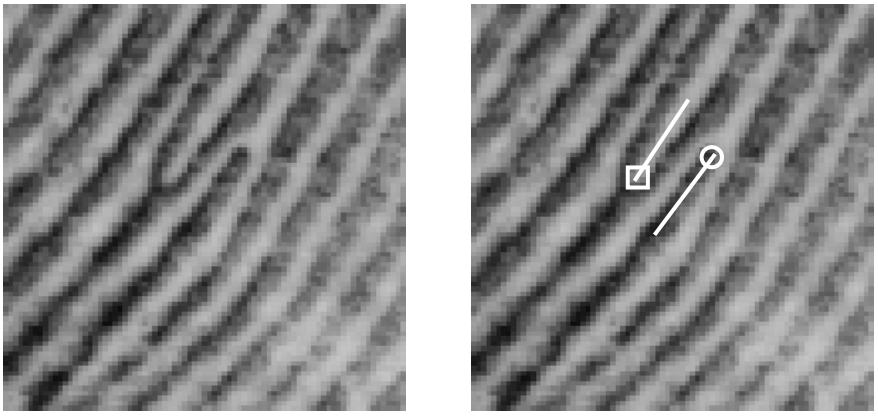

Figure 2.1: Bifurcations and ridge endings [7]

image. The orientation is defined as the angle that the minutu tangent formes with the horizontal axis. The second characteristics is depicted in Figure 2.2. The angle that is utilized is specified by 'A' in the NIST standard. The position is measured with the top left corner of the fingerprint image as the origin.

As some of the detected minutiae might stem from low quality regions of the fingerprint image, each region is assigned a quality factor. In the matching phase this is utilized as a selection weight for quantifying the reliability of the detected minutiae. Therefore, the 4-tuple (x,y,α,%) uniquely quantifies each minutu in the fingerprint image.

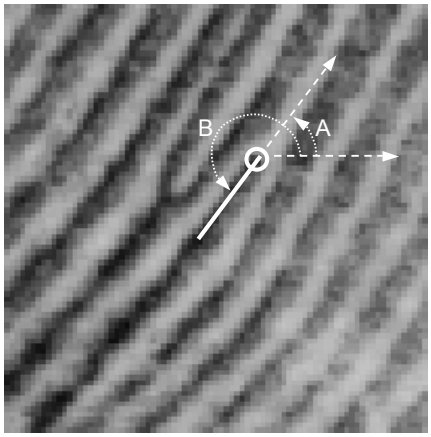

Figure 2.2: Measuring the minutu orientation [7]

2.2 Minutiae Extraction Phases

From the perspective of an application expert the minutiae extraction process may be viewed as an image pipeline. This view model helped us to cope with the applications complexity and allowed a further division of the image extraction process in three phases

[3]. With this viewpoint, we are also able to explore and understand the application in a shorter time. Later this view model served as a guide when studying the application's source code. The three phases in which the image pipeline is divided are the following:

1. **Pre-process**
2. **Minutiae Detection**
3. **Post-process**

An abstract illustration of these phases is depicted in Figure 2.3. The first phase is further sub divided into three steps. Each step performs several image processing operations in order to prepare the image for the detection phase. During this phase the actual minutiae are localized in the image based on a *greedy* pattern recognition heuristic. The last phase tries to remove as much false minutiae as possible that are introduced by the first two processing phases. It should be noted that there is a difference between what is usually the result of an image pipeline and what we see as the result of the minutiae extraction process. The difference is that in a traditional image pipeline the ultimate result is a processed image, while in the fingerprint minutiae extraction application this is a list of detected and stored minutiae.

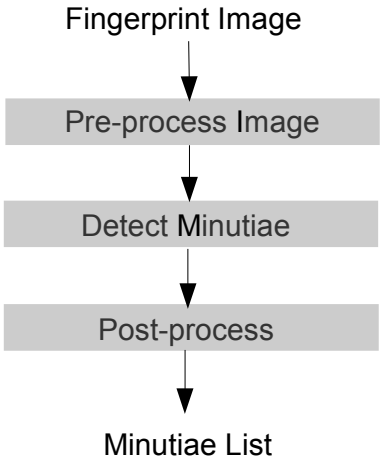

Figure 2.3: Image processing phases – Adapted from [3]

The three image processing phases are depicted in Figure 2.3. The first phase and last phase are further divided and this results in six image processing stages for the image pipeline. The first step performs several image analysis algorithms in order to prepare the image for the detection phase. The last two steps perform false minutiae removal and quality assessment. In the same way we can divide the rest of the processing phases in smaller steps. The resulting steps are depicted in Figure 2.4. In the next section we discuss each of these steps. This discussion is a concise description of the inner workings

of the fingerprint applications. For a more elaborate discussion the reader is referred to [7].

2.3 Pre-process Phase

The pre-process phase sequentially does the following image processing steps:

- Contrast enhancement
- Quality analysis
- Binarization

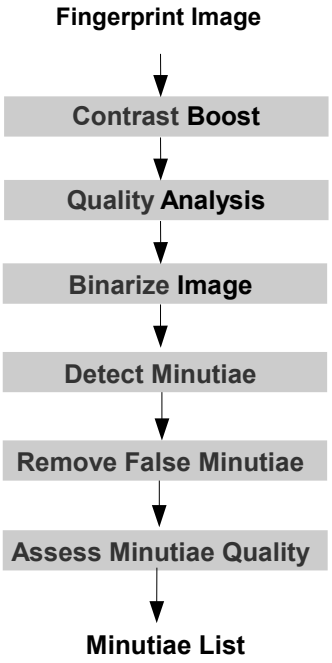

Figure 2.4: Minutiae extraction steps

The first step determines whether there are regions in the image, with a blurry ridge structure [7]. Such regions require enhancement of their contrast level. This is performed by evaluating the pixels distribution intensity [6]. The second step evaluates and decides whether some regions are degraded, too noisy, or belong to pre-defined areas of the image that will cause problems for the detection phase. After this step, the pixel within the identified blocks are assigned a low quality factor. Regions of the image with a high curvature or a low flow are typical examples. This means that in the matching phase such regions will contribute less then areas that were assigned a higher quality factor. In the third step a binarization algorithm is executed that converts the gray-scale image to black and white. The next section presents more details of each of these steps.

2.3.1 Low Contrast Analysis

In some parts of the image, it is very difficult if not impossible to detect minutiae. Such parts are regions without clearly defined ridges, such as the image background or the smudge section [3]. Problems with the background are dealt with by segmentating the fingerprint image from its background. Even so, low contrast regions are problematic for the minutiae detection phase. Therefore, the low contrast analysis creates a map that specifies the blocks that are part of these regions.

Figure 2.5: Contrast boost result [7]

The low contrast analysis phase works on image blocks of 8×8 pixels. The analysis is performed based on the fact that there is little dynamic range in the pixels intensity of such regions. This means that the intensity distribution of these blocks is very narrow [7]. The analysis starts by computing the gray-scale intensity distribution for every block. Some pre-defined part of the region with a minimum and a maximum pixel intensity are ignored since they are located in the unstable tail section of the distribution. Subsequently, the algorithm measures the **dynamic range** of the stable part of the intensity distribution of a block's pixels. When this is smaller then a pre-defined threshold, the block is marked as low contrast. Usually the tails are set to 10%. Consequently, when the dynamic range of the center 80% of a block's pixels intensity distribution is less or equal to 10 shades of gray, the block is marked as low contrast [7]. The results of this step is depicted in Figure 2.5. More details of the contrast boost algorithm are found in the block.c module of the MINDTCT's source code.

2.3.2 Direction Analysis

Before doing minutiae detection it is important to derive a directional map of the dominant ridge flow in a block. This serves as a map for recording regions in the image with well-formed ridges [3]. Moreover, it indicates the general orientation of clearly visible ridges.
The analysis works on a local basis that divides the image into a grid of blocks. As the estimated orientation must be dependable, some local information is taken into

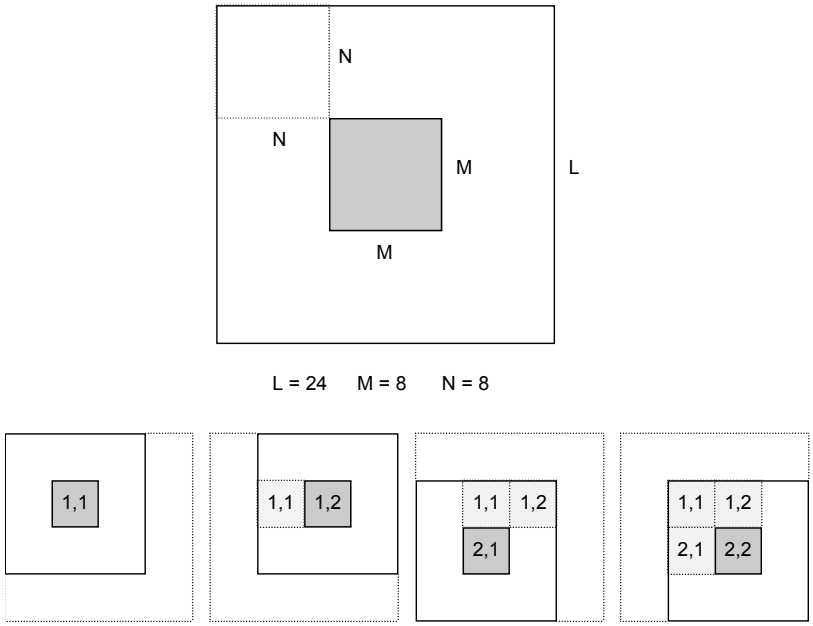

Figure 2.6: Technique of Smoothing [7]

account by making every block be part of a surrounding window. This window by default consists of 3×3 blocks such that it contans 24×24 pixels. During the analysis phase the orientation information from neighbouring blocks is utilized such that the analysis data gets shared when the results are computed. In this way, part of the image that contributes to one blocks results is used for the results of neighbouring blocks. In the fingerprint literature this technique is known as *smoothing* [6]. It allows a block to be part of a surrounding window and lets neighbouring windows overlap by one ore two blocks. The effect of this overlap reduces the discontinuities when the analysis crosses the inter-block boundaries [7]. An illustration of this technique is depicted in Figure 2.6. Here, the parameter L represents the window size, M represents the size of a block. The distance N represents the offset in pixels which the currently analyzed block has from the window's origin.

Let us now look at Figure 2.7. For each block the surrounding window is rotated 16 times. Subsequently, for every rotation the pixels along each of the 24 rotated rows are accumulated such that they form a vector with 24 row sums. As there are 16 rotations an equal amount of vectors are produced. Each of these 16 vectors, then has 24 row sum entries. Subsequently, each vector is convolved with four waveforms having double the frequency of the previous waveform, see Figure 2.8. The first waveform represents ridges and valleys with a width of 12 pixels, the second represents 6 pixels wide ridges, the third represent 3 pixel wide ridges, and the fourth waveform represent 1.5 pixel wide ridges. The four resonance coefficients that are produced from convolving each of the 16 orientations vectors with the four discrete waveforms are stored and later analyzed. The dominant ridge flow direction for a block is determined by the orientation with largest resonance waveform coefficient as each of the 4 coefficients quantifies how well the vector

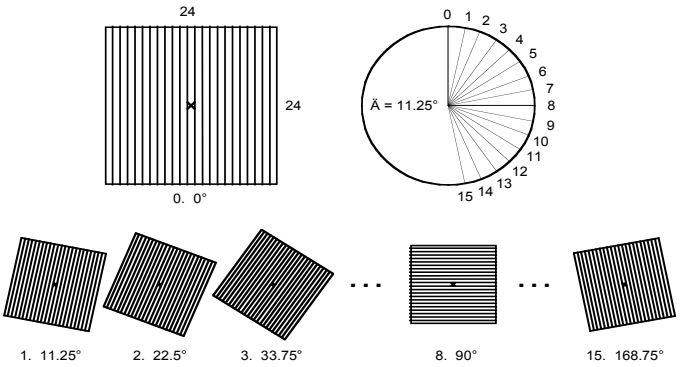

Figure 2.7: Rotating the surrounding window [7]

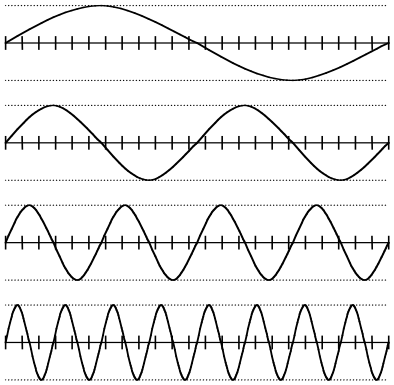

Figure 2.8: Four harmonics [7]

matches one of the waveforms [7]. More details of this analysis are found in MINDTCT source code of the file dft.c. The results of the directional map is depicted in Figure 2.10.

2.3.3 High Curvature Analysis

After the ridges flow direction in a block has been estimated, it is improved in order to increase its reliability. Part of the direction information comes from regions with unreliable or insufficient ridge flow. This is true for core and delta regions which inhibit high curvature in their ridge flow [7]. Therefore, the applications utilizes a map that records such regions using two quantities:

- **Vorticity**: the cumulative change in the direction of the ridge flow between an analyzed block and its neighbour blocks
- **Curvature**: the largest change in the direction of the ridge flow between an analyzed block and that of its neighbour blocks

Figure 2.9: Curvature map result with core and delta regions [7]

When minutiae are detected in blocks with these measured quantities, the detection assigns them a lower quality factor. During the post-process phase these minutiae participate less in the matching process. The resulting map is depicted in Figure 2.9.

Figure 2.10: Direction map result [7]

2.3.4 Direction Map Post-processing

This phase improves the quality of the directional map analysis by removing *voids* [3]. The techniques applied to the image in this step are known as morphological digital image processing. Two basic types are *erosion* and *dilation*. Dilation causes the enlargement of a mapped regions of blocks in the directional map. Erosion works in reverse since it decreases regions of the recorded blocks in the directional map [26].

2.3.5 Binarization

The minutiae detection heuristic only works on black and white images and consequently it requires conversion of the gray-scale image to a binary image. By utilizing the created directional map the application is able to do so. Each pixel gets a value based on the ridge flow of its block [7]. If this flow could not have been assessed, then it is set to white. Otherwise all the pixel within a rotated grid surrounding the target pixel are analyzed. The grid is rotated around the pixel that should be binarized. Due to the rotation, the rows of the window become aligned with the local direction of the ridge. This is depicted in Figure 2.11. The pixel values are summed along the row to form a vector with 24 row sums as entries. The center row sum is multiplied by the number of row sums, 24 in this case. Subsequently, the value produced from this multiplication is compared with the accumulated pixel intensities off all the rowsums that were added together, as to form an intensity threshold. When the value of the scaled center row sum is less than this threshold, the center pixel is set to black, otherwise it is set to white.

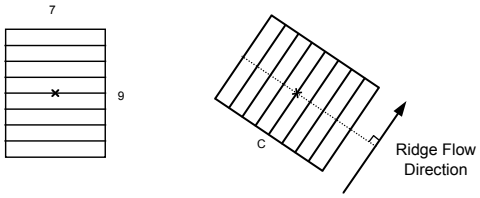

Figure 2.11: Rotated grid for binarization

2.4 Minutiae Detection Phase

During this phase after the image was processed in the previous steps, it becomes suitable for detecting the actual minutiae. The algorithm works by locating the pixel that belong to the bifurcation or ending regions [7]. During the detection phase the application executes a pattern recognition algorithm that perfroms a horizontal and vertical scan of the image. During this scan pairs of pixels are compared with 10 pre-stored patterns. These patterns are depicted in Figure 2.12. Finding the end of a ridge requires matching the current pixel pair with two patterns, the rest of the other eight patterns are used to match bifurcations. A second characteristic is that of *disappearing* or *appearing*. This is utilized to specify the direction that a ridge follows. The horizonal scan detects all vertical pixels and the vertical scan detects all horizontal pixels. In this figure, the star denotes the possibility of repetition of the middle pair of pixels.

2.5 Post-process Phase

This phase essentially removes the false minutiae that were introduced by the previous phases. Furthermore, it determines the final quality of the detected minutiae. A concise description of the major steps follow.

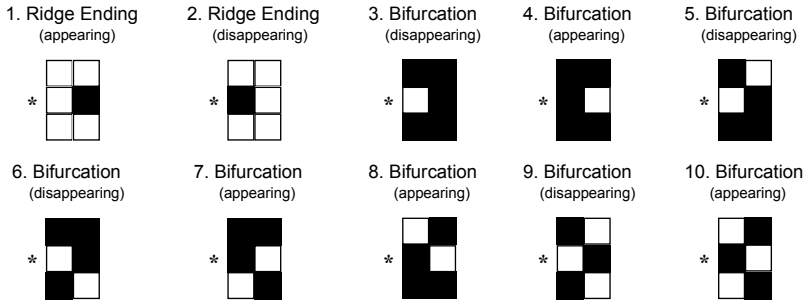

Figure 2.12: Utilized detection patterns

2.5.1 False Minutiae Removal

The minutiae detection algorithm may produce false minutiae. This happens because the algorithm matches minutiae based on a greedy approach which makes a local optimal choice at each stage. The chance of missing a true minutiae is small, but many false minutiae are introduced. In essence, the reason for this is that the pattern matching only utilizes 6 patterns of pixels [3]. As false minutiae degrade the accuracy of the verification system, much effort and code is spent to minimize them. For this reason they will are allowed to contribute less in the final matching phase. Lots of algorithms are implemented such that *hooks*, *holes*, *islands*, *overlaps*, *lakes* and *side minutiae*, *too wide* or to *narrow minutiae* are removed [3]. The exact details of how all these algorithms work is beyond the scope of this thesis. However, the curious reader can consult the source code or the manual that is provided by NIST [7].

2.5.2 Minutiae Quality Assessment

Although in the application lots of algorithms minimize the introduction of false minutiae, during the detection step there will still be false minutiae introduced. The application combines two measures to counteract this. The first, is obtained from the quality map. It assigns a quality factor that ranges from the highest value of 4 to the lowest value of 0. The second, utilizes the pixel intensity distribution. Within the direct neigbourhood of the minutiae, the mean and standard deviation of the distribution is utilized for computing the quality factor. The neigbourhood is set to 11 pixels such that it is large enough to contain the largest part of the average sized ridges or valleys. When the mean is close to 127 and the standard deviation is larger or equal to 64 pixels, a high quality factor gets assigned to the detected minutiae.

2.6 Application Motivation

A stripped down version from the fingerprint application from the CAS research group [1] was chosen as the application that we wanted to utilize as a test case on the VEX processor. The application was chosen as it satisfied the following criteria:

[1]The Circuits And System research group of TU Delft.

1. Fixed-point
2. Stripped
3. Optimized
4. C-program

The original version from NIST of the MINDTC extraction algorithm used floating-point arithmetic for the computations. Although, the VEX ISA does not specify these operations in its ISA, it is possible to utilize an emulation library for the emulation of floating-point arithmetic in software. This library is called SoftFloat [9]. The original version of the fingerprint application could have been utilized as the target application to investigate. However, we chose to port a fixed-point version. This choice is justified by comparing the execution time of the floating- point version with the fixed-point version. This revealed that the latter version was faster than the former version. The fixed-point version of the fingerprint application was already stripped, which meant that all unnecessary calls to printf() or to deprecated functions were removed [3]. As a result the code size of the original application was reduced by more than 73%, from 2825 Kb to 675 Kb. The code was even optimized in software by using lookup tables for the Binarization and Discrete Fourier Transform (DFT) kernels. These used a grid lookup instead of doing the block offset computation for every processed block. The lookup tables were utilized for computing the sine an cosine values that are required by the DFT computation. Most available fingerprint applications are written in Matlab and require floating-point operations to work. These operations are slow when special purpose floating-point hardware is lacking. Consequently, one has to emulate them. On the otherhand the fingerprint application was written in the C language and a fixed-point version was already available.

2.7 Conclusion

This chapter discussed the process of doing fingerprint minutiae extraction. We have seen that this is the most computationally intensive part when doing fingerprint verification. The problem to solve was defined as that of comparing a unknown impression of the fingertip skin to a known fingertip skin impression with the aim of determining a resemblance.

We have discussed the background information required for understanding how the application detects the minutiae. Although this is a complex biometric technique, we have seen that one can cope with its complexity by applying a divide and conquer approach. This approach divides the minutiae extraction process in a pre-process, a detection, and a post-process phase. As these phases are still complex, they are further sub divided such that six image processing steps result. During the first two steps the application analyses and maps the ridge topology structure. In the third step, it converts the gray-scale image to a binary image, such as to be able to detect the actual minutiae. This detection applies a pattern recognition heuristic to the pixels in the image that scans and compares pixel pairs to pre-defined pixel pairs such that it is able to detect

and store their locations. As the first two phases introduce many false minutiae, the fourth step corrects this by removing these minutiae. The last step asses the quality of the detected minutiae before storing the final minutiae list.

We have motivated our choice for the modified version of the fingerprint application. We saw that this version was stripped down from all non essential code. Moreover, it was adjusted to perform fixed-point computations. Furthermore, the source code was optimized by utilizing lookup tables for frequently performed trigonometric computations. As a result, the application had become faster, needed less storage and gave an output comparable with the original version from NIST.

3 System Design

Our design focus in this project was on three related aspects. The first was the fingerprint minutiae extraction application. The second was the VEX processor and the third was the processors development framework. This chapter presents the requirements and specifications that have to be fullfilled for each of these aspects, such that we can achieve our objectives. We will see that our set of specifications can be partioned in three by taking each aspect into account. Furthermore, we would like to pose two important research questions:

1. *Why is a new design of the ρ-VEX processor required for our project's purpose?*
2. *Does our application present problems for the inherited development framework?*

This chapter is structured as follows. Section 3.1 derives the overall requirements that have to be fulfilled in order to design a usable fingerprint minutiae extractor. A list of definitions that will be utilized to discuss the hardware aspects is presented in section 3.2. Section 3.3 presents the design strategy that was devised to guide us through this project. Section 3.4 gives an overview of the different processor technologies that were considered for our hardware objective. Section 3.5 goes deeper into the raised research questions. Finally, Section 3.6 concludes this chapter.

3.1 Requirements

The fingerprint application extracts the minutiae that are present in a reasonable quality fingerprint image. In the previous chapter, we explained that this is the most computational intensive part when doing fingerprint verification. From the perspective of a user, we derived the following requirements for a conceptual minutiae extractor:

- Read a fingerprint image
- Extract the minutiae
- Store the minutiae list

As was discussed in Chapter 1, there is no sensor available in our project. This means that the image must be read from an image file and stored in the system memory. The next step is that the minutiae has to be extracted from the stored fingerprint image. After this step, the fingerprint application generates the list of extracted minutiae. Finally, the minutiae list is stored in the data memory of the system, such that it becomes available for a matcher.

An important design constraint is that the extraction has to be performed in less then 10 seconds. This upper bound on the extraction time was introduced, otherwise a person would have to wait too long for verifying his fingerprint.

3.1.1 Functional Specifications

During the design phase of this project, our aim was to get the fingerprint image into the data memory of the processor. Our target processor had a Universal Asynchronous Receiver and Transmitter (UART) interface. Therefore, the idea was to utilize this interface for storing the image in the data memory of the processor [4]. Although during this phase the specifics of the UART module were not fully investigated, we hoped that this idea was not a premature one. However, exploring the inherited hardware from the ρ-VEX processor in implementation phase of the VEX processor, proved later on that we were too optimistic.

In the next list, we present the set of functional specifications that were derived for the design phase.

1. Initialize the data memory with a WSQ image of the fingerprint.
2. Decode the fingerprint image in software utilizing a WSQ-decoder.
3. Execute the minutiae extraction software on the VEX processor.
4. Write the data memory with a list of the extracted minutiae.

The third specification requires us to investigate whether certain bottlenecks of the applications have to be executed in reconfigurable hardware. Another design option was to execute the application on the VEX processor that runs on an FPGA.

It does not make sense to speed up the decoding phase of the WSQ decoding phase. Although this phase takes about 25% of the computation time, it serves to mimic a real

sensor in the capturing phase of the minutiae extractor [3]. When a sensor module is introduced this phase is no longer required, because the raw pixel values are available for the minutiae extractor. In light of this discussion, it should be noted that the time it will take a sensor module to read the fingerprints image and generate the raw pixels, is far less then what is needed to decode the fingerprint image file in software [3].

3.1.2 Design Flow Requirements

We have analyzed the functional specifications and requirements for the minutiae extractor. This analysis revealed that in order to make the best of our design effort and implementation time, we have to focuses on the three objectives in separate phases. This strategy was taken since the available implementation time and the design effort was limited in a one man thesis project. Therefore, our design focuses would shift sequentially on the following three aspects:

1. Application
2. Processor
3. Toolchain

In the same way, the design phase of the project was divided in three phases. Each aspect requires work that will produce a predefined result for the next phase. The result of the first phase is an adjusted C program of the fingerprint application that can be ported to the VEX processor simulator. This phase was needed as it justifies whether the target processor is suitable for executing the fingerprint application.

The second phase has to result in a processor with a good enough performance to satisfy the requirements of the fingerprint application. This phase starts with a carefull and thorough analysis of the ρ-VEX processor. This processor was designed in the previous project [4]. This phase provides us with answers to what would have to be improved in the target processor such that it becomes suitable or better than the ρ-VEX processor. A pessimistic scenario is that it can become necessary to come-up with a better design when improvements will take too long or do not satisfy our performance needs.

The third phase investigates the inheritted development framework. It has to generate the required software tools for generating the instruction memory for our processor. In the ρ-VEX project the designer stated that a development framework had been designed, so we were eager to put it to the test [4].

3.2 Terminology

Throughout this thesis, we utilize the following terms to describe the hardware aspects of our project. As we want to do this in a clear way, we highlight the following terms and their definitions in the context of the performed research.

- **Microarchitecture** of a computer system is the way a given instruction set architecture (ISA) is implemented on a processor.

- **Organization** defines the dynamic interplay and management of the various components that make up the microarchitecture of a computer.
- **Implementation** is defined as the logical organization of the data-flow and the controls of a computer system.
- **Realization** is defined as the physical structure that embodies the processor's implementation.

These terms and their definitions were taken from [10]. The scope during the hardware design was on all of the above aspects. We note that the ISA is fixed, because we have utilized the inherited ISA from the ρ-VEX processor [9]. We reasoned that one can have different processor designs with different implementations, but that each implementation shares the same ISA. More specifically, the hardware objective is to design a processor with a different microarchitecture than the ρ-VEX processor. Furthermore, the realization technology could be changed as the VEX processor can be realized in a different FPGA technology then its predecessor.

3.3 Strategy

In this section, we reflect on the design strategy that was utilized. The main purpose that the ρ-VEX processor had been designed for was to utilize it as a co-processor within the MOLEN framework [4]. For our project's aim, we had to come up with more of a stand-alone processor. By considering the original goal and the results of the ρ-VEX project, we devised the following design strategy. We would analyse the design and the implementation of the ρ-VEX processor in two phases:

1. A top-down architectural study
2. A bottum-up implementation analysis

The architectural study will enable us to classify the microarchitecture of the ρ-VEX processor. The second approach gives us the ability to identify what was performed at the implementation level, and why it was performed in a particular way. This approach provided us with enough information about the previous implementation, such that we could learn from it. In part, this gives us the ability to have at our disposal knowledge that can be utilized in order to come up with alternative ways for implementing a better design.

3.4 Competing Technologies

We chose to utilize the VLIW design philosophy among the different processor technologies that are available for our target processor. Our motivation is based on the arguments that we discuss through the help of some qualitative design metrics that compare the different processor technologies. The result of this comparison is summarized in Table 3.1. The first column describes the different available processor technologies that

Technology	Performance/Cost	Time to Market	Code Flexibility
ASIC	very high	very long	impossible
DSP/ASIP	high	long	long
Custom VLIW	high	short	short
RISC	low-medium	very short	very short

Table 3.1: High Performance Computing Technologies for ES – Adapted from [12]

can be utilized for our hardware objective. The second column qualitatively indicates how much performance each processors technology can attain for a target application domain. The third column qualitatively indicates how much design time is required to arrive at a working prototype. The fourth column indicates how much time it takes to optimize the processor in order to achieve maximum performance. The last column quantifies how easy it is to change the flexibility of the processor in software.

We determined that our target processor can posses all these characteristics, as it was to be situated on the third row. For example, there is the ability to define custom operations in assembly by utilizing assembly intrinsics. Moreover, in hardware one can add custom operations when it is advantageous to execute these operations during run-time. One reason to do so is when a sequence or a combination of certain operations are frequently executed within a computational kernels of the target application. It then makes sense to combine these operations in a newly defined operation. Within the ρ-VEX processor such operations are known as ρ-OPS [4]. Thus, we concluded that the VLIW processor technology enables our processor to deliver high performance, ease of use and flexibility when it becomes the processing engine for executing the fingerprint application.

3.5 Research Questions

The first implementation of the VEX ISA called the ρ-VEX processor was implemented on sound architectural principles. It was realized in a Xilinx Virtex II-Pro FPGA, and had given some impressive results for the Fibonacci benchmark. However, there was no documentation nor test results to show that the processor was verified in hardware with test files other than the Fibonacci benchmark. To do so, would imply that the development frame work would have to be tested by starting from an application programmed in C, compiling it with the VEX compiler. This produces an assembly file which then must be passed through the assembler in order to create the required instruction ROM in VHDL. Subsequently, a good verification methodology also requires to run the generated instruction ROM on the processor in the target FPGA. However, this had not been done in the previous project [4]. The original development framework consisted of the VEX compiler and the assembler/rom-generator. We have investigated the status of the ρ-VEX processor and its accompanied development framework, by following the just mentioned verification chain.

First, we have concluded that both the processor and the designed rom-generator

were far from complete. More specifically, we have assessed that the ρ-VEX would be effective as a MOLEN co-processor but that it still requires design changes and lots of design fixes in its implementation.

Second, the development frame work was not recognizing a large set of the VEX assembly semantics, (*pseudo-*)operations and the assembler directives that are emitted by the VEX compiler when it processes most C programs that are encountered in the wild.

Now we are able to answer the two research questions that were posted at the beginning of this chapter. We repeat them for convenience:

1. Why was a new design of the ρ-VEX processor required for our project's purpose?

2. What problems does our application present for the inherited development framework?

A new design was required as we have found that in order to reach our projects goal, we needed a stand-alone processor with the ability to deliver more performance than can be delivered by the ρ-VEX co-processor implementation. This may be understood when one considers that such an implementation extracts lots of *spatial* Instruction Level Parallelism (ILP) for a 4-issue processor, but does not extract *temporal* ILP from the application. Yet, an efficiently pipelined VLIW processor extracts both spatial- and **temporal** ILP from the application. Of course there should be enough potential ILP available of both kinds in the application. However, this can be assessed by first running the application on the VEX simulator. The target application has put the inherited development framework to the test and pointed out gaps that had to be designed and implemented if we wanted to automate and mature the toolchain. Put more clearly, from the inherited framework our processor requires maturity, design correctness and an extended back-end. For these reasons the assembler had to be re-designed and extended. In Chapter 7, we present the design and implementation details which originate from issues having to do with the toolchain.

3.6 Conclusion

From a functional perspective, we derived the requirements and the specification that the minutiae extraction system has to satisfy. This perspective considers the design mainly from a users point of view. However, for a designer their are other objectives to achieve. We have reasoned that in order to get the most from our limitted design efforts, we had to sequentially shift our design focus on the following objectives:

- Application
- Processor
- Toolchain

The first objective was to get the application ready such that it could be ported for execution on the VEX simulator. Once this was performed, we could run, analyze and

measure its performance on the cycle accurate VEX simulator. This simulator is provided together with the VEX compiler by Hewlett-Packard (HP), and enables a designer to specify the parameters for a particular processor instance. The second objective was to design the VEX processor according to the parameters that were specified in the simulator such that it yields the same performance. The third objective was to re-design and improve the inherited toolchain such that it could be utilized for porting the target application and to generate the instruction Read Only Memory (ROM) for its bottleneck.

We have presented the design strategy that we will follow in order to implement the VEX processor. An overview of the different available processor technologies for a designer were presented. Our conclusion was that a custom VLIW processor satisfied all the important technology characteristics as performance/cost, time-to-market and flexibility [12]. The definition of the terms that will be utilized to describe and discuss the hardware aspects of the VEX processors were presented.

Finally, we concluded that a new processors design was required, because we determined that target application required more of a stand-alone processor, which could deliver more performance than the ρ-VEX co-processor. We mentioned that the toolchain requires maturity, design correctness and an extention, in order to automate the generation of the instruction ROM of the fingerprint application for the target processor.

4 Application Development

The hardware and the software of the minutiae extraction system we aim to design are hard to separate. However, focusing on each separately increases our understanding of the system's architecture. In this chapter we discuss the software architecture and its design. Their are two objectives that we focus on in this chapter. The first objective is on modifications that enable the fingerprint minutiae extraction software to be ported to the VEX simulator. The second objective is to execute the fingerprint application on the VEX simulator. The simulator is utilized to analyze how the application will run on a processor with the assumed hardware resources.

This chapter is organized as follows. Section 4.1 discusses the structure of the fingerprint minutiae extraction application. Section 4.2 deals with problems that were encountered when we tried to port the application as it was. Section 4.3 deals with the software module that was designed and implemented in order to solve some of these problems. Section 4.4 discusses the process of porting the application to the VEX simulator. Subsequently, Section 4.5 describes the results that were obtained from running the fingerprint application on the VEX simulator. Section 4.6 discusses how we have analyzed these results. Finally, Section 4.7 concludes how we have used these results as a guide for the hardware design choices that laid a head.

4.1 Application Structure

In this section, we investigate the directory structure that the application has because this had a big impact on the way that the application was ported to the VEX simulator. The directory structure is something different than the software architecture of an application. The fingerprint application has a tree structure. A simplified illustration is depicted in Figure 4.1. This figure only partially visualizes the directory structure of the source code from the application. The complete directory structure of the application is a directory tree that contains many sub-directories [3]. For example nodes like the src directory contain many sub-directories which accompany the source files belonging to the header files, as is illustrated in Figure 4.1.

Porting the application meant choosing between two possibilities. First, it was possible to simplify the directory structure. This would mean changes to how header files were included. But the benefit would be that just one make-file would have been required to compile, link and run the application. Second, we could keep the directory structure intact. This would require a distributed compilation of the application with the use of multiple make files. This option was chosen. However, as the application used scripts for its compilation, this step required writing several make files. Although some basic experience with programming these files was present, a distributed approach was not straight forward to implement. This was due to the fact that compiling the application requires traversing the directory tree in a particular order that respects the dependencies between different source files and their accompanied header files.

First, this meant that the top make file in the root directory controlled compiling and linking of each source by calling the local make file in the leave directories. Second, the make files built libraries by composing them out of previously produced object files. Third, during the final compilation phase the complete executable of the application was generated by linking the referenced contents from each library. Running the application required executing this executable on the host system.

4.2 Encountered Problems

Before porting the application the following knowledge had to be gained:

- Get a basic understanding of the VEX simulator
- Differences between four version of the application

Within the received source code from the CAS research group, there were four versions present. Each version was the result of a milestone in the project of Michel van der Net [3]. Although the source code had lots of comment and the corresponding version names were descriptive, it was unclear which version to use in our project. We chose the software version that reads data using the UART from the LEON2 processor. This choice is justified by the fact that the target processor would also utilise such and interface for debugging purposes.

There were lots of problems encountered while porting the application to the simulator. To get an overview they are separated into two types:

```
mindtct_fix_soft
|-- bin
|   |-- mindtct.c
|-- headers
|   |-- bingrid.h
|   |-- dataio.h
|   |-- defs.h
|   |-- dft_lookup.h
|   |-- fet.h
|   |-- filesize.h
|   |-- fixedpointmath.h
|   |-- fixedpointmathcode.h
|   |-- grid_lookup.h
|   |-- img_io.h
|   |-- imgboost.h
|   |-- imgdecod.h
|   |-- lfs.h
|   |-- long_int_math.h
|   |-- morph.h
|   |-- stdint.h
|   |-- swap.h
|   `-- wsq.h
|-- libs
|   |-- libfet.a
|   |-- libimage.a
|   |-- libioutil.a
|   |-- libmindtct.a
|   `-- libwsq.a
|-- src
|   |-- fet
|   |-- image
|   |-- ioutil
|   |-- mindtct
|   `-- wsq
`-- test
|-- results
```

Figure 4.1: Simplified application directory structure

1. Problems due to the inclusion of header files
2. Problems effecting the programs functionality

This section deals with the second type. After the VEX compiler could be invoked to make the application, running it gave problems when trying to parse the fingerprint

image. The compiler complained that it could not find the Start Of Image marker (SOI). This marker indicates the position in the file where the raw pixels of the image begin. Debugging revealed that this had to do with the endian assumptions of the VEX simulator. The bytes in the image-file are stored according to the Big-endian byte ordering. To run the application directly on an x86 PC would require swapping them to the Little-endian ordering. The simulator translates C to VEX assembly and then compiles VEX assembly back to the host binary. This means that the swap is not necessary. Commenting the calls to function swap_short_bytes() in dataio.c, solved this problem.

The first type of problem was harder to solve and was encountered in the decoding stage. After reading the dimensions of the fingerprint image a 2D grid of pixel blocks had to be allocated. For computing the width and the height of this grid a fixed-point division was utilized. The grids width is computed by dividing the image width by the number of pixels in a block. This division casts the divider and divisor to 64-bits. Although this integer type was defined in a local stdint.h file of the application, its definition was not explicit as it was depended on the compilers version being used. Compiling the program with a recent version of GCC also gave problems because this integer was getting defined incorrectly. For the VEX compiler these problems got even worse as the VEX architecture does not support 64-bit integers. This observation was also confirmed by going through the compilers header files. The solution for GCC was to utilize the standard definition of the primitive types as is supported in one of the compiler's header file e.g. <stdint.h>. However, version 3.41 of the VEX compiler required defining a 64-bit object and additional arithmetic algorithms for emulating all the fixed-point operations involving 64-bit integers.

4.3 Software Design

The largest integral data type supported by VEX is 4 bytes. This information can be found in table A.7 of [9]. This means that since our program uses 64-bit integers, it was necessary to write a module that emulates operations on these objects. This module is called `long_int_math.c`. When writing it the challenge was to integrate it in the fingerprint application without too many changes in the original source code. Actually, what was required was that the changes had to be transparent for the application. Furthermore, the program had to be bug free, which meant that extensive testing was required. Otherwise, the recognized minutiae would change. To verify the module each function was separately tested. The data type defined for this program was a struct that contained a high and low word as attributes. There were functions written in C for the following arithmetic operations:

- Sign operations
- Absolute value
- Logical- and arithmetic shifts
- Casts and Coercions

- Basic arithmetic: +, -, *, /
- Comparisons

In hardware implementation of these operations are known, however emulating them in software can be tricky. One example is the implementation of the HI/LO multiplication algorithm. The idea of this algorithm is to perform a 32-bits multiplication by means of 16-bit multiplications. The first implemented version that resembles the 32 bits case seemed correct. However, for emulating 64-bits arithmetic, adding the intermediate values could create overflow problems when the product of the largest 32-bits multiplicand and multiplier are computed. This was resolved by using the user defined 64-bit object for all intermediate values. In this way all possible intermediate carries were accumulated correctly, see code Listng 4.1. Although being somewhat slower than the HI/LO implementation, the advantage is that it is correct for all possible operands. The function that we used for division implements *restoring* division.

Listing 4.1: Multiplication with 32-bits operands

```
int_64 umult(register unsigned long a, register unsigned long b)
{
        unsigned long a_low = a & 0xffff;
        unsigned long a_high = a >> 16;
        unsigned long b_low = b & 0xffff;
        unsigned long b_high = b >> 16;

        unsigned long p_ll = a_low * b_low;
        unsigned long p_lh = a_low * b_high;
        unsigned long p_hl = a_high * b_low;
        unsigned long p_hh = a_high * b_high;

        /* accumulate results with intermediate carries */
        int_64 res = ucast(p_ll);
        res = add(res, sll(ucast(p_lh), 16));
        res = add(res, sll(ucast(p_hl), 16));
        res = add(res, sll(ucast(p_hh), 32));
        return res;
}
```

4.3.1 Arithmetic Module

The design of the arithmetic module started by implementing and testing the arithmetic operations for unsigned numbers. Once the results were verified it was extended for the case that the operands were signed. Eventually the ultimate goal was to emulate the arithmetic that the VEX compiler could not support. After verifying the correctness of the module, it could be integrated in the application. By including the accompanied header file at the lowest level in the file fixedpointmathcode.h, the whole application got access to the newly defined data type. The assumption was that the conversion to fixed-point numbers that had been performed by Van der Net, was correct [3]. This meant that he had correctly analyzed the range of each fixed-point operation and that

they would never produce values larger than 32-bits. The reader is encouraged to read Chapter 3 of his thesis. Subsequently, the fixed-point operations had to utilize the 64-bits objects for its calculations. There were two approaches possible: one could either use the 64-bit integers explicitly or implicitly. The former meant that every fixed-point operation that used 64-bit integers had to be adjusted such that it could utilize the newly defined 64-bit types. The assumption that we made implied that the results would never exceed 32 bits, so that the high-word would only contain sign information or zeroes in the unsigned case. Moreover, this approach would require that over a hundred location would need to be changed manually, which would take lots of development time. As the original assumptions were that the application could be ported "as it was", the implicit approach was chosen. Even more important was the fact that porting was already taking longer than what was planned. The implicit approach was transparent as it required that only the operation within the defines of the application would need to make calls to the arithmetic functions of the arithmetic module. The rest of the source code would stay the same, as is the case for all fixed-point calls made in the application. Hence, the header file of the fixed-point operations included long_int_math.h, which is the header file of the arithmetic module.

We illustrate the above discussion by means of a code excerpt. The first define computes the division of two fixed-point numbers in the original fixed-point version. This representation utilizes 8 bits for the integer part and 24 bits for the fractional part. The division requires storing the dividend in a 64-bits register (cast). Then, a left shift is required by the number of fractional bits in the fixed-point representation. Finally, an integer division follows. The new version does a similar series of operations. First, it does the unsigned cast of the dividend and the divisor using the respective function. It also requires that the divisor is cast to 64-bits as the callee expects both operands to have the same size. This is no problem since it only involves sign extending the smaller divisor. This adds only sign information to the divisor, the magnitude stays the same. Next, the division operation calls the divide function. As the high word of the quotient is assumed to contain only zero bits, we cast it to a 32-bit integer. To make sure that this assumption holds, each cast function gets the added responsibility to generate an exception when this assumption is violated. Additionally, we made the trade-off to use right arithmetic shifts instead of using logical ones. When performing 32-bits fixed-point multiplication, one never needs to shift right more than 32 bits, which means that the sign bit will never have to be copied to the most significant bit of the lower word. However, having such a function gives the ability to check whether each signed cast to a 32-bit integer only throws redundant sign bits away. As correctness was important, we have utilized the right arithmetic shifts in the module.

```
Before:

#define _divufp8p24(x,y) (((uint64_t)(x) << 24) / (y))

After:

#define _divufp8p24(x,y) (cast_uint32(udivide(sll(ucast(x), 24), ucast(y))))
```

4.3.2 Complications

Although the chosen strategy to adjust the application was paying off, there were some complications. As was mentioned in chapter 3 of Van der Nets dissertation, there would be some differences between the fixed-point version and the original floating-point version [3]. All high quality minutiae should be detected. However, some low quality minutiae could be falsely detected. There was also a possibility that some of these minutiae would not be detected. When inspecting the differences, it was observed that some minutiae were indeed incorrectly detected. This was confirmed by the results of the cast functions. This situation indicated that the made assumptions were no longer holding since information was being lost in the high-word of the 64-bit objects. In other words when casting from the 64-bit integers back to the 32-bits, the high-bits sporadically contained magnitude information. To solve this problem, we concluded that somewhere in the fixed-point process or in the process of adding the arithmetic module, something had gone wrong. Thus the following steps were taken:

1. Re-check the arithmetic module's source code
2. Test arithmetic functions under new scenarios
3. Re-check every adjustments to the application
4. Run the old fixed-point version with GCC

All of the above steps required extra programming time. The first three steps did not reveal any new information, but did increase our believe that the module was bug free. The last step gave us a reference to compare the detected minutiae with. In Section 6.3.3, we see that our debugging approach presents some interesting results.

4.3.3 Software Adjustments

The adjustments to our program were mostly performed to improve its exception behaviour. In the last simulation phase the multiplication was implemented to deal with the case that the input operands can become large enough to cause an overflow. Subsequently, every cast function was extended with checks that verified whether the made assumptions were still holding i.e., we would never need a fixed-point representation that was larger than 32-bits. Moreover, these checks were to confirm the correctness of the precision analysis that was perfromed in the previous project [3]. Furthermore, there was measurement code added to check how fast the application was running in the VEX simulator.

4.4 Porting to VEX Simulator

Before porting any application it is wise that the programmer gets a basic understanding of the application he intends to port. When things go wrong, and they usually do, he can debug the application in a better way. Consequently, we utilized the following preliminaries before staring to port the application to the VEX simulator:

1. Understand the application.
2. Follow a structured approach.
3. Understand the VEX tool-chain.

During the project's definition phase, we had already performed the necessary literature research on the fingerprint minutiae application [6, 7, 8]. This gave a good basis for understanding and studying the application's source code. The first preliminary was fullfiled by exploring the applications source code during a weeks time. We utilized the approach to start simple after which we would gradually increase the design and implementation complexity. During this process we would learn about the provided tool-chain as we were using it. This meant running a basic C program on the host PC utilizing the VEX compiler and the simulator from Hewlett-Packard labaratories [15]. "The VEX compiler is and industrial descendant of the Lx compiler, which is a descendant of the *Multiflow* compiler" [9]. We have chosen this compiler as it combines the robustness of an industrial compiler with the flexibility of a reaserch platfrom [12]. In retrospect, our approach enabled us to separate the process of getting a basic understanding of how the provided tool-chain worked from the process of dealing with the real complexity of our application. Once we felt confident enough that the second and third preliminaries were satisfied, we started to port the application.

4.4.1 Bugs

There were some bugs found that were a result of porting the application. However, one was also discovered in the old fixed-point version received from CAS. A short discussion of the hardest one follows accompanied with their fixes.

While reading the input image the decoded pixels were observed to be zero. However, the image parameters where computed correctly. The problem was even harder to solve because the part where the image-file was being read was actually accessing the pre-stored pixel values from a header file. This input file kept the image pixels in a large array in the old fixed-point version. What was confusing is that the input string in the main function was suggesting that the file was being read from the directory where the test image files where stored. As a first step to solve this problem the input image was now directed such that it got accessed from the location that the input string in main was suggesting. This was in contrast to what had obviously been done to speed up the decoding phase in Michels project. Next, as we were still reading in zero pixel, we traced the problem in the source code back to the contrast-boost phase of the image. This phase still utilized the unsupported 64-bit objects to enhance the value of the pixels. This meant that every pixel was getting zero assigned as its value. When the defines got replaced as is described in Section 4.3.1, the expected pixels of the fingerprint image were finally appearing on the console.
When performing directional analysis as is described in the file dft.c, the computation of the power magnitude utilized the define for an unsigned fixed-point multiplication. In the C language each define is replaced with an inline substitution and the result receives a type that is inferred from the type of the actual arguments. In the previous version, an unsigned multiplication define was computing the magnitude of the resonance coefficient.

This computation is presented in Equation (4.1). The index w represents one of the four wave forms with a particular frequency. This represents ridges and valleys with a certain width of pixels. The index d indicates which one of the 16 directions was used when accumulating the pixels in a row of the rotated window.

$$power_{w,d} = \cos^2(x) + \sin^2(x) \tag{4.1}$$

The problem was that the unsigned multiplication were performing incorrect computations as their arguments were signed. The solution was to define a signed multiplication version called mulfp27p5 for computing the squares correctly.
In Section 4.3, we saw that when casting from the 64-bit to 32-bit integers, the high-bits sporadically contained magnitude information. For most of the defines there was a one-to-one relation between the sign of the arguments and the results. However, in the following locations there were two exceptions:

- image/imgboost.c:165: divfp8p24
- image/imgboost.c:172: mulfp8p24

The call to the macros looked like they performed signed computation, however as the input arguments were unsigned they in fact did unsigned computations. The new 64-bits version was doing signed computation, which was incorrect for the above cases. The reader will be spared from the hairy details of detecting this bug. The solution on the otherhand was easy as it involved replacing the macro with its unsigned version.

4.4.2 Testing

At this point testing the ported application has revealed that all made assumption regarding the range of the fixed-point representation are true. After we fixed the last bug, we observed that all minutiae that were being detected by the original fixed-point version were also being detected by the ported version. In addition, we were even detecting low quality minutiae that the old version was not detecting. This was also observed when converting the original floating-point version to fixed-point [3]. To compare the old and the ported fixed-point versions required running the former with GCC. This took one day of work but at this point we were exceeding our porting time budget. The ultimate test would be to get the floating-point version, strip it, and run it with GCC. A comparison would reveal if our version was indeed completely bug free. We did not perform this, as we argued that the matchers output would not significantly change because only low minutiae were introduced. The next phase in testing is to run the three versions on all 30-fingerprint test images and check all results. This will need even more implementation time.

4.5 Simulation Results

The initial goal to run the application on the VEX simulator was to get answers to two important questions:

Issue-width	**1 cluster** [ms]	**2 clusters** [ms]	**4 clusters** [ms]
1	14863.2	14863.2	14863.2
2	11395.5	11395.5	11395.5
4	9536.45	9701.55	9439.81

Table 4.1: Simulation cycles for different processor configurations

1. Will a multi-cluster design be required for executing the application?
2. What are the effects of different issue-widths on the execution time?

Answers to these questions guided us during the hardware design decisions. In the next subsections we discuss and provide these answers. The basis of our experiments of running the application on the VEX simulator were performed using the *ceteris paribus* reasoning. These experiments were done for different data-path configurations. We wrote nine machine configuration files of which the one chosen to derive the architectural parameters for the target processor is presented in Appendix A. Among others, the nine machine configuration files specifed the cluster size and the issue-width of the VEX datapath, while keeping other datapath parameters as the number and sort of functional units constant. Additionally, we experimented with the size of the register files. We changed the size of the general-purpose register file to 32 registers and the size of the branch register file was also altered to 4 branch registers. As the branch frequency in our application is high we observed that this increased the execution time for all clustered organization. This is explained by the fact that the compiler has to schedule extra operations to spill and restore branch register values. Spilling and restoring of registers causes an extra execution overhead. Furthermore, our applications contains a high ILP level of 1.60. This motivated our choice not to decrease the general-purpose register file, as this will increases the presure on the register file and their will be more code generated to spill and restore general-purpose registers.

The number of clock-cycles it takes to run the application on the VEX simulator with different configurations are presented in Table 4.1. A graphically illustration is depicted in Figure 4.2. From these results we concluded that the speed up that a multi-cluster datapath will provide the application is marginal. It looks more promising to concentrate the hardware design efforts on a single cluster. On the contrary, increasing the issue-width per cluster seems to provide marginally more speed up. Therefore, an issue-width of four for a default cluster organization with 64 general-purpose registers and 8 branch registers seemed to be more promissing.

4.6 Analysis

The fingerprint application was profiled in order to investigate and determine the computational intensive kernels. Furthermore, this analysis provides us with feedback on what impact the integration of our arithmetic module has on the desired performance. In this phase of the project the following steps were performed:

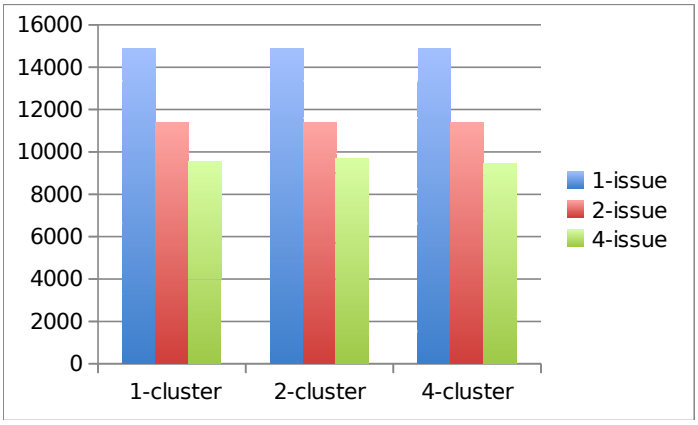

Figure 4.2: VEX simulator results

- Profile our fingerprint application
- Discuss latency of functional units
- Perform profile based optimization

4.6.1 Profiling

The Vex tool-chain supports profiling applications by utilizing the *gprof* utility [15]. Additionally, users can graphically visualize their profile results by pulling emitted graph information through the Data Flow Graph (DFG) with the *rgg* utility [9]. We investigated the graphical profile results of the fingerprint application and observed that the directional analysis phase, takes up 34.5% of the execution cycles. On the otherhand, we observed that the binarization kernel, which was one of the bottlenecks for the Leon2 microcontroller, is no problem for the target VEX processor. This kernel requires only 0.53% of the total number of execution cycles.

4.6.2 Software Optimization

We would like to discuss one of the advanced scheduling techniques from the VEX compiler. Trace scheduling comes into effect when one optimizes sections of the program with the -O3 compiler flag [14]. We experimented with this option and observed that it has a big impact on performance. The execution time from our application reduced form 18.530 to 9.499 ms. Furthermore, this type of scheduling can be performed in conjunction with profile base optimization. With this conjunction the most use is made out of the scheduling abilities of the VEX compiler. This conjunction is performed in three steps:

1. Compilation is done with *-prob-gen* flag.
2. Branch statistics are collected.
3. Re-execute application after recompilation.

First, we enabled the compiler to instrument the application's code in order that it can count branches statistics. This information gets stored in a module's directory. Second, the program was executed on a significant data set in order to collect the branch statistics. Third, a re-compilation is needed with the *-prob-use* flag that tels the VEX compiler to utilize previously collected statistics for the selection of traces. When the chosen trace coincides with the fall-through path, the program structure stays intact. As a side effect, the compiler may re-transforms the code structure such that it can generate compensation code for the less favourable off-trace paths.

4.6.3 Latencies of Functional Units

We utilized two latency specifications for the functional units in the machine configuration files. First, we extended specifications of the default VEX configurations to multiple VEX clusters. This gave us the ability to estimate the execution time when we will target a design that utilizes multiple VEX clusters. Second, we specified the configuration files such that they indicate the target machine organization to the VEX simulator. The default VEX latencies that are specified for the LOAD/STORE unit are longer than what was implemented in our design. Consequently, these latencies were adjusted accordingly. The details of these files are given in the Appendix A.

4.7 Conclusion

We investigated the structure of the application and the consequences it had for compilation and execution on the VEX simulator. Furthermore, the problems that we encountered while porting the application and their solutions were discussed. We discussed that in order to be able to port the application, we had to develop an arithmetic module and incorporate it transparently into the application. Adjusting the application such that it could deal with arithmetic exceptions and could provide and initial execution time, proved also necessary.

We reasoned that based on the simulation results a multi-cluster approach does not seem to be the best design option. A single cluster with an 4-issue wide instruction looks more promising. After this step CCU can be used as the means of getting more speed up. This could be done by using these units to implement the remaining bottleneck. Another option is to use customized operations. These operations are called *assembly intrinsics* in the VEX ISA. It is worth mentioning that this assumed that the target application could be ported on a ready to use processor. Although the ρ -VEX could run at 89.3 MHz on a Virtex-II Pro FPGA, we still had to investigate whether it was able to function as a stand-alone processor. In the next chapter, we see that for our project's purpose a new processor design was required.

5 Hardware Design

This chapter focuses on the hardware design of the VEX processor that was simulated on the simulator. We need to understand the VEX Instruction Set Architecture (ISA) such that we are able to comprehend the requirements that each VEX operation implies for the target hardware. We will see that in order to deal with most of the architectural problems that Thijs van As had faced in his ρ-VEX design, it is required to come up with a different instruction encoding scheme [4]. Furthermore, if we want to design a better micro-architecture, we need to identify and understand the micro-architectural decisions that had limited the performance of the previous ρ-VEX implementation.

The structure of this chapter is the following. Section 5.1 discusses the details of the VEX ISA. Section 5.2 looks at the organization of the VEX processor such as to create an architectural basis for the processor's micro-architecture. In Section 5.3, we present the instruction format that we devised in order to encode all of the native VEX operations. Section 5.4 looks at the micro-architecture of the VEX processor and focuses on all of the related pipeline details. Moreover, this section provides an explanation of the inner workings of the pipeline stages of the processor. Section 5.5 discusses three architectural improvements that will make the designed micro-architecture excel in its performance. Finally, Section 5.6 presents a summary of all of the discussed design aspects.

5.1 The VEX ISA

The VEX ISA defines a Load/Store VLIW architecture in terms of the *visible* micro-architecture, which concerns the syntax, semantics and constraints of *operations* [9]. An encoded operation is called a *syllable*, whereas a set of syllable issued in a single cycle together, is defined as an *instruction* by the ISA. This means that operations in VEX terms are equivalent to RISC instructions. Each operations is specified to have a size of 32-bits. The VEX compiler has the responsibility to find, select and schedule independent operations for the target VLIW processor. The compiler contains additional features that increase its schedulling flexibility. A complete exposure off all architectural latencies and the resource constraints of the proccessor are made available by specifying a machine configuration file [9]. This file provides the means for specifying a blue-print of the architecture by instantiating the parameters of the desired processor. As we saw in Chapter 4, this file stores the parameters that were set when running the target application on the VEX simulator. There are two set of constraints that have to be fullfilled when specifying the desired architecture [9]. These set of constraints are:

1. A set of *common rules* for all VEX implementations.
2. A set of *specific rules* for a particular VEX instance.

The first set specifies the base ISA, the register connectivity, the coherency of memory and the architectural state of the processor. The second set specifies the issue-width, the latency and behaviour of functional Units, the size of the register file and the ISA of the processor instance. Both rules constraint the processor's design space such that a processor architect is able to explore and determine the specifics of the desired architecture. Furthermore, with the aid of the VEX simulator he is able to assess whether the desired architecture will match the target application domain.

In the coming sections, we discuss several specifics of the first set of rules. The second set of rules is discussed in Section 5.2, where we present the details of the designed VEX micro-architecture.

5.1.1 Execution Model

The Vex ISA specifies an *in-order* multiple-issue VLIW micro-architecture. In particular, it defines that among the operations of an instruction there may not be any sequential constraints present. For our in-order machine, this means that all syllable within an instruction must start together and should commit their results together. During the commit phase, registers are updated. Moreover, the ISA specifies that a *precise exception* model should be supported by the micro-architecture. This implies that the latest exception stage is the second execution stage. This stage precedes the commit stage and is known as the WriteBack stage in our design. Operations may have different latencies. This means that the compiler must schedule instructions while it obeys the latency constraints among any pair of operations. This architectural model is known as *non-uniform assigned latency* or a *NUAL* model. When an in flight operations requires longer than the assumed architectural latency to execute, the hardware must stall execution until the assumed latency holds again. Our VEX architecture specifies a *less-than-or-equal*

machine (*LEQ*) in particular we require that store operation commit their results one cycle earlier than other operations. More specifically, they write their result to Data memory in the second execution stage. Additionally, the VEX encoding scheme specifies that the ISA must provide the following two bits:

1. *instruction-stop bit*
2. *cluster-start bit*

These bits have to be utilized by the sequencing logic to fold operations of neighboring instructions into their empty syllable slots. This technique is known as *nops folding*. The first bit then indicates the end of an instruction. It is utilized by the sequencing logic to recognize the execution boundaries of an instruction. The second bit indicates the beginning of the instruction section from a **new** cluster. It can be utilized by the *dispersal* logic to route the right part of the instruction stream to the corresponding fetch logic of the target cluster. In this way a clustered architecture directly supports scalability of the processor's micro-architecture, especially its register files.

5.1.2 Architectural State

The architectural state is defined as the set of all visible micro-architectural storage elements [9]. In VEX this set consists of the General-Purpose registers, the branch registers, the control registers and other addressable storage elements. Because the program counter is not addressable, it does not belong to this set. All operations work on 32-bit containers. In principle 64-bit values are not supported. Operations that work on branch register have to provide the required data conversions when data is moved between branch registers and General-Purpose registers. Furthermore, the VEX ISA specifies support for three type of immediates: 24-bits branch offsets, 9-bits short immediate and 32 bits long immediate. Except for the latter, all three immediate types fit within a single syllable. Long immediates have additional bits that are carried by an adjacent syllable in the same instruction and cluster, provided that every syllable maps to a Load/Store Unit. As constant values are very common in operations, this requirement of the ISA means that all immediates can be decoded in the same instruction, during the same clock-cycle, such that they do not penalize the depending operations. Implementing this in the micro-architecture makes the common case fast.

5.1.3 Multi-Cycle Implementation

After the first analysis phase we concluded that the ρ-VEX processor had a four-stage, 4-issue Harvard architecture. The processors is comprised of a fetch, decode, execute and writeback stage. Moreover, the ρ-VEX micro-architecture was loosely based on the standard VEX configuration cluster. This processor has four Arithmetic Logic Units (ALUs), two Multiplier Units (MULTs), one Memory Unit (MEM) and a Control Unit(CTRL) as functional units [4]. The processor's state elements consist of a General-Purpose (GP) Register File (RF) with 64 32-bit register and a Branch (BR) Register File with 8 1-bit registers. The second analysis phase allowed us to classify that the ρ-VEX processor was implemented as a **multi-cycle** machine [21]. This means that operations within an

instructions are executed in multiple clock-cycles. Such an implementation enables the processor to share functional units within the execution of a single instruction during different clock cycles. The main advantages of this implementation choice is that operations may require different numbers of execution cycles, which allows them to share functional Units during execution time.

A centralized Control Unit for controlling the processor's stages was absent. Instead, the designer had chosen for a distributed Finite State Machine (FSM) to implement the control of the processor. This allowed him to deal with the reduced complexity of having to implement separate FSM's instead of implementing a single complex FSM for controlling the processor. Thus, a divide and conquer approach was utilized to implement the Control Unit. However, from an micro-architectural point of view a direct consequence of this approach is that controlling the processor requires additional *inter-stage synchronization* [4]. This was implemented by using handshakes of control signals between stages that allowed information to flow in two directions. Although for Thijs' design a multi-cycle implementation did the job, the downside is that the average clock-cycles of operations (CPI) increases as the processor spends additional cycles for synchronization of neighboring stages.

5.2 Processor Organization

For our project's purpose we have designed a pipelined processor based on the VEX ISA of a default cluster. This means that our architecture has four ALUs, two 16×32 MULT's, a Load/Store Unit (MEM) and a Branch Unit (CTRL). The architectural state elements are comprised of a General-Purpose Register File with sixty-four 32-bits registers and a Branch (BR) Register File with eight 1-bit registers. Like any RISC ISA our VLIW ISA was designed for a pipelined implementation. Hence, our approach is different from the previous multi-cycle design, because this micro-architecture has a pipelined organization. In order to achieve this goal, we have restricted the VEX processors design to utilize only uni-directional inter-stage signaling. Furthermore, our micro-architecture requires a design with pipeline register between neighboring stages. For the VEX organization the idea was to get rid of bi-direction inter-stage signaling. This would enable us to make control and data information available where they would be required in the processor's pipeline. When the design was implemented efficiently, we would not loose additional clock-cycles due to the inter-stage signaling. In turn this implies a reduction of the average clock-cycles per instruction (CPI) compared to the multi-cycle processor. Of course the ultimate goal is to reduce the execution time when a program runs on the VEX processor. Thus, its instructive to quantify the execution time of a processor [22]. This time may be formulated as in Equation (5.1).

$$CPU_{time} = Instruction_{count} \times Clock_{cycle-time} \times CPI \tag{5.1}$$

Form a designer's perspective the assumption that both processors run the same program, will cause the compiler to emit the same number of instruction for both implementations. Hence, the instruction count becomes equal. Let us now assume for simplicity reasons that we utilize the same FPGA technology, then we can argue that the clock-cycle time of

both designs is the same. Subsequently, the purpose from an architectural perspective is to reduce the average clock-cycles per instruction of the VEX processor. This is precisely what a pipelined design can accomplish. Moreover, it is possible to reduce the clock-cycle time by proposing a better processor organization. Let us look at how the basic technologies are involved when attempting to change the performance characteristics.

- Clock-cycle time: hardware technology and organization.
- CPI: organization and instruction set architecture.
- Instruction count: instruction set architecture and compiler technology.

We see that changing a separate characteristic is challenging, since they are all interdependent. However, when the application, the ISA, and the compiler are fixed, the machine's organization forms a good starting point to improve the execution time. In the best case, we can improve both the average cycles per instruction and the clock-cycle time with a better organization. An extra requirement was to design the micro-architecture in such away that it can anticipate extentions and future improvements. This happens when for example a designer wants to consider a multi-cluster implementation.

Our processor's design is based on a Harvard Load/Store VLIW architecture. By separating instruction and data memory, the micro-architecture has the advantages of being able to support the processor with a higher memory bandwidth compared to a Von Neumann architecture. Second, at a any specific time instant, data and instruction do not have to compete for a single memory port [13]. The processor's micro-architecture is based on an in-order four-issue 5-stage pipeline. All operations in an instruction are fetched in the same clock-cycle and commit their results together in the WriteBack stage. The VEX pipeline consits of the following stages: ***Instruction Fetch***, ***Instruction Decode***, ***Execute*** 0, ***Execute*** 1, ***WriteBack***. An high level illustration of the designed micro-architecture for the VEX processor is depicted in Figure 5.1.

The Fetch stage reads an instruction from instruction memory and sends this to the Decode stage. Additionally, it works in conjunction with the Branch Unit in the Decode stage to select the next instruction address for the program counter. Usually this is the address of the next instruction, but when a branch syllable is decoded it represents a potential branch target. The last responsibility of the Fetch stage is to signal the pipeline that the instruction stream has become exhausted, such that the processor stops processing instructions.

The Decode stage receives from the Fetch stage a bundle and splits this into four syllables. These syllables are decoded in parallel such that all operation fields accesses the Register File and can feed the 4 pipe-lanes concurrently. Subsequently, decoding the four syllables and accessing the register file is performed in parallel, as we are able to take advantage of our regular instruction layout. The Decode stage generates the correct value for syllables that contain immediates. Apart from immediate generation, this stage stores the Branch Unit as part of a *two-step branch architecture* that exposes the branch latencies to the compiler. Upon decoding a conditional branch operation, the branch flag that was stored in a branch register is concurrently read and sent to the Branch Unit. Later, we will see the advantages of this branch micro-architecture for our

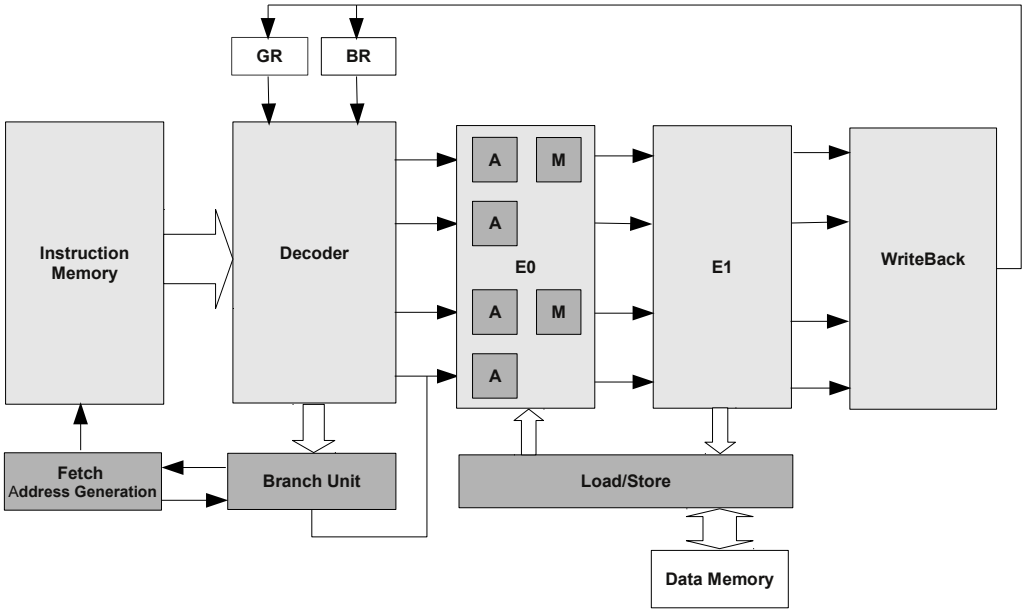

Figure 5.1: VEX Processor Organization

VEX processor. During the end of the second clock-cycle the values that were read from the General-Purpose Register File are sent as operands to the first execution stage. The datapath of the VEX processor has two execution stages. This organization is motivated by three micro-architectural aspects:

1. Multiplications operations
2. Memory Load/Store operations
3. Exception architectural support

First, the ISA specifies a latency of two clock-cycles for multiplication operations. This is because a 32×32 bit multiplications is executed by scheduling it in two phases. During the first phase, two 16×32 multiplications have to be performed in parallel for generating the partial products. In the second phase, these products have to be added in parallel to form the 32-bits product. For the design of our VEX processor this means that multiplications has to be performed during the first execution stage after which the addition of partial products follows in the second execution stage.

Second, because the VEX ISA specifies that load and store operations can contain source values with a smaller granularity than the natural 32-bit word size. For example it is possible to store a byte in Data memory. This requires pre-loading the memory word containing the target byte, after which the source byte is written to the target position

within the pre-loaded word. Clearly, the pre-loading the memory word that will receive the target byte may be performed during the first execution stage.

Third, during the second execution stage the updated word is stored at the address in Data memory. On the other hand, choosing two execution stages implies complexer implementation issues than when a single execution stage is chosen. However, as our aim is to devise a better processor organization than that of the ρ-VEX processor, we argue that this choice is justified. The implementation issues that result from it are discussed in Chapter 4.

Fourth, the WriteBack stage is responsible for committing all the received values from the second Execution stage to their destinations registers in the General-Purpose Register File. The results are either from a functional unit or from a value which was loaded from Data memory. This happens when a load word operation is being processed in lane 2 of the pipeline. It should be noted that the Branch Unit is regarded as a functional unit, since it computes the new stack pointer value or the return address. One of these two values is carried through the execution stages and is committed in the WriteBack stage. The pipelined micro-architecture requires that the WriteBack stage performs Register File forwarding in collaboration with the Decode stage. The micro-architectural details of this stage are presented in Section 5.4.2.

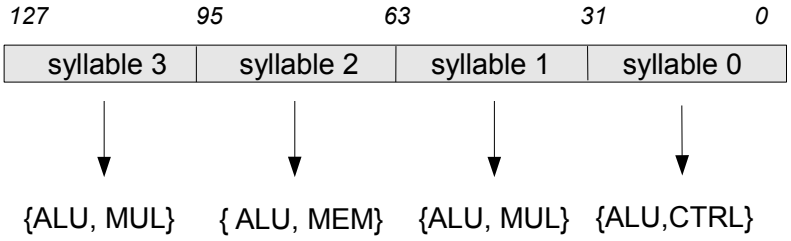

Figure 5.2: Instruction layout – Adapted from [4]

5.3 Instruction Formats

We decided to regulate the instruction layout that was utilized by the ρ-VEX design, in order to utilize it in the micro-architecture of the VEX processor. This is performed such that the instruction layout becomes suited for a pipelined micro-architecture. At first, we started with the instruction layout from the ρ-VEX design. However, during the implementation phase we concluded that in order to solve some of the architectural problems that the designer had ended up with, required changing the instruction layout and all of its specified opcode values. Together, these changes allow us to encode the native VEX operations. In the implementation phase we came across several challenging issues and it appeared that sticking to the old instruction layout and its encoding, would prevent us from solving these. All four issue-slots still issue an ALU operations, but the MEM Unit has been swapped with the second MUL Unit. We have performed this change in order to solve implementation problems for generating long-immediates operands. We have utilized the instruction layout that is depicted in Figure 5.2. The

syllable ordering within an instructions is Little endian, but this should not be confused with the Big endian byte ordering of Data memory. Like in the previous design, an instruction consists of four syllables, as the issue-width of the most promising solution was determined to be four during the simulation phase. Thijs mentioned in Section 4.2 of his thesis that the standard VEX ISA consists of 73 operations, but their are really 70 native operations [9]. The different operation classes and their cardinality follow:

- 24 arithmetic ALU operations
- 11 multiplication operations
- 15 logical ALU operations
- 9 memory operations
- 9 control operations
- 2 inter-cluster operations

We add one additional operations to this set, which is the *move from branch* (**MFB**) operations. As we are not aiming to design a *multi-cluster* architecture, the inter-cluster **SEND/RECV** operations are reserved, but are not implemented in our VEX design. The pre-fetch operation is not supported in our design neither, because we are not going to implement a separate data cache. Thus, we end up with a set of three additional VEX operations that have to be encoded and supported. This includes the **STOP**, **LONG_IMM** and **MFB** operations. The latter is actually a pseudo-operation that the VEX compiler emits for copying the content of a branch register to a General-Purpose register when it spills the branch register. We choose to implement this operation as a new operation, since the VEX documentation is unclear whether this operation may target a branch register. Furthermore, we support the **STOP** operation from the multi-cycle implementation. However, in our design this operation signals to the pipeline that the instruction stream has become exhausted and that no more instruction have to be read from Instruction memory. The long immediate syllable (**LONG_IMM**) deals with support for 32-bit integer constants. As was mentioned earlier, the VEX operation set contains 70 native VEX opcodes. We partition this set in the operation classes that are presented in Table 5.1. Moreover, we have devised the encoding scheme that is depicted in Figure 5.3. Although this was a tedious task, it proofed necessary as our first priority was to support all the native VEX operations. We had to perform this step well, because design mistakes would ripple through the implementation phase of the VEX processor. In essence, this is an encoding problem that has to do with the following three issues:

1. Logic operations with a branch register as target and a 32 bit *long immediate* as operand.
2. Encoding utilization of long immediate as operands in ALU operations or as a memory offset.
3. Architectural support for control operations with a branch offset that is larger than 12 bits.

Opcode	Class	VEX
10-----	LOGIC	16
000----	MUL	11
0100---	CTRL	12
0010---	MEM	8

Table 5.1: Partioned VEX opcodes

Formats	Opcode		Imm Type	GR / Br Dst		GR Src 1	GR/BR/IMM Src 2		L	F
	31		23			15	7		x	x
ALU3_Imm	LogicOp	1	0 1		Br	Gr	9 bit imm		x	x
ALU3_Reg	AluOp	0	0 0	Gr		Gr	Gr	Br	x	x
Call/Branch	Branch		20 bit imm					Br	x	x
Load/Store	Memory		0 1	Gr		Gr	9 bit imm		x	x
Special	Reserved								x	x

Figure 5.3: VEX syllable layout

As was discussed above, we concentrated on the augmented set of 73 VEX operations. Other *pseudo-operations* like **MOV** and **MTB** are removed from the opcode list and are implemented using native ALU operations in the assembler. The same applies to the **XNOP** operation when the VEX compiler emits explicit multiple nop operations. Consequently, we gain extra free opcode values as incomparisson with the the multi-cycle design. We note that this encoding requires more opcode space than what was previously required, but that it does solve the encoding problems.

5.3.1 Arithmetic Syllables

The ALU syllable layout is adjusted such that a clearer distinction becomes possible between arithmetic and logic operations. Arithmetic operations have a General-Purpose register as their destination and are executed on the ALU and MUL Units. ALU operations with a branch register as destination are assigned to the class of logical operations. Consequently, we have devised an encoding scheme to indicate this in a more explicit way. Arithmetic operations have two operands of which the second one can be a GP register field or an immediate. Usually their destination is a GP register, but for logical

Immediate type	Size[bits]	Immediate switch
none	NA	00
short immediate	9	01
branch offset	12-20	–
long immediate	32	11

Table 5.2: Immediate types – Adapted from [4]

operations, the destination is a branch register. Like for ALU operations, the second source operand of a logical operation can be a short- or long immediate. This requires us to utilize the adapted *immediate switch* that is presented in Table 5.2.

The operations **ADDCG/DIVS** and **SLCT/SLCTF** are slightly more complex, as they have three source operands: two GP registers operands and one BR register as their third source. Moreover, within this operation set the **ADDCG/DIVS** have a GP register as destination. We have utilized the *source address packing* technique that assigns shorter opcodes to all these syllables, in order to pack the BR source field in the first three opcode bits [4]. As the **ADDCG/DIVS** operations can not have an immediate as source operand there is no problem to utilize bit position 2 until 11 for placing the second GP register field and the BR source field. This is illustrated in Figure 5.4.

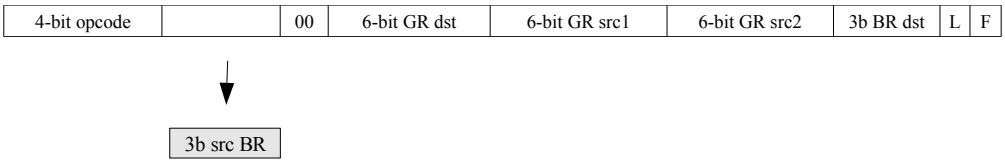

Figure 5.4: Syllable layout for **ADDCG** and **DIVS** operations

.

Despite the fact that the partial-predicate select operations, **SLCT/SLCTF**, can not target a BR register as their destination, the source branch field is packed in the same way as for the **ADDCG/DIVS** operations. This makes the **SLCT/SLCTF** operations and the **ADDCG/DIVS** operations similar. Furthermore, this solves the encoding problem when the subset of **SLCT/SLCTF** operations contains an immediate field as their second source, as it would be overlapped when the BR source register field is placed in the empty BR destination field. By packing the BR register source field in the first three opcode bits, it becomes possible to support a short immediate as the second source operand for **SLCT/SLCTF** operation. For a short immediate field this is depicted in Figure 5.5. However, it becomes necessary to introduce two additional arithmetic types for the **SLCT/SLCTF** operations because the contained immediate can be a short or a long immediate. The long immediates are dealt with by carrying the upper 22 bits by one of the other syllables with the opcode **LONG_IMM**. The lower 10 bits are dealt with by the syllable with the actual **SLCT/SLCTF** opcode. Needles to say that the syllables immediate field must be set to long immediate value in the assembler.

5.3.2 Logic Syllables

The biggest difference between the VEX encoding scheme and the ρ-VEX encoding scheme is how logical operations are specified. Logical operations have boolean results and write their condition in a branch register. When this operation will write a branch destination register, a *write_branch_flag* is reserved in the first opcode bit and must be set by the assembler. As these operations may also target a General-Purpose register, we

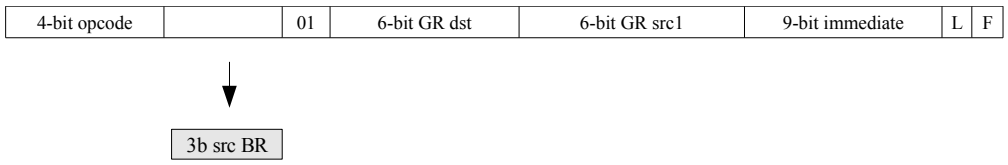

Figure 5.5: Syllable layout for **SLCT** and **SLCTF** operations

.

lower the branch target flag such that the operation 'morphes' to a standard three source ALU operation. Although logical operations with two General-Purpose source registers were implemented in the multi-cycle processor, their was no support when they had an immediate as their second source and a branch register as their destination register. With the proposed, scheme this is solved in our design. Analyzing the instruction layout of the ρ-VEX design, we conclude that the problem is caused as the last 3 bits of the immediate field overlap with the branch source register field [4].

Thijs van As proposed in his thesis the solution to utilize the branch immediate switch when a branch register is a destination, and to utilize the short- and long immediate switches to indicate when a General-Purpose register is a destination [4]. As a consequence, one can pack the branch destination field in the General-Purpose destination field. Subsequently, the branch immediate switch has to indicate that the logic operation contains an immediate operand. Although this solution appears to be correct, the problem we faced when we tried to implement it, was that a similar problem resulted in the encoding space. With this approach and by sticking to the same instruction encoding, one can not encode whether the immediate operand is a short or a long immediate. The reason is that there are simply no more syllable bits left to encode both alternatives.

As is described, above we utilize the lowest opcode bit to encode that a branch register is a target of a logical operations, but this requires remapping the opcode values of all memory, control and alu operations. As a result the logical and arithmetic ALU operations are now one bit shorter in their opcode field. The first opcode bit indicates whether a branch register is the destination of a logical operation. Moreover, this remapping requires partitioning the the set of ALU operations with two sources in an arithmetic subset and a logical subset. Additionally, it is required to map pseudo-operations to basic VEX operations. As a result all logical operations are now correctly supported as is depicted in Figure 5.3. Moreover, we keep the formats of logical and arithmetic operations with 3 source registers similar by keeping the branch destination and General-Purpose source fields in the same position. This similarity reduces the hardware complexity of the Decode stage.

5.3.3 Branch Syllables

There were three design issues that had to be addressed in order to devise the correct architectural support for control operations. After Analyzing the ρ-VEX design, we determined that these issues stemmed from the following:

1. Having to change the branch offset field from 12-bits to 20-bits.
2. Support for indirect control operations: **IGOTO** and **ICALL**.
3. Architectural support for *PC-relative* addressing in the pipeline.

As is stated in the ρ-VEX design it was decided to limit the branch offset to 12 bits for efficient utilization of the syllable layout [4]. The 6 bits of the GP destination register field are utilized to store the link register field that is needed by the **CALL**, **RETURN** or **RFI** syllables. The **CALL** syllable indicate that the return address (address of the next instruction) has to be written to the link register in the WriteBack stage. When a routine passes control back to the caller, the **RFI** or **RETURN** operations support the interrupt routines and function call mechanisms for reading the return address. This address was previously stored in the link-register by a **CALL/ICALL** operation. As the multi-cycle implementation was to run as a co-processor for the MOLEN processor, this limitation of the branch offset will likely present no problem for ρ-VEX design. However, as we are designing a stand-alone processor the compiler can easily emit a branch instruction that requires an offset larger than 12 bits. Conseqeunty, we have taken care to increase the branch offset to 20 bits by devising a different encoding scheme for control operations. Since the **CALL**, **RETURN** or **RFI** operations by definition will always write to the link register, there is no need to occupy 6 bits in the GP destination register field when we hard-wire this in the Decode stage. More specifically, when the Decode Unit encounters these opcodes it sets the destination in the case of a **CALL**, or the source address in the case of a **RETURN/RFI** syllable, to the link register address. This frees up 6 encoding bits. The next 2 bits can be added to the offest by simplifying the micro-architectural connections between the Decode Unit and the Branch Unit. We choose to always sent the offset bits to the Branch Unit in the case of control operations, as we can take advantage of a more regular syllable layout in which these bits are always located on the same place. Consequently, this keeps the instruction format regular. With these results we have regarded the branch offset immediate switch to be redundant. Clearly, immediates that are not *long* or *short* must be branch immediate. Furthermore, the immediate switch is no longer utilized to overload indirect control operations as was previously perfromed. The Branch Unit can deduce the immediate type from decoded opcode value of an control operation. This means that these operations may involve a 20 bit branch-immediate [1]. Therefore, the last 2 bits increase the number of offset bits to 20 bits, which is only 4 bits less than what VEX ISA defines. When we elaborate on the re-designed assembler in Chapter 8, it becomes clear that the assembler must emit an exception in the event that the branch offset is larger than 20 bits.

The VEX compiler no longer emits the **IGOTO** and **ICALL** indirect control operations. Although this issue was addressed in the ρ-VEX design, we have determined that it was implemented incorrect. The semantics of **IGOTO** operations are that the program counter has to be loaded with an instruction address that is stored in the link register. This makes it possible to execute an absolute indirect jump in the micro-architecture. Because the compiler no longer explicitly emits these operations their semantics must be inferred from the context of the direct control assembly operations. This is necessary,

[1] The *branch offset immediate* may have a size in the range of 12–20 bits

Operation	Description
GOTO off	Unconditional relative jump
IGOTO lr	Unconditional absolute indirect jump to link register
CALL lr = im	Unconditional relative call
ICALL lr = lr	Unconditional absolute indirect call to link register
BR b, off	Conditional relative branch on true condition
BRF b, off	Conditional relative branch on false condition
RETURN t = t, off, lr	Pop stack frame (t = t + off) and *goto* link register
RFI	Return from interrupt

Table 5.3: Control operations in VEX

otherwise the micro-architecture can no longer support function pointers, nor can it support case statements with more than eight possible alternatives. The **ICALL** operation has the semantics of an unconditional absolute function call that has its return address stored in the link register. This operation implies that for the **ICALL** operation the return address must be stored in the link register when the callee returns to the caller. For the multi-cycle design, the problem partly was in the assembler and a **peculiar** decision of the designer not to make a distinction between indirect and direct control operations. The indirect nature of these operations can be inferred from the context, as the VEX compiler emits a **CALL** or an **GOTO** operation with the link register as the source register. In the assembler it seemed that Thijs van As did not overload the **CALL** or **GOTO** opcodes with the indirect **ICALL** and **IGOTO** opcodes. This was required as unlike the compiler, the processor must still support these control operations such that the Decode Unit can decode them for the Branch Unit. In turn this unit can perform the required micro-operations with the semantics of indirect control operations. Therefore, we took extra care to provide the required micro-operations in the Decode stage and the required programming in the re-designed assembler.

Another strange feature follows from the fact that the multi-cycle implementation does not distinguish between *absolute* addressing and *PC-relative* addressing, while the VEX ISA clearly dictates this. All control operations were implemented as absolute jumps to the target instructions. Although such a scheme can be some what faster, it clearly violates the hardware/software interface of the ISA. This could have caused some of the design problems and errors that we encountered in the assembler and in the multi-cycle processor's hardware. We chose to stick to the ISA to make a clear distinction between these two types of addressing modes in the micro-architecture of the VEX processor. For PC-relative addresses, this means that the Branch Unit adds the branch-offset to the current instruction address, such that the branch target can be passed to the Addresses Generation Unit. The *absolute addressing* mode requires that the program counter is loaded with either the address of the target instruction in the case of an **IGOTO** syllable, or with the indirect address in the case of a **ICALL** syllable. As a last remark, we note that their was no decoding support for the **RFI** syllable in the ρ-VEX processor. In our VEX architecture these are supported in the same way as the more common **RETURN** operations.

5.3.4 Memory Syllables

Memory operations were only slightly adjusted as we have utilized the syllable template from the multi-cycle design. In that design the immediate switch did not encode the presence of an immediate. This must represent the offset type that has to be added to the base address for accessing Data memory. Within a memory syllable, the second source register field contains the address of the base General-Purpose register. The destination field contains either the source register for a store operation, or the addresses of the destination register. This register receives the value which is loaded from Data memory. The 9 bit offset becomes too small when integer arrays with more than 512 entries have to be addressed. For this reason, we have swapped the position of the Load/Store Unit with the second MUL Unit. This is a valid micro-architectural alternative for dealing with long immediates. Furthermore, we utilize the special opcode value to indicate that the upper 22 bits of the immediate are carried by a free syllable slot [4]. We note that for our design with four issue slots, the VEX compiler will take care to schedule at most two long immediate in the same instruction. This means that there will always be enough free operation slots available to carry the remaining 22 high immediate bits [9]. The syllable template for memory operations is depicted in Figure 5.1.

5.3.5 Special Syllables

The **STOP** and the **NOP** syllables are utilized in our design [4]. Their templates are left unchanged, but the **STOP** operation has a different opcode as we consider it to be a special branch operation. It signals the end of a programs execution. For convenience the layout of these syllables is depicted in the last row of Figure 5.1.

5.4 Pipeline Micro-Architecture

Most operation in VEX are translated to low level micro-architectural operations that we abbreviate as micro-operations. Five sets of micro-operations are identified, where each set roughly corresponds to a stage of the pipeline's micro-architecture:

1. **IF:** Instruction Fetch
2. **ID:** Instruction Decode
3. **E0:** Execute and Memory Read
4. **E1:** Memory write and Pre-select
5. **WB:** WriteBack

By comparing Figure 5.6 with the proposed organization, we see that the micro-architectural components correspond to roughly the way how operations and data move through the datapath. In general this direction is from left to right, since data and control information passes through the datapath stages before committing the results in the WriteBack stage. As this is a 4-issue machine, the datapath consits of four parallel *pipe-lanes*, where each *lane* models an independent pipeline. Within this figure their are

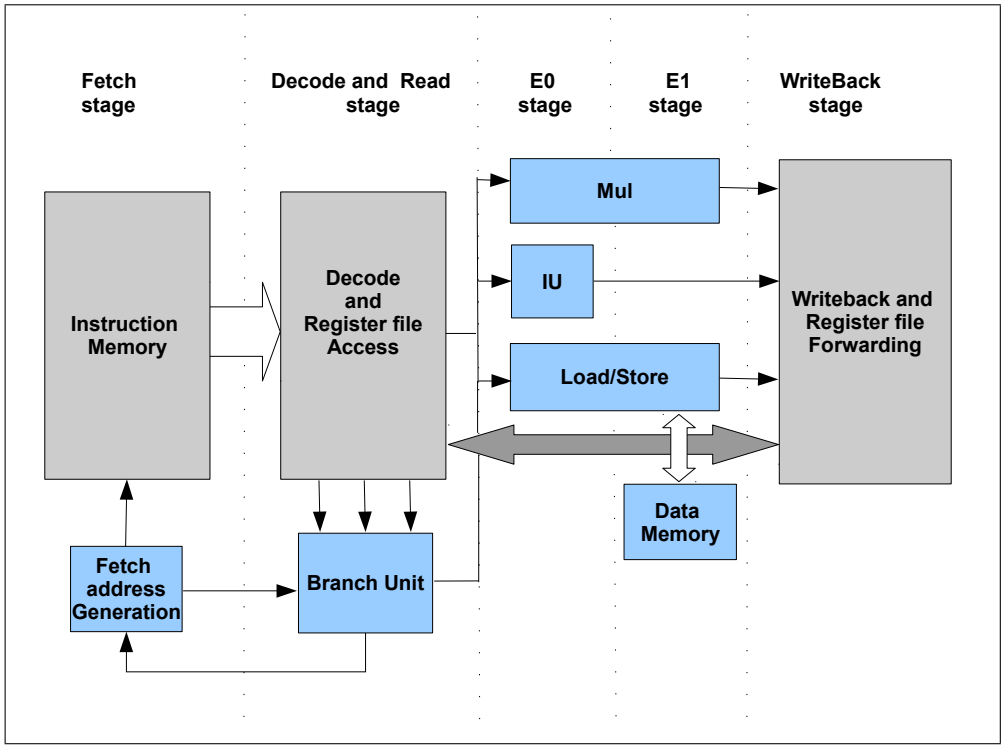

Figure 5.6: Pipeline Architecture

two exception to this left to right flow. First, the selection between the next instruction address and the branch target address. This is controlled by the Branch Unit and it is executed by the Address Generation Unit in the Fetch stage. Second, the WriteBack stage sends the results and register addresses back to the Register Files in the Decode stage.

In the VEX pipeline neighboring stages are separated by pipeline registers. The VEX pipeline has four of these registers:

- **IF/ID**
- **ID/E0**
- **E0/E1**
- **E1/WB**

Except for the different state elements like the Register Files and Data memory within each stage, operations are executed in an asynchronous way such as to produce results that can be stored in time in the next pipeline register. The set of asynchronous micro-architectural components consits of:

- Address Generation Unit
- Decode Unit

- Branch Unit
- Functional Units
- Load/Store Unit

In the next section we discuss the design of these components and we present the architectural details of the pipeline of the VEX processor.

5.4.1 Fetch Stage

The first stage of our five-stage pipeline is the Fetch stage. Considering the hardware of the designed architecture three related micro-operations are performed in this stage:

1. Address Generation
2. Access Instruction Memory
3. Stop Synchronization

Based on these micro-operations, we have divided the micro-architecture of this stage in three architectural components. The names of these components match the above three tasks. During our initial design of the pipeline architecture, we assumed that the Instruction memory was an integral part of the Fetch stage. When the design matured, we pulled out the instruction memory out of the this stage, such that our VEX core obeys the Harvard architectural principles [13]. The Address Generation Unit computes the instruction address of the next instruction. Consequently, an important part of this unit is formed by the program counter register. The output from this register is the address of the current instruction that is being fetched from the Instruction memory of the processor. This memory receives the address and knows what instruction to send to the **IF/ID** pipeline register. Another part of this Unit consists of an adder to compute the next instruction address and some selection logic. Upon decoding a branch operation, the Address Generation Unit is controlled by the Decode stage to select the branch target address such that the desired branch instruction is fetched from the instruction memory. The branch target is computed by the Branch Unit in the Decode stage. In the case a taken branch is encountered, the Address Generations Unit flushes the instruction that was read from the Instruction memory. When the instruction stream becomes exhausted the Synchronization Unit has the responsibility to start signaling that the pipeline must stop with processing instructions. Consequently, it signals to the Decode stage that processing the instruction stream is finished. This is the last micro-operation performed in the Fetch stage. A conceptual block diagram of the **Fetch** stage is depicted in Figure 5.7.

5.4.2 Decode Stage

The task of the Decode stage is to receive the instruction that was stored in the **IF/ID** pipeline register in order to transform this to micro-operations for execution in the rest of the pipeline stages. This is the most complex stage in the pipeline. Consequently, great

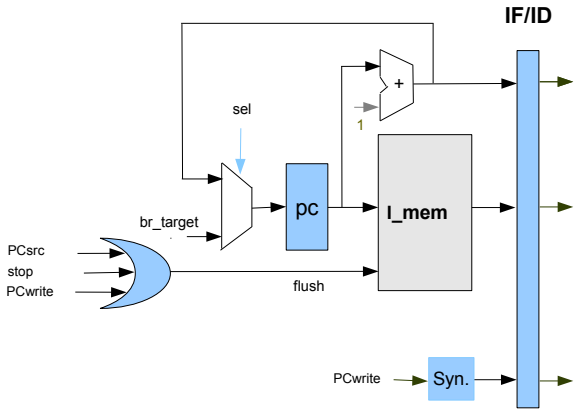

Figure 5.7: Fetch stage design

care was taken to come up with a correct and efficient stage design. This concurrent nature is one of the major differences between the VEX design and the ρ-VEX design. In fact, all five pipeline stages execute their micro-operations in parallell. Focussing on the design of the Decode stage, the following micro-operations are performed in parallel:

- Split an instruction from the Fetch stage into four separate syllables.
- Decode opcodes and extract the register identifiers from each syllable.
- Access the Register Files or generate immediates for some operations.
- Send operands or immediate values to Branch Unit or to ID/E0 register.
- Encode the targets that an operation will write in the WriteBack stage.
- Commit the values from the WriteBack stage to their target registers.
- Pre-compute the effective address for accessing Data memory in E0 stage.

An instruction received from the Fetch stage is split into four syllables. For each syllable, the Decode Unit decodes the the source and destination fields, such that it may read the required values from the Register File and write the datapath results in the WriteBack stage. The decoded destination fields are sent to together with their operand values to the Branch Unit to the **ID/E0** pipeline register. Furthermore, communication with the Branch Unit as part of our **two-step branch architecture** is performed such that this unit receives the required opcode, control information, and data values. In essence, such an architecture decouples the compare operation from the actual branch operation. This exposes the branch latencies to the VEX compiler and enables it to schedule independent operations during these cycles. The branch flag that was read from the branch register is sent to the Branch Unit, which directly controls the selection of the branch target in the Fetch stage. We have composed the micro-architecture of the Decode stage from the following components:

- Decode Unit
- Branch Unit
- GP Register File
- BR Register File

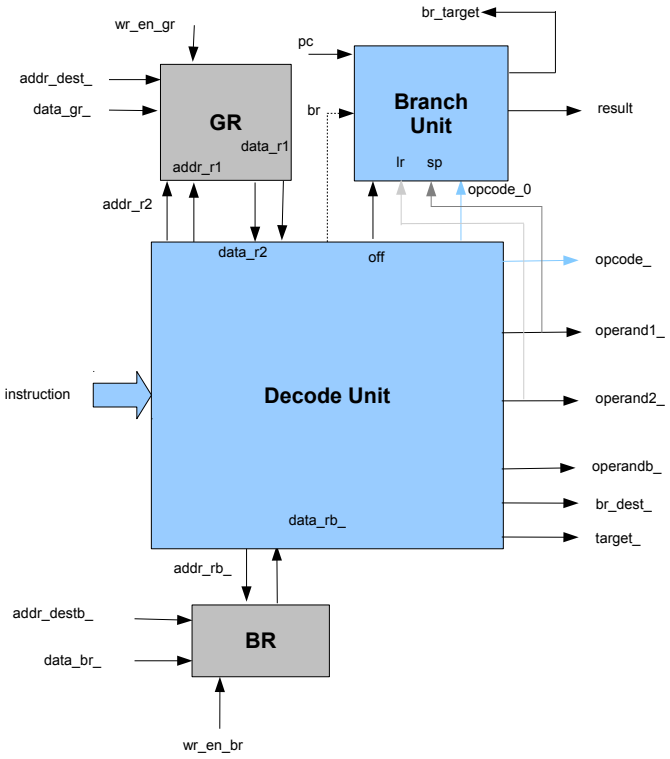

Figure 5.8: Micro-Architecture Decode stage

From the above discussion, it should come as no surprise that the Decode Unit is the central component that orchestrates most of the micro-operations that were just discussed. The Branch Unit performs the actions that are required in the second phase of our two-step branch architecture. Furthermore, all syllable are decoded with combinatorial logic in collaboration with the Branch Unit. Later in this section, we see that the Branch Unit decodes and schedules all control syllables. The micro-architecture that is designed for the Decode stage is depicted in Figure 5.8. The Register Files (RF) just store the architectural state of the processor. Even so their design is critical in a sense since the WriteBack stage and the Decode stage may exhibit Read After Write hazards (RAW). Furthermore, the design of the Register Files is important, because they are accessed on almost every cycle in our Load/Store architecture. The in depth design details of the register files are discussed next.

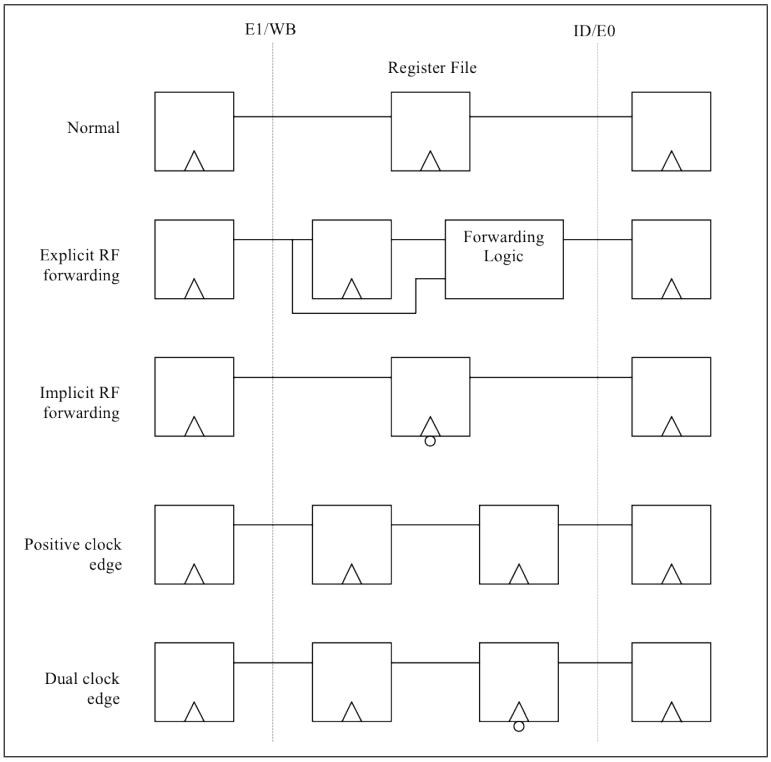

Figure 5.9: Register File timings

Register Files

We designed the Register File (RF) by reflecting on a subset of the pipeline timing requirements and by anticipating on the relationship that the Register File would have with other pipeline stages. What complicates this design is that the number of read and write port is high, because each of the 64 registers have 8 read ports and 4 write ports. This may be understood by taking into account that our pipeline is actually composed of four parallel *pipe-lanes*, where each lane requires additional read an write ports for the registers in the Register File.

We have investigated different timings alternatives, in order to find suitable alternatives that can be utilized in the Register File design. The result of this investigation is depicted in Figure 5.9. For each of the drawings, we have unfolded a conceptual representation of the VEX pipeline between the WriteBack and the Decode stage. An explanation of the depicted timing diagrams follows:

- **Normal Register File**: The first illustration depicts a datapath that contains a straight forward design of a RF where a set of registers is simply a stack of 64 registers. Each register is written and read on a positive clock edge. Since each register in the RF is build of banks of parallel 32-bit D flip-flops, with such a design our Decode stage will supply the old value in the event that the WriteBack stage writes to the same register that is being read by the decode stage. So this will present a RAW data hazard in our pipeline. Although it supplies the incorrect

value, this alternative of the RF hands the designer the convenience of having to deal with a single clock edge in the entire pipeline micro-architecture.

- **Explicit RF forwarding**: In addition to stacking registers to form a RF, we need additional forwarding logic to detect the RAW hazard so that we can forward the updated values to the read ports. Because this logic must be added for each of the read ports, the area of the RF increases. But the good thing is that we can deal with the RAW hazard.

- **Implicit RF forwarding**: By trading-off performance for area, one can optionally deal with RAW hazard from the Register Files. In this case, we have to require that the register's content is read in an asynchronous way and that the writes only happen on a falling clock edge. This means that the RF does not require additional forwarding logic. This saves on area but makes the RF design slower. The write logic in the RF should fit within half a clock-cycle, which implies a higher lowerbound on the clock-cycle time. Therefore, this alternative will decease the clock frequency.

- **Positive clock edge**: The fourth alternative illustrates the RF as it was designed for the multi-cycle processor. For our purpose, this presents an invalid alternative because we will get an extra delay between updating the register value and making that value available for the **E0** stage. This alternative does explain why writes in the multi-cycle design require an extra clock-cycle.

- **Dual clock edge**: This is a variation of the previous alternative, since it also allows us to update the RF on a negative clock edge. Although this version comes close, we see that their is a delay of half a clock-cycle between updating the RF by the WriteBack stage and outputting the correct value at the **ID/E0** pipeline register. This means that the FU in the E0 stage will still receive the old values from the RF as their operands.

A more detailed timing diagram of what happens at the signal level in the Register File is depicted in Figure 5.10. From the above discussion and based on our investigation of the possible alternatives, we conclude that their are two valid designs possible for our Register File. Because the Branch and General-Purpose Register files have the same timing requirements, it is not required to investigate an extra set of timing requirements. There are some implementation details that are different in these two types of Register Files. However, these are mainly caused by the fact the Branch Register File has a single write port per pipe-lane and its storage containers are just one bit wide. We discuss the implementation details of both Register Files in the Chapter 4. This chapter covers the implementation of the micro-architecture of the VEX processor.

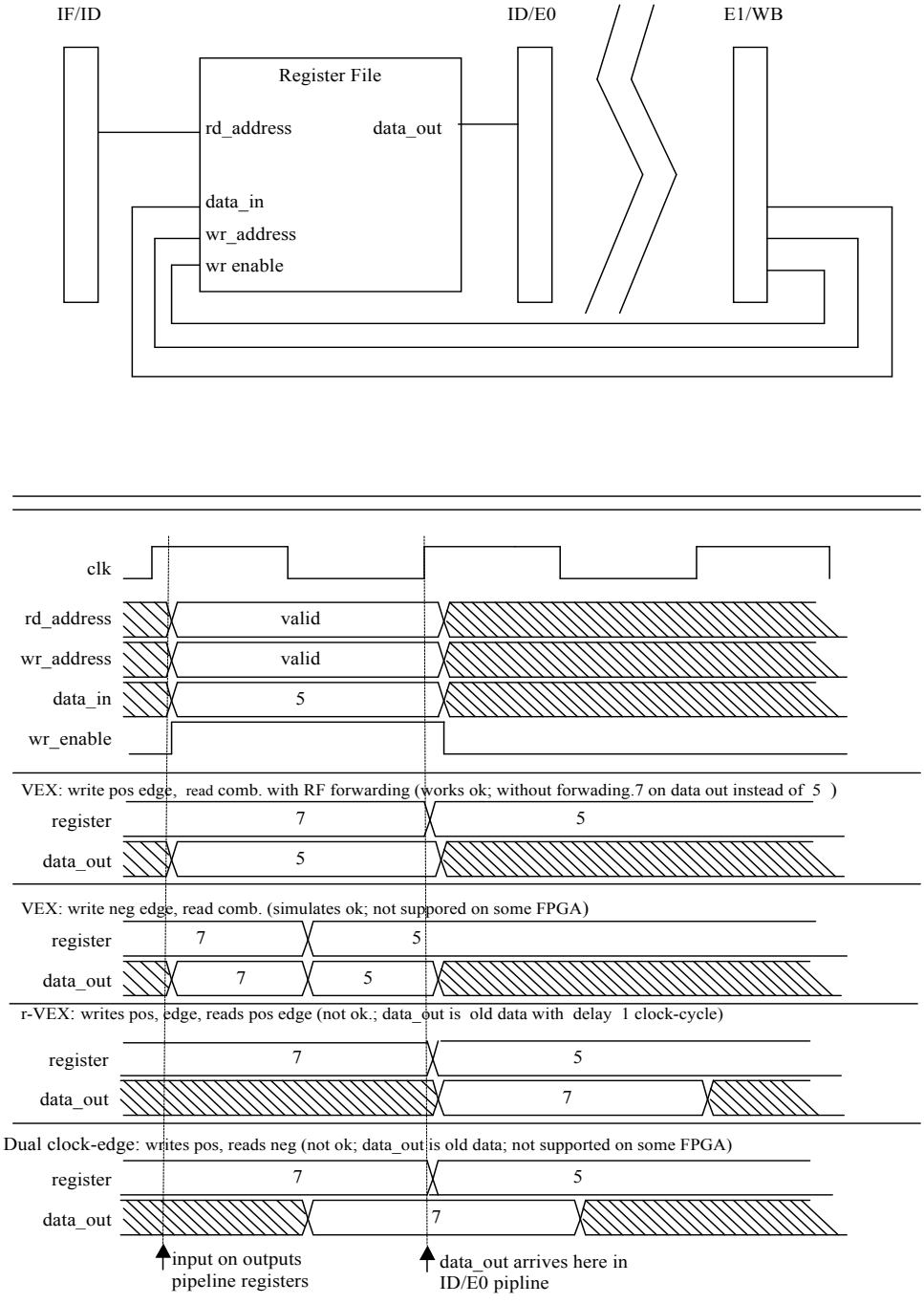

Figure 5.10: Detailed Register File timings

Branch Unit

Branch operations are mapped to syllable number 0 and are scheduled on the Branch Unit in the Decode stage. In VEX ISA their are four types of branches that the branch micro-architecture must execute:

- *Conditional branches* these change the control flow based on a condition. When the condition is false the execution proceeds on the fall-through path.
- *Unconditional jumps* these always change the control flow as the branch is always taken. This means that the execution continues on the taken-path.
- *Function calls* better known in other architectures as *jump-and-link*. These operations are unconditional jumps that save the address of the instruction following the call, e.g. the return address.
- *Indirect jumps* that unconditionally cause the execution to jump to a variable address that is usually stored in a General-Purpose register. Function return belong to this type of control operations as they cause an indirect jump that returns to the location saved by the caller.

A more concise description of all the branch operations and their semantics is given in Table 5.3. The Branch Unit is assisted by the Decoder and the Address Generation Unit in the Fetch stage. Looking back at the Figures 5.6 and 5.7, we see that the Branch Unit receives the control and data information from the following architectural components:

- **IF/ID** pipeline register: The address of the next instruction is received as $PC+1$. This value is needed for executing *PC-relative* control operations.
- **Decode Unit**: The partially decoded opcode from syllable number 0 tells the Branch Unit what kind of control operations is currently being decoded.
- **GP Register File**: The Decode Unit also orchestrates sending both the values of the link register (lr) and the stack pointer (sp) to the Branch Unit. Note that the Branch Unit needs both values for a **RETURN** operation. For **IGOTO** the value of the lr represents the absolute branch target address. For **ICALL** operation the same holds, however the address of the next instruction (return address) should be saved in the link register such that the callee is able to jump back to the instruction after the call. In the case of a **CALL** instruction, the branch to the callee site is performed relative to the program counter (PC). At the same time the return address ($PC + 1$), is send as the value that has to be written to the link register. By reading this value when a **RETURN** operation is decoded, the callee is able to return to the caller.
- **Branch Register File**: The *branch on true/false* operations (**BR/BRF**) are PC-relative control operation, which are executed conditionally. A branch is performed based on the condition that is stored in a branch register. Furthermore, this means that the Decode Unit must send the branch condition that was read from the

Branch Register File to the Branch Unit. In the event of a **BR/BRF** operation, the branch condition **directly** controls the **selection** of the branch target. This target is computed by addition of the branch offset value to the program counter such that the result is send during the same clock-cycle to the **Address Generation Unit** in the Fetch stage.

- **Branch offset**: All PC-relative branch operations like **GOTO** or **BR/BRF**, define that the branch target must be computed by adding an offset to the value of the current program counter. Note that by the time the program counter reaches the Branch Unit, it is already incremented such that the Branch Unit actually receives the address of the *next* instruction as the program counter value. For the **ICALL/CALL** operations, this is convenient as this value can be used as the return address. However, for all other relative branch operations, the incremented value must be corrected. In the implementation phase, we see how the micro-architecture and the re-designed assembler deal with this issue in order not to slow down the branch offset computation. The branch offset is decoded and extracted from syllable 0. For the decoder this is easy as the branch offset is always located in the same position of a control syllable.

From the above control and data information, the Branch Unit is able to calculate the branch target for all control operations. For **CALL/ICALL** operations, the Branch Unit computes the return address as the new value for the link register. This value is placed on the result output of the Branch Unit. For **RETURN/RFI** operations, this result represents the value that micro-architecture has to pop from the callee's stack frame, such that its **activation record** is removed from the *function call stack*. Control operations have to be executed in combinatorial logic such that their result is ready and available to be written to the **ID/E0** pipeline register.

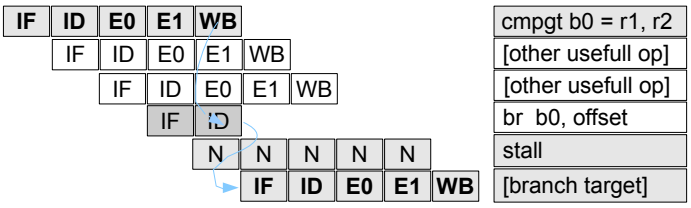

Figure 5.11: Two-step branching

Two-step Branch Architecture

The branch architecture of the processor is designed such that it does not to offset the assumption that the VEX compiler makes when it schedules branch operations [9]. For a VLIW processor this is important as the compiler is responsible to find the ILP in the

program. To reduce the miss-penalty conditional branches are broken and executed in two separate phases:

1. *Whether*: This prepares the condition by specifying the branch in advance of the control flow change point, then it stores the condition in one of the eight 1-bit branch registers.

2. *Where* ***and*** *When*: This computes the branch target based on the condition and selects the instruction stream as the very last action of an instruction.

The two-step branch architecture also breaks the conditional branch operations into *compare-branch* micro-operations [9]. Based on requirements of this micro-architecture, we placed the Branch Unit in the Decode stage. When the Branch Unit is in the decode stage the computation of the branch target and its selection can be performed in the same clock-cycle during which the branch operation is decoded. Although it can be argued that moving the Branch Unit to the Decode stage complicates our pipeline's branch micro-architecture, the main micro-architectural component which is effected by this design decision is the forwarding logic. Since the Decodes stage has a set of branch register at it's disposal, legal code sequences can also prepare multiple branches at a time so that the compare operations can be moved any where by the VEX compiler [9]. The actual branch operation (*when*) still requires an address computation together with the branch decision (whether) itself. In our micro-architecture the branch target is computed speculatively for each control operation during the second pipeline stage, because this is the earliest point where the branch target offset can be decoded. Furthermore, selecting the branch target is controlled directly by the condition that is read from the branch register. This is our second motivation to place the Branch Unit in the Decode stage.

Looking at Figure 5.11, another point of discussion is that one may argue that the VEX compiler can be smart enough to find the first useful operations but that it can fail in its second attempt. In this case a nop will be scheduled and the pipeline will stall for one clock-cycle. The solution to this problem is that the result of the branch condition may be forwarded, since it is already available in the **E0/E1** pipeline register. As a result the architectural visible latency will be one cycle shorter, so that the branch target can be reached one clock-cycle earlier. We discuss this aspect of the design in Section 5.5.1. In essence, this problem requires adding an extra set of forwarding equations in order to detect this situation at run-time.

The above discussion concerns conditional branch operations. However, looking at Table 5.3, we see that a larger subset of branch operations is formed by the unconditional branches. The operations **IGOTO** and **RETURN** are examples of unconditional branches. For our target application the branch frequency for unconditional branches versus conditional branches is high: 15.84% vs. 1.54%. When the Branch Unit is placed in the first execution stage, the miss-penalty of a taken branch will be 2 clock-cycles of work, because we need to flush the operations in the Fetch and Decode stage. This means that 40% of the pipeline's work is thrown away. For our 4-issue pipeline the effect is even worse, as the number of mispredicted operations is equal to the mispenalty times the issue-width of the machine. This results in actually eight operations that will have to be flushed.

For all of these reasons that we have discussed, we decided to move the Branch Unit in our pipeline's micro-architectural to the Decode stage. What is then required in hardware are the following architectural changes:

- Add extra forwarding logic from the **E0/E1** pipeline register to the branch input of the Branch Unit. In the case that the branch latency is set to one cycle in the machine configuration files, this logic will detect the dependency and bypass the branch condition to the Branch Unit.
- Detect the situation that a branch operation reads a register produced by a previous operation. This would actually require hazard detection logic. However, our VEX compiler has the ability to detect this situation, as we can set the issue-use latency to one clock-cycle for these operations.

In Section 5.5.1, we discuss the forwarding logic and see how we can optimize the pipeline such that it becomes better suited for high-performance embedded applications.

5.4.3 Execution Stage 0

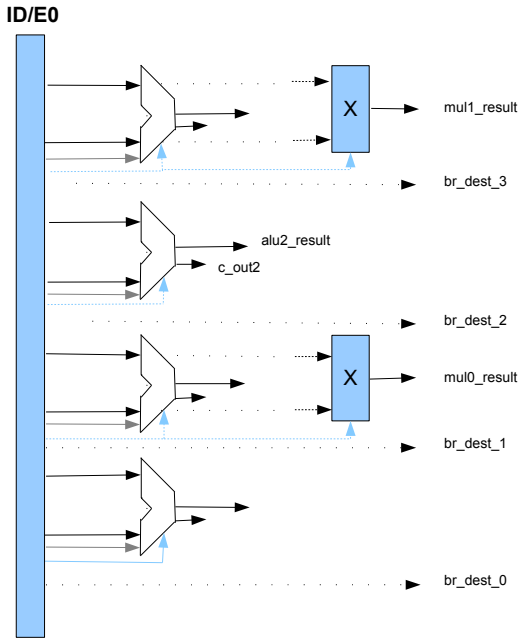

Figure 5.12: Execution 0 stage

The first stage of execution is where ALU- or MUL operations and the *pre-load phase* of a Load/Store operations are all executed in parallel. Since the effective memory address was already computed in the Decode stage, we pre-load the Data memory value. From

the Decode stage, every functional unit receives its operands together with their opcodes, so that each knows which operations it must execute. Like the other functional units, the ALU and MUL Units were designed in a combinatorial way such that in a single clock-cycle all results are computed and ready to be written to the **E0/E1** pipeline register. A conceptual illustration of the micro-architecture that we designed for the first execution stage is depicted in Figure 5.12. Within this figure, the following functional units are present:

1. 4 parallel ALU Units
2. 2 MUL Units
3. 1 Load/Store Unit

Comparing Figure 5.1 with Figure 5.12, we see that the operation that each functional Unit can execute matches with their position in the pipe-lanes of the processor's micro-architecture. For issue-slot number 3 and 1 the operations can be of type ALU or MUL. Both units receive an opcode and their two operands. Subsequently, depending on the type of the operations the ALU or MUL Unit compute the result in datapath lane number 3 and 1 of the pipeline. In the case of a MUL operations this is simple and the 32-bit (partial) product is computed and ready to be written to the **E0/E1** pipeline register. For ALU operations this situation is different as the ALUs can compute either a 32-bit arithmetic results, or a 1-bit logical operations. In the case of logical operations, the ALU sets the first bit of the 32-bit result. However, it can write and intermediate logical result as part of a compound logical expression, to a General-Purpose register. When the result of a logical operations will write to a branch register, and this result will be utilized as a branch condition, the Decode Unit detects this situations and asserts a the **br_dest** flag to indicate that the execution stage will not want to utilize the output carry of an ALU. The (**ADDCG/DIVS**) operations are example where this happens. Instead the first bit of the ALU result represents the logical condition. For simplicity sake, we omit three things in Figure 5.11:

- The load phase of the Load/Store Unit.
- The 32 bit result from the Branch Unit.
- The *targets* from the Decode Unit.
- The write addresses for WriteBack stage.

The Load/Store Unit receives a byte offset, a write enable signal and an effective address. Utilizing this information it is able to indicate the Data memory which byte should be adjusted in the loaded word for a **STB** or **STH** operations. For a *Store Word* operation the complete 32-bit value that was read from the source register in the Decode stage, is written to Data memory. The same discussion holds, for other store operations like the Store Byte (**STB**) operation.

During the second pipeline stage for **CALL/ICALL** operations the Branch Unit had computed, the return address value for the link register. For a **RETURN** operations

the updated value was computed for the stack-pointer register. So either the updated link register or stack pointer value is recorded as a *result_0* signal. This value remains unchanged as it passes through the rest of the pipeline stages. Each operation that was going to commit a result requires that the Decode Unit has to encode the specified targets. These targets are written in the last pipeline stage by utilizing the write addresses for the register files that were passed to the WriteBack stage. In our design there are only three possible targets that each operations can write:

- a GP register
- a BR register
- a BR and GP registers

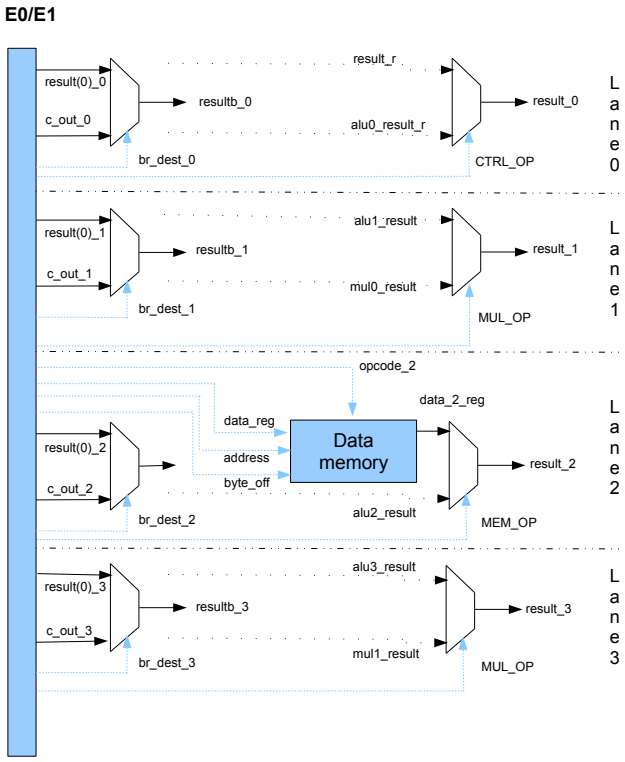

Figure 5.13: Execution 1 stage

The combination of writing both General-Purpose and branch registers is only possible for the **ADDCG/DIVS** operations. General-purpose register are written by arithmetic or logical operations. Branch register are written by logical operations. Of course the **NOP** operation may not change the processor's state, because it does not write any registers.

5.4.4 Execution Stage 1

The second pipeline stage is where the results of the different Functional Units become available. This stage selects what particular result will move to the WriteBack stage. As their are 4 pipe-lanes with two types of Functional Units per lane, this stage performs a pre-selection of results that will move to the WriteBack stage in the next clock-cycle. Each issue-slot contains an ALU. For operations that will write a branch register, we have to select whether either the first bit from the ALU result, or the carry-out of the respective ALU will function as a branch flag. This selection is controlled by the *branch_dest_* flag. This flag was passed from the Decode stage via the **E0/E1** pipeline register as a control signal for the **E1** stage. Furthermore, for pipe-lane 3 and 1, the result can be from an ALU or MUL Unit. Because we mapped the opcode values such that each operation class forms a disjoint set among the different operation types, we can take the opcode of each operation and test it to determine which set it belongs to. Based on this test, we are able to select the output from the right Functional Unit. This is why in Figure 5.13, the second set of multiplexers is controlled by a matched signal of an opcode field to a particular operation class. When a load operations was scheduled on lane 2, we receive the Data memory address with the byte position, and a signal that indicates whether data has to be loaded from memory. By asserting these control signals correctly, we can load the desired value from Data memory such that the result appears on the *data_2_reg* output. Once all results are ready, we concurrently select the required results in the pipe-lanes and write them to the **E0/E1** pipeline register.

In Figure 5.13 we have omitted several important signals, such that this figure is not clouded by wires. As was already stated the Decode stage sends the addresses of the destination registers with their encoded targets to the **E0** stage. Subsequently, the **E0** stage sends this information to the **E1** stage. In Section 5.4.1, we see that the WriteBack stage decodes the targets and receives the control signals and functional results from the **E0** stage.

5.4.5 WriteBack Stage

The last stage of the processor controls how the results of the previous stages commit in the Register Files. As was already explained in section 6.4.2, the Register File within the Decode stage may exhibit RAW hazards. For this reason, the WriteBack stage needs to take care of *implicit* RF forwarding. This means that reading the RF must be performed by utilizing combinatorial logic as soon as the register identifiers for an operation are decoded. In this case, the WriteBack stage must always update the Register File on the falling clock edge. When we implement the Register Files with *Explicit* RF forwarding, this stage only writes on the rising clock edge and the forwarding logic forms an integral part of the Register File design. We could decide to add forwarding logic from the WriteBack stage to the input operands from the **E0** stage or to the inputs of the Branch Unit in the Decode stage, but as we see in Section 5.5.1, this is part of the forwarding hardware that minimizes the architectural visible latencies for the operations that inhibit such dependencies.

In every one of the four pipe-lanes this stage receives the *target* of an operation together with the write address and the data results from the **E0** stage. The target

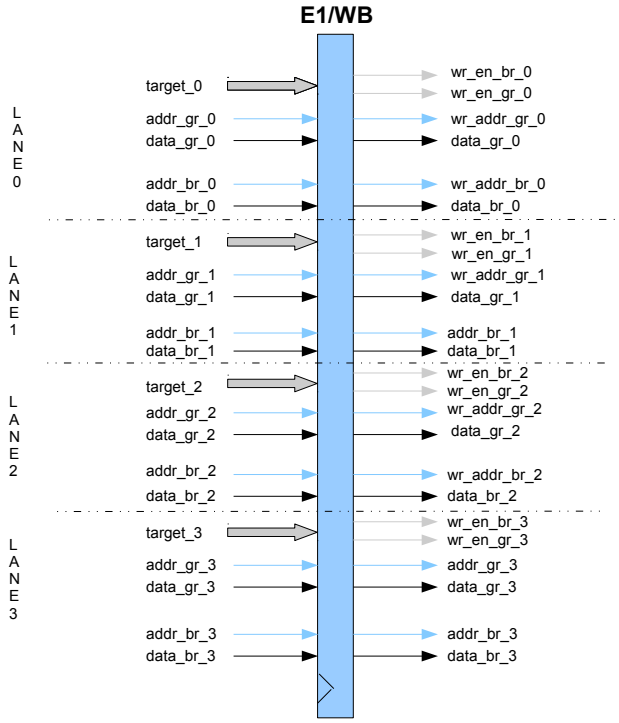

Figure 5.14: WriteBack stage

encodes what kind of register will be updated and their are four possibilities:

1. A GP register as target
2. A BR register as target
3. A GP and BR are targets
4. There is no target

When both register will be written in the WriteBack stage the operation is either a **ADDCG** or a **DIVS**. The **NOP** operations make up the fourth possibility, as they do not write the register files. Based on the targets that are received for each operation, the WriteBack stage decodes the targets and generates an enable signals for the corresponding destination registers. In the fifth stage the data results and the address fields are passed through the **E1/WB** pipeline register. An illustration of the architecture of the WriteBack stage is depicted in Figure 5.14.

The next section discusses how Data and Instruction memories were effected by the pipeline's micro-architecture and how we dealt with the resulting issues in the VEX processor's design.

5.4.6 Data Memory

We designed Data memory such that it can deal with *structural* hazards and *data* hazards. There is a sequence of memory operations that can cause RAW hazards for Data memory. Furthermore, if we were to utilize the Data memory from the ρ-VEX design in our pipeline, it would inhibit structural- and behavioral hazards. The read latency of a load operations for the multi-cycle design requires an extra cycle when accessing Data memory. This would imply stalling the pipeline after each load word operation, as the result would not yet be ready. Loads from memory would require an extra clock-cycle for reading their results. This operations has a latency of 6 cycles in the multi-cycle design. Of course this would slow us down, which is why we came up with a better design that deals with the said architectural issues.

First let us look at the following code sequence that would create a RAW hazard for the VEX pipeline.

Figure 5.15: Data memory RAW hazard

As we see in the code sequence, there is a data dependency on the effective memory address between the load and store operation. This address is computed for all store operations in the ID stage in the second stage. In the third stage, this memory value and its address are stored in the **E0/E1** pipeline register. Actually, this is done in the **E0/E1** pipeline portion of the Load/Store Unit. This allows the data to be written to the Data memory, during the first half of the fourth clock-cycle. The load operation lags one clock-cycle behind the store operations, but computes the same effective address. In the second half of the fourth clock-cycle, we read the Data memory location in a combinatorial way. However, we take care to use the synchronous read address that was pre-stored in the previous clock-cycle. The data dependency is indicated by the red arrow. The blue arrow illustrates in Figure 5.15 that we are able to read the updated value from the Data memory address by forwarding it.

5.5 Architectural Improvements

The previous section was concerned about the micro-architecture of the components that make up the VEX processor's pipeline. In this section, we discuss three architectural techniques that improve the VEX design. In general there are three aspects that can be integrated in the design such that pipeline matches the most high-performance requirements of any ES application. These aspects are:

1. Forwarding logic
2. Register Files

3. Functional Units

The goal of the first is to minimize the visible architectural latencies. However, as we will see in the implementation phase, this can decrease the clock frequency as forwarding requires selection logic that is located on the critical path of the design. We can utilize register files with Explicit Register File forwarding. This can decrease the clock-cycle time, but it can also increases the area of the Register File. The last technique increases the clock-frequency as it utilize pipelined functional units. When we discuss the implementation phase, we see that the critical path of the design runs through the MULT Units. The next section, elaborates on these techniques in order to improve the micro-architecture of the VEX processor.

5.5.1 Forwarding Micro-Architecture

We mentioned that data hazards are the only potential bottlenecks in the micro-architecture of the VEX design. However, we have not yet discusses how the VEX processor can deal with this. Although we can make use of the VEX compilers ability to deal with data hazards, we can strive to do better in hardware by decreasing the visible architectural latency of the processor's pipeline. Using forwarding hardware, we are able to do so for all of the basic VEX operations. The issue-use latency of most operations in the processor's micro-architecture is 2 clock-cycles when there is no forwarding hardware present. By correctly adding forwarding to the design there are four types of data hazards that are detected:

1. ID data hazards
2. E0 data hazards
3. E1 data hazards
4. WB data hazards

The data hazards are created by the data dependencies that come into existence during run-time between different stages of the processor's pipeline. This happens when a functional unit produces results that other functional units read in order to calculate their own results. The pipeline of the VEX processor consists of four lanes. In order to derive the forwarding equations, we can utilize a simplified pipeline model that resembles a scalar processor. However, this model must be scaled up, in order to derive the data hazards that exist in a same lane of the pipeline and between other lanes. By doing so, we have derived Figure 5.16. This figure illustrates most of the hazards that were listed above. The first dependency represents an E0 data hazard. Such a hazard exists between the outputs of the functional units within the E1 stage and the inputs of the functional units that must read these results in the E0 stage. The second dependency depicts Register File forwarding and was already discussed. Remember that we concluded that this type of forwarding can be incorporated in an *Explicit* or *Implicit* manner in design of the Register File. The third data dependency may hold between **E1/WB** pipeline register and the inputs of functional units in the E0 stage, which can cause a E0 data

hazard. Figure 5.16 visualizes and helps us to derive the set of forwarding equations that is listed in 5.1.

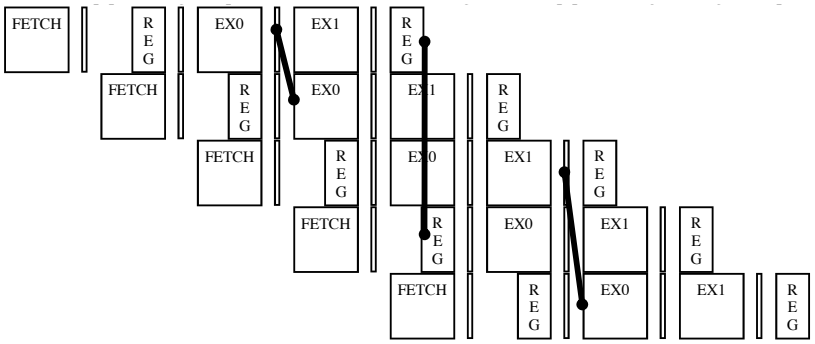

Figure 5.16: Forwarding for register files and functional units

Listing 5.1: Forwarding first operand in E0 stage

```
-- forward logic for operand_1_0
IF (ID/E0.imm_sw != NO_IMM) THEN
  -- no forwarding if we are not reading register
  operand_1_0 <= ID/E0.operand_1_0;
ELSIF (ID/E0.address_r1_0 = 0) THEN
  -- reading register 0 is always treated as reading value 0
  operand_1_0 <= (others => '0')
ELSIF ((E0/E1.target_0_r = WRITE_G) OR (E0/E1.target_0_r = WRITE_G_B)
       AND (E0/E1.dest_0_r = ID/E0.address_r1_0)
  -- newer value is present in 1st pipeline register (lane 0)
  operand_1_0 <= E0/E1.result_0;
ELSIF ((E0/E1.target_1_r = WRITE_G) OR (E0/E1.target_1_r = WRITE_G_B)
       AND (E0/E1.dest_1_r = ID/E0.address_r1_0)
       AND (E0/E1.opcode_1_r != MUL))
  -- newer value is present in 1st pipeline register (lane 1)
  operand_1_0 <= E0/E1.result_1;
ELSIF ((E0/E1.target_2_r = WRITE_G) OR (E0/E1.target_2_r = WRITE_G_B)
       AND (E0/E1.dest_2_r = ID/E0.address_r1_0))
  -- newer value is present in 1st pipeline register (lane 2)
  operand_1_0 <= E0/E1.result_2;
ELSIF ((E0/E1.target_3_r = WRITE_G) OR (E0/E1.target_3_r = WRITE_G_B)
       AND (E0/E1.dest_3_r = ID/E0.address_r1_0)
       AND (E0/E1.opcode_3_r != MUL))
  -- newer value is present in 1st pipeline register (lane 3)
  operand_1_0 <= E0/E1.result_3;
ELSIF ((E1/WB.target_0_r = WRITE_G) OR (E1/WB.target_0_r = WRITE_G_B)
       AND (E1/WB.wr_address_gr_0_r = ID/E0.address_r1_0))
  -- newer value is is present in 2nd pipeline register (lane 0)
  operand_1_0 <= E1/WB.result_0;
ELSIF ((E1/WB.target_1_r = WRITE_G) OR (E1/WB.target_1_r = WRITE_G_B)
       AND (E1/WB.wr_address_gr_1_r = ID/E0.address_r1_0))
  -- newer value is present in 2nd pipeline register (lane 1)
```

```
    operand_1_0 <= E1/WB.result_1;
ELSIF ((E1/WB.target_2_r = WRITE_G) OR (E1/WB.target_2_r = WRITE_G_B)
        AND (E1/WB.wr_address_gr_2_r = ID/E0.address_r1_0))
    -- newer value is present in 2nd pipeline register (lane 2)
    operand_1_0 <= E1/WB.result_2;
ELSIF ((E1/WB.target_3_r = WRITE_G) OR (E1/WB.target_3_r = WRITE_G_B)
        AND (E1/WB.wr_address_gr_3_r = ID/E0.address_r1_0))
    -- newer value is present in 2nd pipeline register (lane 3)
    operand_1_0 <= E1/WB.result_3;
ELSE -- newer value is not present in a pipeline register
    operand_1_0 <= ID/E0.operand_1_0;
END IF;
```

In order to infer the forwarding micro-architecture from these equations, we pass the read address form the ID stage via the **ID/E0** register, to determine whether to forward values from certain pipeline registers. Furthermore, the write addresses from the last three pipeline registers are passed to the forwarding hardware. Additionally, we need to derive the forwarding equations for the second operand for lane 0, 1, 2 and 3 of the pipeline. In the same way, we derive the forwarding equations for the BR operands in the E0 stage. This is depicted in Listing 5.2. Although this approach is straight forward when utilizing Figure 5.16, we see that due to the multiple issue nature of the VEX processor's pipeline, their are many forwarding equations. Likewise in Figure 5.16, we can derive a similar set of equations for operand_2_0, and for all two operands in the other three lanes. This means that we get a total of 8 forwarding equations for the General-Purpose operands of the functional units and 4 sets of equations for the branch operands of the four ALUs.

Listing 5.2: Forwarding branch operand from E0 stage

```
-- forward logic for operandb_0
IF ((E0/E1.target_0 = WRITE_B) OR (E0/E1.target_0 = WRITE_G_B)
    AND (ID.imm_sw = NO_IMM)
    AND (E0/E1.destb_0 = ID/E0.address_rb_0)) THEN
  -- forwarding br from E0/E1 register (lane 0)
  operandb_0 <= E1/E0.resultb_0;
ELSIF ((E1/WB.target_0 = WRITE_B) OR (E1/WB.target_0 = WRITE_G_B)
        AND (ID.imm_sw = NO_IMM)
        AND (E1/WB.destb_0 = ID/E0.address_rb_0)) THEN
  -- forwarding br from E1/WB register (lane 0)
  operandb_0 <= E1/WB.resultb_0;
IF ((E0/E1.target_1 = WRITE_B) OR (E0/E1.target_1 = WRITE_G_B)
    AND (ID.imm_sw = NO_IMM)
    AND (E0/E1.destb_1 = ID/E0.address_rb_0)) THEN
  -- forwarding br from E0/E1 register (lane 1)
  operandb_0 <= E1/E0.resultb_1;
ELSIF ((E1/WB.target_1 = WRITE_B) OR (E1/WB.target_1 = WRITE_G_B)
        AND (ID.imm_sw = NO_IMM)
        AND (E1/WB.destb_1 = ID/E0.address_rb_0)) THEN
  -- forwarding br from E1/WB register (lane 1)
  operandb_0 <= E1/WB.resultb_1;
IF ((E0/E1.target_2 = WRITE_B) OR (E0/E1.target_2 = WRITE_G_B)
```

```
    AND (ID.imm_sw = NO_IMM)
    AND (E0/E1.destb_2 = ID/E0.address_rb_0)) THEN
  -- forwarding br from E0/E1 register (lane 2)
  operandb_0 <= E1/E0.resultb_2;
ELSIF ((E1/WB.target_2 = WRITE_B) OR (E1/WB.target_2 = WRITE_G_B)
       AND (ID.imm_sw = NO_IMM)
       AND (E1/WB.destb_2 = ID/E0.address_rb_0)) THEN
  -- forwarding br from E1/WB register (lane 2)
  operandb_0 <= E1/WB.resultb_2;
IF ((E0/E1.target_3 = WRITE_B) OR (E0/E1.target_3 = WRITE_G_B)
    AND (ID.imm_sw = NO_IMM)
    AND (E0/E1.destb_3 = ID/E0.address_rb_0)) THEN
  -- forwarding br from E0/E1 register (lane 3)
  operandb_0 <= E1/E0.resultb_3;
ELSIF ((E1/WB.target_3 = WRITE_B) OR (E1/WB.target_3 = WRITE_G_B)
       AND (ID.imm_sw = NO_IMM)
       AND (E1/WB.destb_3 = ID/E0.address_rb_0)) THEN
  -- forwarding br from E1/WB register (lane 3)
  operandb_0 <= E1/WB.resultb_3;
ELSE -- newer value ID/E0 pipeline register
  operandb_0 <= ID/E0.operandb_0;
END IF;
```

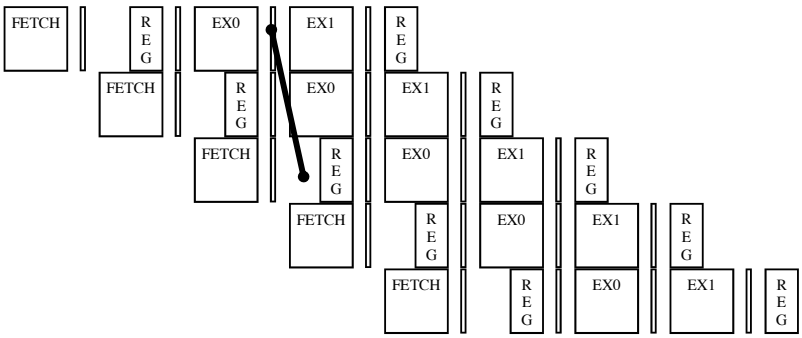

Figure 5.17: Forwarding requires less usefull operations

The Branch Unit within the Decode stage, can read either the branch register, or the link register or the stack-pointer register as its inputs. One of these results may be produced by a later pipeline stages of the processor. This creates the first type of data hazard in the Decode stage. Data hazards in this stage are solved in a similar way by utilizing the read addresses of the branch operands in the Decode Unit, to determine whether to forward branch results from the **E0/E1** pipeline register. For the the more recent values of the link register or the stack pointer register, the source pipeline registers are **ID/E0** and **E0/E1**. Subsequently, the design requires to route the read addresses form the Decode Unit and destination addresses from the **ID/E0** and to the **E0/E1** pipeline registers. In this way, we can derive three set of extra forwarding equations. Forwarding of the branch condition is presented in Listing 5.3. Forwarding of the link register and the stack pointer is illustrated in Listing 5.4 and in Listing 5.5, respectively.

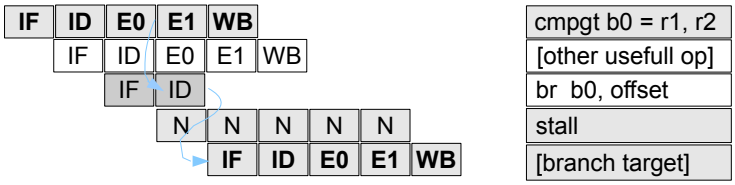

Figure 5.18: Forwarding branch condition to Branch Unit

We have investigated how forwarding of a branch condition effects our **two-step branch architecture**. As forwarding decreases the visible architectural latency, we concluded that it releases the presure on the VEX compiler to find usefull operations. As is depicted in Figure 5.18, we see that it is sufficient for the compiler to find and schedule a single useful operation instead of two operations. This means that it is more likely when such operations can not be found that the pipeline stalls for one clock-cycles less compared with a branch architecture design when with no forwarding of the branch condition.

Listing 5.3: ID hazard on branch operand

```
IF ((EO/E1.target_0_r = WRITE_B) OR (EO/E1.target_0_r = WRITE_G_B)
    AND (E0/E1.destb_0 = IF/ID.address_rb_0)
    -- br condition present in E0/E1 pipeline register (lane 0)
    br <= E0/E1.resultb_0;
```

Listing 5.4: ID hazard on link register

```
IF ((EO/E1.target_0_r = WRITE_G) OR (EO/E1.target_0_r = WRITE_G_B)
     AND (E0/E1.dest_0 = decode.address_r2_0 = address_lr)
   -- lr present in E0/E1 pipeline register (lane 0)
   lr <= E0/E1.result_0;
ELSIF ((ID/E0.target_0_r = WRITE_G) OR (ID/E0.target_0_r = WRITE_G_B)
      AND (ID/E0.dest_0_r = decode.address_r2_0 = address_lr)
   -- lr present in ID/E0 pipeline register (lane 0)
   lr <= ID/E0.result_0;
```

Listing 5.5: ID hazard on stackpointer

```
IF ((EO/E1.target_0_r = WRITE_G) OR (EO/E1.target_0_r = WRITE_G_B)
     AND (E0/E1.dest_0 = decode.address_r1_0 = address_sp)
   -- sp present in E0/E1 pipeline register (lane 0)
   sp <= E0/E1.result_0;
ELSIF ((ID/E0.target_0_r = WRITE_G) OR (ID/E0.target_0_r = WRITE_G_B)
     AND (ID/E0.dest_0_r = decode.address_r1_0 = address_sp)
   -- sp present in ID/E0 pipeline register (lane 0)
   sp <= ID/E0.result_0;
```

Furthermore, as the effective address calculation is pre-computed within the decode stage, there can be a data hazard present when an earlier stage produces the base address register. From the above discussion and with the help of the depicted figures we derive the following set of forwarding equations. These equations describe the input values for the first and second input operands for a Functional Unit within the first pipeline lane. Thus the second set of equations is the same for the second operand within pipeline lane 0. These equations are the basis for implementing the forwarding logic that runs in concurrently with the VEX processors pipeline. Their are other implementation issues that concern the exception behaviour of the VEX processor. However, we have considered these to be out of our project's scope.

5.5.2 Register Files

One important improvement for the processor's pipeline is to design a Register File with *Explicit Register File Forwarding*. As already mentioned in section 6.4.2.1, their is a possibility to trade-off area for performance in the RF design. In the same way as the above forwarding equations can detect data hazards, we have incorporated forwarding equations within the design of the RF. Their purpose is to detect RAW hazards between data values that are written back to the register file and those that are read in the same clock-cycle as operands of the Decode Unit. An illustration that governs the forwarding logic within the RF is presented in Listing 5.6. In general we see that the WriteBack stage writes the same registers that the ID stage reads. The forwarding logic controls how the first operand from the E0 stage gets the bypassed value from the WriteBack stage in lane 0 when the read and write addresses are the same. By extending the RF design with the next set of forwarding equations and by adjusting the RF to process its control and data input signals on the rising clock edge, we were able to come up with an Explicit forwarding design for the RF. Listing 5.6 presents the set of logic equations that controls how values are forwarded for the first operand of a FU in lane 0 of the pipeline. Note that a similar set of equations has to be added for each of the other 3 lanes of the processor. In addition, their are other design features that must be adjusted in the design of the RF.

Listing 5.6: Register File Explicit Forwarding

```
...
process(write_en_0, write_en_1, write_en_2, write_en_3, address_w_0,
        address_w_1, address_w_2, address_w_3, address_r1_0,
        data_in_0, data_in_1, data_in_2, data_in_3)
  begin

   if (write_en_0 ='1' and address_w_0 = address_r1_0) then
       data_out1_0 <= data_in_0;
   elsif (write_en_1 ='1' and address_w_1 = address_r1_0) then
       data_out1_0 <= data_in_1;
   elsif (write_en_2 ='1' and address_w_2 = address_r1_0) then
       data_out1_0 <= data_in_2;
   elsif (write_en_3 ='1' and address_w_3 = address_r1_0) then
       data_out1_0 <= data_in_3; -
   else
       data_out1_0 <= registers(conv_integer(address_r1_0));
   end if;
 end process;
```

5.5.3 The Data path

The E0 and E1 stages of the VEX micro-architecture compose the datapath of the processor's pipeline. In section 6.4.3, and 6.4.4, we mentioned that there are two types of functional units within these pipeline stages. Their design may provide improvements in the micro-architecture of the pipeline. These functional units are:

1. ALU Unit
2. MUL Unit

We have taken care to design these Functional Units in such a way that the synthesized hardware becomes simple and good enough for our VEX design. However, we have not utilized a customized approach, which of course will result in the best possible design for both functional units. In fact in the datapath both designs strongly rely on a behavioural hardware description. For the ALU this means that the synthesized logic will consist of multiple sub-componets for each ALU operations. This means that the resulting ALU performs well enough, but will be large in area and is power hungry. We could re-design the ALU by utilizing a structural hardware description. Such a description allows the design to be mapped to hardware with the smallest possible area such that there is still enough parallell hardware to perform every ALU operation fast enough as not to penalize the clock-cycle time.

For the MUL Units, the situation is more demanding as the critical path of the design runs through this part of the first execution stage. The current design utilizes the 18×18 multipliers of the target ViRTEX 6 FPGA chip. Basically, the VEX ISA requires two set of 16×32 MUL Units to perform signed or unsigned multiplications. In fact, the VHDL compiler will map the multiplication operations of the MULT Units to the multipliers that are provided by the FPGA. The result of each multiplication operation is ready after a single clock cycle. This means that computing the 16 partial-products and adding them to form the final product will create a lower bound for the clock-cycle time. By devising a customized design in which the MULT Units are designed by following the **Modified Baugh-Wooley** approach, one can divide the hardware for every multiplication operations over the E0 and E1 pipeline stages [16]. Indeed, this means that during the first execution stage all 16 partial products are computed in a parallel way and that their results are stored in the **E0/E1** pipeline register. Subsequently, during the second execution stage all partial products are added in a fast and parallel way by using a 4-level Carry Save Adder tree or an other fast adder. With such an approach, we can strive to design better hardware than what is currently provided by the target FPGA, as we have the freedom to divide the multiplier hardware over two pipeline stages. This will improve the clock-cycle time of the VEX processor, as their is less logic in the E0 stage that signals have to pass through.

5.6 Conclusion

In this chapter we elaborated on all of the hardware design aspects that were required in order to come up with a new and better design for the VEX processor. As is usually the case in processor design, we started with a study and an discussion of the VEX ISA, because this would provide us with the important hardware/software interface for designing the VEX processor.

We investigated the execution model that is assumed by the VEX ISA. This has provided us with a general overview of how the target hardware must execute the VEX operations during run-time. From this investigation we have taken the architectural

state into consideration, such that we could have a precise idea of how architectural components as the register files and functional units had to be designed in order to fit the processor's micro-architecture. An architectural classification of the ρ-VEX processor was presented as a mean of enabling us to make better and wiser decisions when composing the VEX processor's micro-architecture. Furthermore, this gave us a general strategy how we had to implement the processor's pipeline.

We presented a novel organization of the VEX processor as to provide a foundation for the lower-level architectural details like the branch architecture and the number of execution stages. The former has been designed such that it does not offset the assumption that the VEX compiler makes when it schedules branch operations. We have reduced the miss-penalty add the cost of more complicated forwarding logic. Moreover, we have discussed regulating the instruction format such that it can serve as a basis for the VEX pipeline's micro-architecture. There we saw that almost all operation types had to be re-designed in order to overcome all of the encoding problems of the multi-cycle implementation. Here after, we looked at the micro-architecture of the VEX processor.

We zoomed into each of the five pipeline stages, such that a clear description could be presented of the entire design of the processor's pipeline. By doing so, we gave a clear motivation behind each of the design decisions that were made as the final design cristilized. In the last section, we focused on three set of architectural improvements that maximize the delivered performance of the VEX processor. In the Chapter 6, we look at the implementation details that come into existence as we move on to implement the required hardware for the VEX processor.

6 Hardware Implementation

In the previous chapter, we discussed the hardware design of the VEX processor. In this chapter, we present how we have implemented its design such that a novel softcore VLIW processor came into existence. We have called this new implementation the VEX processor. This chapter has the following organization. Section 6.1 present a brief discussion of the implementation strategy that we applied in order to implement the processor's design. Section 6.2 looks at the implementation details of the five-stage pipeline of the VEX processor. Section 6.3 looks at the implementation details of Data memory and Instruction memory. Section 6.4 discusses additional peripherals that make up our VEX processor. Finally, Section 6.5 describes how we have composed the VEX processor system from the constituting peripherals and the VEX core.

6.1 Implementation Strategy

We utilized a **bottum-up** implementation strategy during the process of programming our VEX processor design in VHDL [13]. In general we programmed every pipeline stage as a separate architectural component. This gave us the ability to test each pipeline stage as a separated design entity such as to reveal bugs early in the implementation phase. We started by implementing the Fetch stage of the processor's pipeline. For each stage we have strived to come up with an efficient implementation that is still simple enough to form a particular microarchitectural component of the 4-issue pipeline. By thinking in terms of simplicity the synthesized logic became simple enough such that the resulting clock frequency would become better then the previous multi-cycle implementation [17]. After the Fetch stage was verified, we took enough design time to concentrate on the Decode stage, since the design phase showed that this was the most complex stage of the VEX processor's pipeline. Furthermore, we have mimiced each stages as they would function in the final processor by verifying its implementation as a separate implementation entity. This enabled us to verify the basic functionality of each pipeline stage. To do so each stage was tested with inputs signals from a testbench that passed data and control like they would come from a preceding stage. In the same way we proceeded with the other stages. Once every stage was implemented and verified, we connected five of them such that the VEX pipeline came into existence. Subsequently, we proceeded on a system level to connect other **I/O** devices such that the VEX processor System got finalized.

We have ensured that our implementation is parametric and that it is synthesized from the ground up, by hand coding all VHDL that specified the different architectural components and by inheriting the parametric characteristics from the ρ-VEX design [4]. First, we have utilized through an interface file that specified the parametric size of the different architectural components in the design, by utilising *generics*. The name of this file is called **r-vex_pkg**. Second, we programmed the processors design by sticking to VHDL semantics that can be synthesized an by keeping the target FPGA in mind [17]. Furthermore, the complete implementation was realized in a modular fashion such that future improved microarchitectural components could be plugged into design with little effort.

6.2 The VEX Processor

The VEX processor consits of five major implementation components as the design is based on a five-stage pipeline. The **E0** and **E1** stages are combined within the **execute** stage wrapper such as to deal with the implementation complexity of the processor's datapath. This holds the **E0/E1** pipeline register together with the ALU, MUL and Load/Store functional units. A top level implementation of the processor's core is depicted in Figure 6.1.
Furthermore, each pipeline stage contains a pipeline register to hold intermediate results for the next pipeline stage. Like the execute wrapper, the four processor components that are depicted in Figure 6.1, contain the following pipeline registers in their implementation:

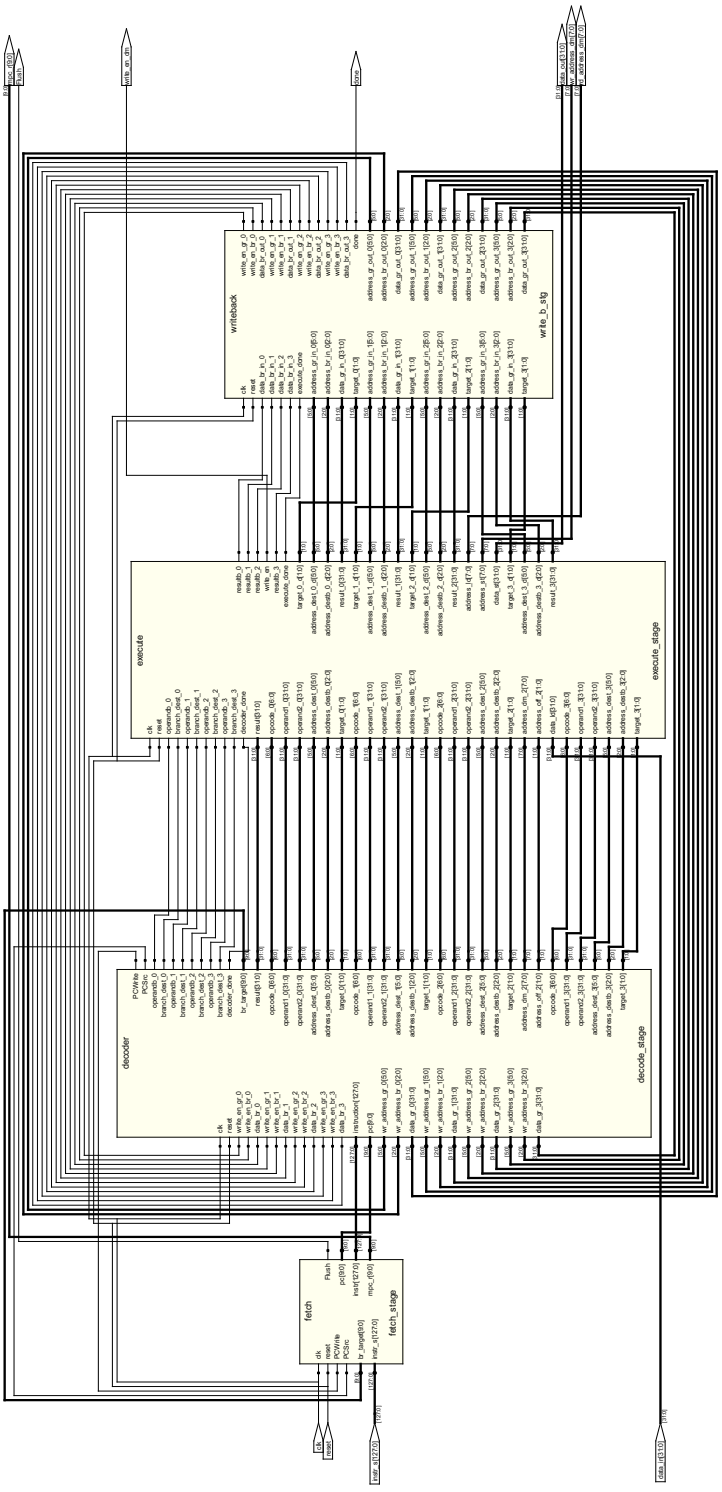

Figure 6.1: The VEX Core

1. Fetch: IF/ID
2. Decoder: ID/E0
3. Execute 0: E0/E1
4. WriteBack: E1/WB

The Fetch stage synchronously supplies on a system level the instruction address for Instruction memory such that it is able to read a 128-bits VLIW instruction. The execute stage is connected to the Data memory for reading or writing data. It supplies the control and data signals for the Data memory from the Load/Store Unit and containes part of the **E0/E1** pipeline register from the datapath. The **Decoder** forms the Decode stage wrapper. The design of this stage contains the Branch Unit, Register files, and the Decode unit as is specified by the designed micro-architecture. The WriteBack stage contains all the logic required to comitt the functional results to the Register files. We look at the specific details of the implementation of each of the five pipeline stages.

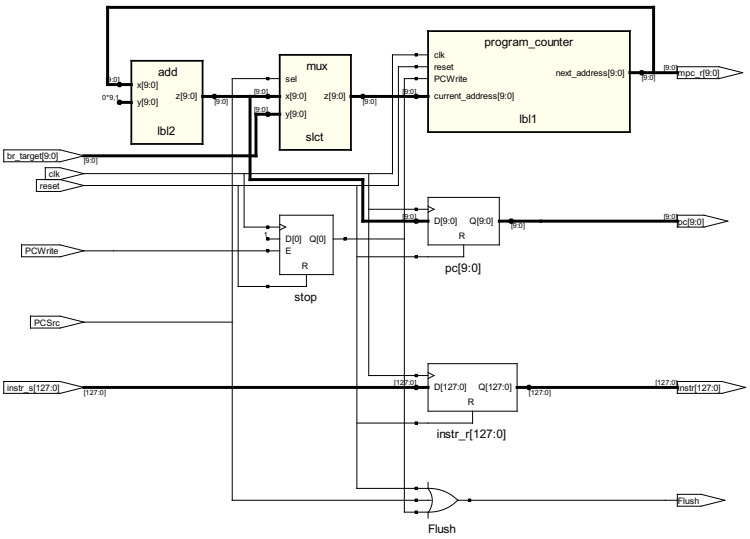

Figure 6.2: Fetch stage design

6.2.1 Fetch Stage

We implemented the design of the Fetch stage by utilizing a hybrid approach. The micro-architecture of this stage was specified in a structural way, while the lower-level specifics of the microarchitectural components where programmed by utilizing behavioral VHDL code. In Section 5.4.1, we saw that the design of Fetch stage is comprised of three architectural components:

1. Address Generation Unit

2. Instruction Memory
3. Synchronization Unit

Instruction memory is generated in the form of a program Read Only Memory (ROM) by the new **ROM-generation** as part of the toolchain. To read it we need to synchronously increment the instruction address at the beginning of each clock-cycle. The Address Generation Unit consits of an address that is incremented, a program counter register and a mux. By default this unit always increments the instructions address such that we can sequentially step through the Instruction memory. However, when the Decode stage, in particular the Branch unit, has decoded a branch syllable it directly asserts the **sel** signal of the mux such that the computed branch target is selected as the next input for the program counter. At the same time the Synchronization Unit asserts the **PCSrc** signal, which flushes the instruction that was fetched in the previous clock-cycle. This is required as we have to discard the instruction that gets affected by the single cycle mispenalty. In the next clock-cycle the Fetch stage supplies the instruction at the branch target. This explains why the Fetch stage has an adder, multiplexer and a program-counter register connected as is depicted in Figure 6.2. The adder together with the program counter register perform the address incrementation in the Address Generation Unit.
All clock-cycles that proceed the clock-cycle when the last instruction is fetched, must keep signalling that the processor's pipeline is finished executing instructions. This is accomplished with the **stop** and **PCWrite** signals. The latter stays high during the last processing cycle, while the former will remain asserted until the processor pipeline is reset from outside. The last component from the Fetch stage is the **IF/ID** pipeline register. It is programmed using behavioural VHDL. Like any other pipeline register it serves to buffer data and control information that the Fetch stage computed for the Decode stage.

6.2.2 Decode Stage

The Decode stage contains the General-purpose and Banch Register Files. Moreover, these storage containers are connected to the Decode Unit *and* to the Branch Unit, such that the instruction that were temporally stored in the **ID/E0** pipeline register can be decoded during the second clock-cycle.

Register Files

Lets first look at a naive partial implementation of the Register File for lane 0. This is depicted in Figure 6.3. In principle this Register File is build out of a stack of 64 registers which in turn are composed of banks of 32 D flip-flops. By implementing the Register File in this way, the two read ports from any lane consist out of two parallel multiplexer that have their inputs connected to the output of each register. The selection signal of the multiplexer is controlled by the register's read address input signal. When a certain address is asserted, we select the target registers as the output utilizing the multiplexer. The write port of the Registe File is implemented as a register identifier of which the address is first sent to a 6 to 1 decoder. This is required as from the 64 possible register,

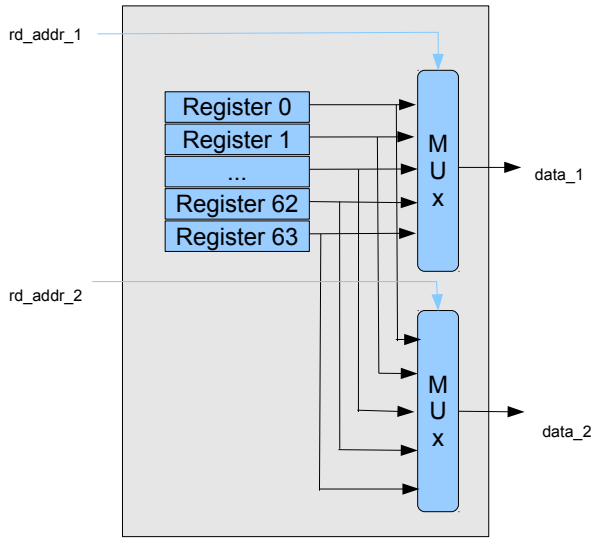

Figure 6.3: Simple read port implementation

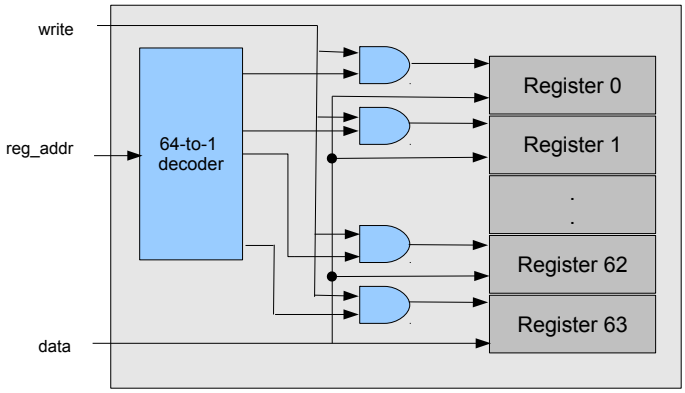

Figure 6.4: RF write port

only one register forms the lane target. The output of this decoder is anded with the **wr_en_** signal which asserts the enable signal of the target register. This is depicted in Figure 6.4. Note that we still must utlize the clock signal to determine when to write data to a target register. However, this is not shown in the figure. The VHDL statements for all four pipelanes are depicted in Listing 6.1. Because we do not want to experience any additional delay cycles, we read the outputs combinatorially such that the operand value appears right away on the outputs of the Register File. The first process statement enables the VHDL compiler to infer that writes to the register file have to be scheduled

synchronously on the rising clock edge. Moreover, writes are allowed to all registers except to register zero.

Listing 6.1: Write port partial implementation Register File

```
architecture behavioural of gr_registers is
  ...
  begin

  process(clk, reset)
    begin

      if(reset ='1') then
         registers(0 to (GP_DEPTH - 1))<=(others => x"00000000");
      elsif(rising_edge(clk)) then -- writing data to registers

        if (write_en_0 = '1' and not(address_w_0 = "000000")) then
            registers(conv_integer(address_w_0)) <= data_in_0;
        end if;

        if (write_en_1 = '1' and not(address_w_1 = "000000")) then
            registers(conv_integer(address_w_1)) <= data_in_1;
        end if;

        if (write_en_2 = '1' and not(address_w_2 = "000000")) then
            registers(conv_integer(address_w_2)) <= data_in_2;
        end if;

        if (write_en_3 = '1' and not(address_w_3 = "000000")) then
            registers(conv_integer(address_w_3)) <= data_in_3;
        end if;

      end if;

    end process;
    ...
```

The next process statement presented in Listing 6.2, controls **bypassing** the input data of the RF to the input operand ports from the ID/E0 pipeline register. When the read addres of the decode stage and the write addres in the WriteBack stage are the same, the WriteBack stage attempts to commit the data to the target register. This register is also read by the Decode unit. This is the condition that actually holds when a RAW hazard exists between the Decode stage and the WriteBack stage. Therefore, we have two compare these address and we must set the enable port of the target register accordingly. This means that in hardware, the Registe File is augmented with additional Explicite Forwarding Logic equations as was explained in Section 5.4.2 of the previous chapter. Furthermore, their are 3 more similar process statements required to control writes of the data outputs for the other data_out_ signals in the three other lanes. These statements have not been included in the code listing for the briefness sake. From these equation the VHDL compiler will infer the required write, read and forwarding logic in the Register file.

Listing 6.2: Explicite Forwarding Implementation RF

```
...
process(write_en_0, write_en_1, write_en_2, write_en_3,
        data_in_0, data_in_1, data_in_2, data_in_3,
        address_w_0, address_w_1, address_w_2, address_w_3, address_r1_0)
begin
        if (write_en_0 ='1' and address_w_0 = address_r1_0) then
            data_out1_0 <= data_in_0;
        elsif (write_en_1 ='1' and address_w_1 = address_r1_0) then
            data_out1_0 <= data_in_1;
        elsif (write_en_2 ='1' and address_w_2 = address_r1_0) then
            data_out1_0 <= data_in_2;
        elsif (write_en_3 ='1' and address_w_3 = address_r1_0) then
            data_out1_0 <= data_in_3;
        else
            data_out1_0 <= registers(conv_integer(address_r1_0));
        end if;

end process;
...
```

Moreover by trading-off performance for area, we have devised a Register file design that incorporates Implicit RF forwarding. The idea in doing so, is to postpone writes to the RF as long as possible and to read the data from the registers as the last possible action during the second cycle. This effectively deals with the RAW data hazards, as we force the writes to precede the read operations. This implies that we will always read the updated or written value. The changes required to the Register File in VHDL are the following:

1. change rising_edge to **falling_edge** in the sensitivity list of the first process statement

2. replace the second process statement by asynchronous reads that will always take place in the RF

The first adjustment is easy. The second one is depicted in Listing 6.3. The above discussion only concerns the implementation of the General-Purpose Register File. However, as we saw in section 5.4.2, the same timing principles can be applied to the Banch Register file. The difference is the number of read ports, the data size of the registers and the number of registers. The branch Register file has a single read port and contains a single bit of storage in each register. Furthermore, the stack of registers is much smaller as their are only eight of them.

Listing 6.3: Combinatorial reads for Register File

```
 ...
 end process write;
  ...
      data_out1_0 <= registers(conv_integer(address_r1_0));
      data_out2_0 <= registers(conv_integer(address_r2_0));
```

```
        data_out1_1 <= registers(conv_integer(address_r1_1));
        data_out2_1 <= registers(conv_integer(address_r2_1));
        data_out1_2 <= registers(conv_integer(address_r1_2));
        data_out2_2 <= registers(conv_integer(address_r2_2));
        data_out1_3 <= registers(conv_integer(address_r1_3));
        data_out2_3 <= registers(conv_integer(address_r2_3));
...
```

Decode Unit

The Decode unit orchistrates all micro-operations in the second pipeline stage. Its implementation is specified by utilizing strict combinatorial logic. In essence this stage reads a 128 bit instruction from the Fetch stage, splits it into four 32-bit syllables and decodes each syllable in parallel. The goal is to generate the control- and data signals for the Branch Unit, Register File and the **ID/E0** pipeline register. Although the design of the Decode Unit was programmed with 479 lines of VHDL code, this section presents a brief discussion of its implementation. The implementation of Decode Unit performs the following tasks:

1. Extract opcode field from the split syllables
2. Set the addresses to read the register values
3. Send the operands to the Branch Unit and **ID/E0** register
4. Set the lane targets and the branch_dest_ flag
5. Set destination addresses for WriteBack stage
6. Pre-compute effective memory address for **E0** stage

All of the above tasks are performed concurrently by this micro-architecture. As we devised a regular and consistent syllable layout, we know in advance where the register fields for reading the source registers reside. Recall that this layout was presented in Figure 5.3 of Section 5.3. Moreover, from the mapping that the instruction layout specifies for each operation type, any particular lane of the processor knows what type of operations to expect. We can determine what kind of operation we are dealing with by extracting the opcode from the syllable field. This field serves the following purposes:

1. It is utilized for further internal decoding work.
2. It is sent to the **E0** stage as a FU's opcode.

The first task is to extract from each syllable the first and the second source fields for the GP registers and to set the read addresses for the GP Register file accordingly. Depending on the operations type, the source branch register field may be situated in two location of the syllable. Conditional Branch operations and the **MFB** ALU operation have this field located in the bit positions 4 upto 2 from the syllable. For **ADDCG/DIVS** or **SLCT/SLCTF** operations this field resides in the first 3 bits of the opcode field, see in

target_i	WriteBack target	Mnemonic
00	NOP	WRITE_NOP
01	GP	WRITE_G
10	BR	WRITE_B
11	GP and BR	WRITE_G_B

Table 6.1: Targets in WriteBack stage

Section 5.3.1. In the same way, we check the opcodes in each of the four pipe-lanes for certain operations *or* operation classes. Based on these checks, we set the read fields of the source operands as is needed.

Sending operands is the second task that the Decode unit performs concurrently. When doing so, it sets the source addresses of the registers according to the operation or the operation class that is being decoded. For branch operations, we extract the branch offset bits for the Branch Unit in lane 0. These bits are located in a fixed position. Consequently this extraction is simple. Furthermore, we set the first operand value to the first data input that comes from the General-Purpose Register File. The second operand is complexer as it can be a register- or an immediate value. In the case that the operation's *immediate switch* indicates the presence of a short immediate, we must first sign extent the immediate field before passing it via the **ID/E0** pipeline register as the second source operand in one of the four pipe-lanes. Otherwise, we will perform incorrect 32-bit arithmetic in the ALU or MUL Units from the E0 stage. When the operation is a long immediate, this action is not required. However, we have to correctly read-out and concatenate the remaining 22 bits of the long immediate value. When a lane has no immediate present, the Decode Unit sets operand number 2 to the second data output of the Register File. Moreover, we have to set the outputs of the branch data to the output of the branch operands.

Depending on the extracted opcodes, we encode the write target of an operation when it leaves the WriteBack stage. Our architecture was designed as a strict Load/Store architecture. This means that last pipeline stage commits its results to either the GP or BR Register files, and that NOP operations do not change the the processor's architectural state. Consequently their are only four possible encodings of the target signal in our VEX implementation comparred to the eight possibilities of the ρ-VEX implementation. The alternative targets that the Decode Unit can encode while it decodes an operation, are given in Table 6.1.

Based on the opcodes or the operation's class, we check and decide how to set the target fields when performing the fourth task. As an example, we can determine whether a logic operations will write a branch register in the commit stage and that it has the last bit of its opcode field asserted. In this case, we set the target field to WRITE_B. Moreover, we need to set the *branch_dest_* signal to indicate that for this operation the first bit of the ALU represents a boolean condition. Other values for the target field are set by doing similar checks. For lane 0 List 6.4 includes a code excerpt of the Decode Unit. It describes how this is actually implemented for the target_0. As was explained at the beginning, we see the checks that are required to set the branch as a destination

signal for logical operations or for the **MFB** operation.

Listing 6.4: Encoding WriteBack targets

```
...
if (std_match(opcode_0_i, ALU_ADDCG) or
    std_match(opcode_0_i, ALU_DIVS)) then
  -- ALU operation with BR and GP dest
  target_0 <= WRITE_G_B;
  branch_dest_0 <= '0';
elsif (std_match(opcode_0_i, CTRL_OP)) then
  -- CTRL operation
  branch_dest_0 <= '0';
  if (std_match(opcode_0_i, CTRL_RETURN) or
      std_match(opcode_0_i, CTRL_RFI) or
      std_match(opcode_0_i, CTRL_CALL) or
      std_match(opcode_0_i, CTRL_ICALL)) then
    target_0 <= WRITE_G;
  else
    target_0 <= WRITE_NOP;
  end if;
elsif (std_match(opcode_0_i, LOGIC_OP) or
       std_match(opcode_0_i, ALU_MFB)) then
  -- ALU operation with BR or GP dest
  if (syllable_0(25) = '1') then
    -- br as destination
    branch_dest_0 <= '1';
    target_0 <= WRITE_B;
  else
    -- gr as destination
    branch_dest_0 <= '0';
    target_0 <= WRITE_G;
  end if;
elsif (std_match(opcode_0_i, NOP)) then
  -- no operation
  branch_dest_0 <= '0';
  target_0 <= WRITE_NOP;
elsif (std_match(opcode_0_i, STOP)) then
  -- STOP operation
  branch_dest_0 <= '0';
  target_0 <= WRITE_NOP;
else -- normal ALU or MUL operation
  branch_dest_0 <= '0';
  target_0 <= WRITE_G;
end if;
...
```

The task of setting the destination addresses for the WriteBack stage is performed by checking the operation's class or its specifics. Referring to the syllable layout in Figure 5.3, the destination of all operation classes are situated in the same location of a syllable. Hence, based on the the syllables opcode, decoding the general-purpose register's write address can be performed in a straight forward way. Logical- and **MFB**

operations target a branch register of which the identifier is located in the first part of the general-purpose destination field, syllable bit position 19 upto 17. For **ADDCG/-DIVS** operations this field is located in bit position 4 upto 2 and by utilizing such checks the Decode Unit sets the Branch destination field. Moreover, as we have increased the branch offset field we can hard-wire the destination addresses for the control operations of the **CALL/ICALL** and **RETURN/RFI** operations. The last micro-architectural task that the Decode unit performs is that it pre-computes the effective memory address in advance for the E0 stage. We utilize the first operands source field and set it to read-out the base registers contents. The result in lane 2 is placed on the first data_in_2 source. Subsequently, in the same clock-cycle we add the extracted memory offset from syllable 2 in order to compute the effective memory address. Furthermore, the Load/-Store Unit needs the byte offset for accessing data bytes of data types that are smaller than a word. This signal is derived from the first two bits of the final effective memory address.

Branch Unit

The implementation of the Branch Unit (BU) requires combinatorial hardware. This is due to the asynchronous nature of its design. Let us look back at the interface of this architectural component. For convenience, we re-illustrate the encompassing architecture in Figure 6.5. The pc signal represents the value of the next instruction address (program counter) and is received from the **IF/ID** pipeline register during the second clock-cycle. Furthermore, as syllable_0 in the first lane was partially decoded, the Branch Unit receives **opcode_0** and the branch offset signals (off) from the Decode Unit. Subsequently, based on the opcode that is decoded in parallel by the Decode Unit, the following set of micro-operations is executed in the Branch Unit:

- ***Unconditional jumps***: When the BU and Decode Unit decode the **GOTO** operation the next micro-operations are scheduled. The BU unit adds the branch offset to the pc value to compute the branch target. It also asserts the selection signal of the Address Generation Unit such that this selects the target address as the next pc value. For **IGOTO** operation which implements indirect jumps, we assert the selection signal for the Address Generation Unit. However, while the BU decodes the **IGOTO** operation in parallel with the Decode Unit, it puts the link register value on the lr input for the BU. In turn the BU utilizes the link register value as the desired branch target. In this way, we are able to jump to the instruction address that was stored in the link register.

- ***Function calls***: For **CALL** operations a number of concurrent micro-operations are scheduled. First, the BU computes the branch target by adding the pc value to the branch offset. This represents the address of the callee site. The micro-architecture must be able to return after the callee finishes execution. Therefore in the same cycle we copy the value of the pc to the *result* output. This then represents the return address value that the Decode unit concurrently directs such as to latter commit the *result* to the link register in the Write Back stage. This is realized by

encoding the **target_0** as WRITE_G. For **ICALL** the BU performs similar micro-operations when it computes the return address. However, the branch target is directly computed as the link register value that is received from the Decode Unit.

- ***Indirect jumps***: For the **RETURN/RFI** operations, the BU copies the link register value (lr) from the Decode Unit to the branch target. This provides support to jump back to the caller site (link address). At the same time the BU sets the result to the value that micro-architecture has due to popping the callee's stack frame. This is required since we must remove the callee's **activation record** from the function call stack. Moreover, during this cycle, we assert the **PCSrc** signal such as to select the branch target in the Address Generation Unit. The **PCSrc** and **PCWrite** signals were omitted from Figure 6.5 to keep it clear.

- ***Conditional branches***: In VEX these are formed by the **BR/BRF** operations. The BU performs a branch based on a **condition** that was stored in the branch register. This register was already read during the second clock-cycle in the Decode Unit. In the event of a **BR/BRF** operation the branch condition **directly** controls the selection signal of the multiplexer in the Address Generation Unit for promoting the branch target. This target is computed by adding the offset value to the **pc**, subsequently the result is send in the same clock-cycle to the **Address Generation Unit** of the Fetch stage.

The preceding discussion somewhat simplified the branch offset calculation for *pc-relative* control operations. We note that as the program counter originates from the **IF/ID** pipeline register, by the time it reaches the Branch Unit it was already incremented. This was already explained in Section 6.4.2.2. Therefore, the branch architecture must somehow correct this without delaying the Branch micro-architecture. By making a hardware/software tradeoff we are able to do so. Focusing on the branch offset computation, the left side of (6.1) suggests that we can also subtract the unit in the least position (*ulp*) from the offset bits to compensate for the program counter incrementation. By being clever, we have performed this in the assembler for all *pc-relative* offset computation. This has allowed us not to penalize the Branch architecture with an extra addition delay. Furthermore, for function call operations, we conveniently have the return address to our disposal in the form of the **pc** value in the Decode stage.

$$br\_target = offset + pc - 1 = (offset - 1) + pc \tag{6.1}$$

The last micro-operation that the Branch Unit schedules is performed when it decoding the **stop** syllable. For the Address Generation Unit, it asserts the **PCWrite**- and the *decoder_done* signals. The former triggers the Synchronization Unit to flag that the Fetch stage must remain finished. The latter flags the E0 stage to stop Execution during the third clock-cycle. To complete the implementation of the Decode stage, we utilize the **decoder** wrapper. This connects the three constituting micro-architectural components as is depicted in Figure 6.5. Note that each of the signals that ends with an underscore actually is a **vector** of four signals. This is true as the pipeline has four lanes due to the multiple-issue micro-architecture of the VEX processor.

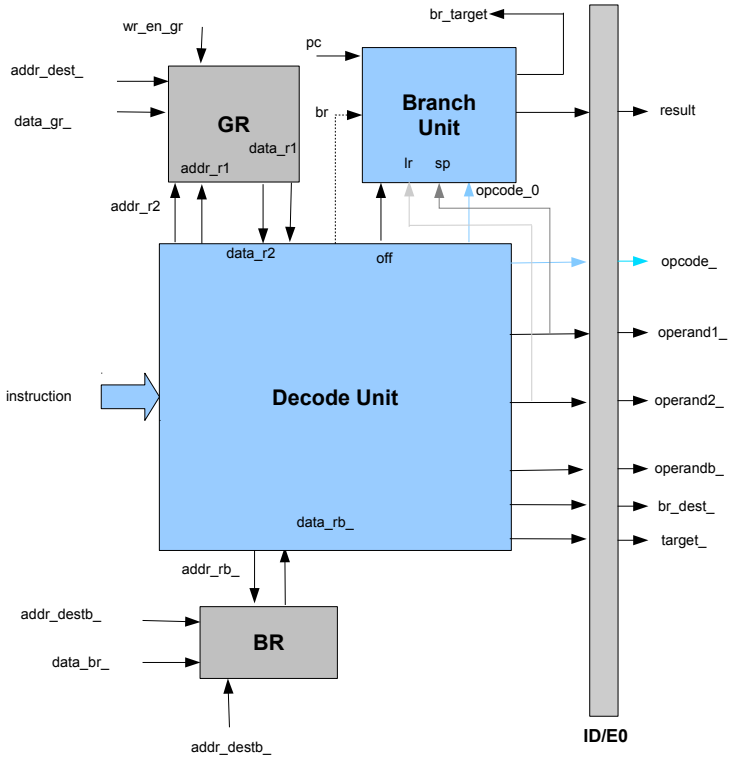

Figure 6.5: Implementation Decode stage

6.2.3 Execute 0 Stage

We implemented the E0 stage by programming the micro-architectural components of its design in VHDL. To explain how this stage works, we focus on the top level issues from the implementation that is depicted in Figure 6.6.

The four parallel ALUs were re-designed and programmed in behavioural VHDL. Moreover, for our datapath it is required that their implementation is asynchronous up to the point were they specify parts of the **E0/E1** pipeline register. This requires synchronous reads of the ALU outputs. Compared to the multi-cycle implementation, we have added the **MFB** operation to the set of ALU operations. This operations reads the carry input from the ALU and flags this on the carry output of the ALU. We corrected the implementation of the **SUB** operation. Additionally, other operations like **ADDCG** and **DIVS** were implemented in a more efficient way. The combinatorial part of the ALU is implemented in a rather straight forward way. The ALU unit receives the **opcode_** signals, decodes this to an opcode value and depending on this value, it executes the required logic operations on its input operands. The result appears without delay on the input of the **E0/E1** pipeline register. For example for a **ADD** operations the ALU executes the adder hardware on the input operands to produce the sum as the result. This is then read by the **E0/E1** pipeline register portion that passes through the ALU

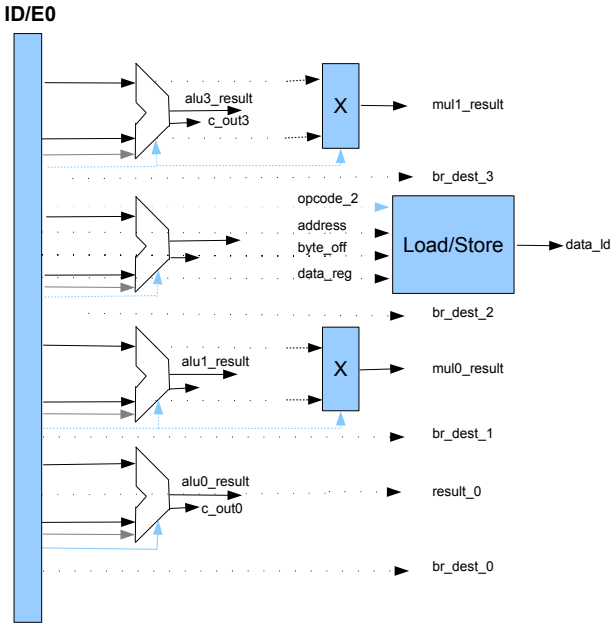

Figure 6.6: Execute 0 stage

units. Other operations are implemented in a similar way. Like the ALU Units, the MUL Units were also re-designed and implementation flaws were corrected or replaced with more efficient hardware. The multi-cycle implementation decided to clip the products that fell outside the 32-bits range. Although the reason behind this make sense in a DSP architecture, for our designed VLIW processor this violates the VEX ISA, as it does not specify any clipping semantics for its multiplications arithmetic [9]. We have chosen not to include the encoded **target**_ vectors and the register destination addresses that are destined for the WriteBack stage in Figure 6.6.

For memory operations the Load/Store unit acts as the memory controller of Data memory. We come up with a better design of the datapath, as the previous design would not suite our VEX pipeline. Executing a pre-load operations requires that the Load/Store unit sets the Data memory address to the effective address. This was computed and send to the **ID/E0** pipeline register during the second clock-cycle. In the third clock-cycle, our new Data memory design is able to deliver the desired data. For store operations this Unit takes the byte offset to choose in which position it places the target byte. Subsequently, it places the target byte at the required position in the pre-loaded data, asserts the memory enable signal, and writes the updated data at the effective address that comes from **E0/E1** pipeline register. In the fourth pipeline stage the commit destination target is Data Memory. This means that store operation have a (pipeline) latency of four clock-cycles. For load operations a similar set of micro-operations is scheduled except that their is no enable signal generated in the fourth pipeline stage.

Moreover, the effective address is placed on the **address_r1** input of Data memory in the third pipeline stage. Subsequently, in the fourth pipeline stage the loaded data is sent from the **data_out1** output to the **data_2reg** output of the Load/Store unit. Here it will be written to the **E1/WB** pipeline register. For loading data types smaller than a 32-bit word (**LDB**), after the data is loaded an intervening byte selection for the desired data size is scheduled. In the E0 stage, we see that each lane of the processor may produce only one of the two possible results. For the next stage we will see that a pre-selection of the actual result is needed, because at run-time the pipe-lane is not able to know which of its two functional units really produces a result.

6.2.4 Execute 1 Stage

The implementation of the second Execution stage was realized within the **execute** stage wrapper. When we designed the datapath, we saw that this stage may be viewed as the pre-selection stage of the WriteBack stage. For every pipeline their are two set of multiplexer to control the following actions:

- **branch result**: The selection between a logical ALU result and a carry output is controlled by the **branch_dest_** signal. The input to the multiplexer are the **alu_result_s(0)** and the **alu0_cout_s** signals.
- **FU result**: Selection between an ALU result and the result of an other functional unit in the same lane. This is controlled by the opcode class. The input to the multiplexer are the **alu_result_s** and a functional unit result like the **mul1_result_s** signal.

The next code excerpt illustrates part of the behavioural VHDL code for the selection process. The VHDL compiler will infer two multiplexers. Pre-selection in the other lanes of the pipeline is governed by similar VHDL statements.

Listing 6.5: Result pre-selection in pipe-lane 0

```
...
        if (std_match(opcode_0_r, CTRL_OP)) then
            result_0 <= result_r
        else
            result_0 <= alu0_result_s;
        end if;

        if (branch_dest_0_r = '1') then
            resultb_0 <= alu0_result_s(0);
        else
            resultb_0 <= alu0_cout_s;
        end if
...
```

Moreover, the **E1** stage controls the load phase of the Load/Store Unit. It forms the memory controller for reading Data memory. The loaded data appears on the **data_2_reg** output of the Load/Store Unit. Additionally, this stage passes on the encoded **target_** vectors and the register destination addresses that will be utilized in the WriteBack stage.

6.2.5 WriteBack Stage

The WriteBack stage is the commit stage of the processor's pipeline. From the encoded **target_** signal this stage sets the general-purpose and branch register destination addresses, the write enable signals and the data outputs for the target registers in the Decode stage. The WriteBack stage is implemented with 4 parallel banks of five D flip-flops, as our pipeline has 4 execution lanes. Furthermore, we have omitted the clock and reset signals that are routed to each D flip-flop. From the encoding of the WriteBack targets in Table 6.1, it is easy to see that the enable signal to write a general-purpose or a branch register can be derived from the bit position in the **target_i** value. The benefits of the pre-selection from the E1 stage are apparent when we look at Figure 6.7 and imagine how its design will change with additional selection logic from the E1 stage.

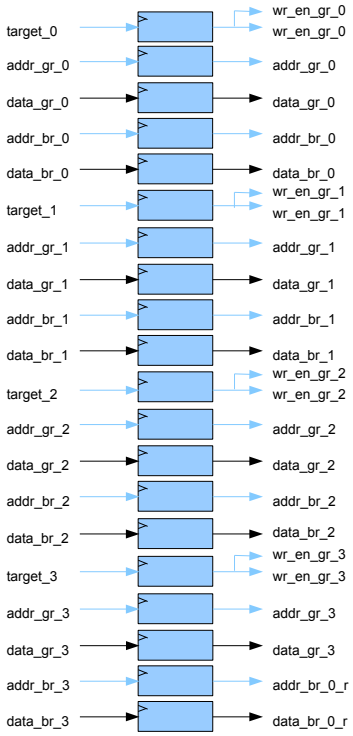

Figure 6.7: Implementation WriteBack stage

6.3 Memories

This section discusses the implementation details of the Instruction memory and Data memory. Both of these architectural components interface efficiently to our VEX micro-architecture. We have implemented each differently compared to the multi-cycle imple-

mentation [4]. Data memory has no latency for load or store operations in its implementation. Otherwise, we would have to stall the pipeline during the third and fourth clock-cycle when executing memory operations. This slows us down as it eventually increases the number of stall clock-cycles for the VEX processor's pipeline. The implementation of Instruction memory was adjusted such that it could be parameterized with the rest of the processors implementation.

6.3.1 Instruction Memory

The program ROM (prom) required to store the instruction contents for our processor is automatically generated by the last part of the re-designed toolchain. This module is called elf2vhd and transforms the received instruction binary into VHDL. This allows simulation and synthesis of the VEX processor.

We carried on the parametrization of the design to our prom implementation such that we can change the size of the instruction memory. The **ADDR_WIDTH** generic controls the number of bits in the program counter register and the input instruction address of the prom. As we explain in the next chapter, we extended the back-end of the VEX compiler such that it emits assembly that is essential for the generating the program ROM. Among others, excessive comments are filtered out such as to pass only useful information to the re-designed assembler. This extension of the back-end was accomplished with a parser that allows to shift some pre-processing semantics tasks from the assembler to the parser. Additionally, the parser does some trivial address computations of memory offsets and evaluates all types of expressions within the assembly file. The details of the toolchain are discussed in Chapter 7.

6.3.2 Data Memory

We implemented Data memory by keeping into account that it must be able to deal with the RAW hazards as was explained in Section 5.4.6. There we saw that an efficient pipelined design can not afford to have an extra load latency of one clock-cycle or a two cycles latency for store operations. By keeping the target FPGA in mind, we have come up with a different Block RAM implementation without architectural visible read or write latencies. Our implementation makes use of a **Read-through** technique to satisfy the memory requirements [17]. Writes to memory are scheduled synchronously, while we concurrently pre-store the read address in a D flip-flop. To deal with the load latency, we read out the target value in a asynchronous manner, but we make use of the synchronous read address that was pre-stored in the D flip-flop in the previous clock-cycle. By implementing Data memory in this way, we are sure to deal with all of the issues that were mentioned. As the UART requires additional reads of Data memory, the read and data output values are duplicated in the implementation. When data memory contains only one read ports, it will create a structural hazard for the pipeline of the VEX processor.

6.4 I/O Devices

After all pipeline stages were implemented and verified within the processors pipeline, we connected the five pipeline stages such as to form the VEX processor. In the VEX implementation this is realized within the **r-vex** processor wrapper. On a system level the VEX processor is augmented with a number of peripherals, such that we are able to realize the VEX processor system. This set of peripherals is composed out of the following I/O devices:

- Two Clock dividers.
- The Cycle-counter.
- The UART module.

The clock dividers are utilized to scale down the FPGA's clock frequency to a realization frequency at which the VEX processor can run. The cycle-counter is utilized for benchmarking applications in hardware that execute on the VEX processor. The UART is required to read-out the contents of Data memory in order to verify whether the correct result were computed by the VEX processor. In this section, we briefly discuss the most important implementation details of these I/O devices. It should be noted that these devices were already present in the multi-cycle implementation. However, we mention the changes that were performed [18, 19].

6.4.1 Clock Dividers

We added an extra clock divider to scale down the FPGA clock frequency from 200 Mhz to 100 MHz. Our design had a 200 MHz Virtex 6 FPGA chip as its implementation target. Furthermore, the UART can run at 50 MHz. Therefore, the first clock-divider scales down the on chip 200 MHz clock frequency to 100 MHz, which feeds the VEX clock signal. The next clock divider receives the 100 MHz as the input clock frequency and in turn scales this down to a 50 MHz clock signal to drive the clock signal of the UART module. This implementation is depicted in Figure 6.8.

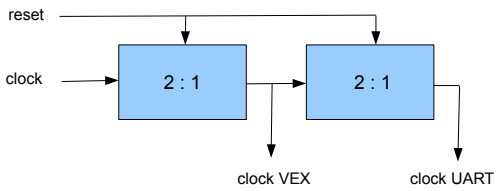

Figure 6.8: Cascading clock dividers

6.4.2 Cycle-counter

We moved the cycle-counter from the fetch stage to the processor system. We argued that this level was a more appropriate place for a cycle-counter as it is an output device. Furthermore, its implementation was improved to deal with synthesis issues when realizing the design on the target FPGA. Our improved implementation utilizes the rising_edge VHDL clock attributes, and has a better description of the incrementation hardware [18].

6.4.3 UART

The implementation of the UART module was kept the same as in the multi-cycle implementation. As a minor change we utilized the attribute rising_edge within its implementation. This module was developed by Thijs van As and provides debugging services for our VEX processor System [4]. The UART interface transmits data via the RS-232 protocal from Data memory, at a rate of 115200 bps in hexadecimal format. Additionally, it sends the contents of the cycle-counter device as the benchmark result for the executed application. The UART is invoked after the WriteBack stage asserts the done signal in the VEX pipeline. The transmitted data is sent in the American Standard Code for Information Interchange (ASCII). It can be displayed on a Personal Computer using a serial terminal program like *minicom*.

6.5 The VEX System

After the three I/O devices were tested and verified to function correctly, we integrated them with the VEX processor such that the complete VEX processor system came into existence. This marked the last implementation phase of our processor. Furthermore, it gave us the ability to test and verify the complete processor system as everything was connected together. We utilized a structural VHDL hardware description to support our modular bottum-up implementation strategy [13]. This allowed us to connect each architectural component of the processor system as was specified in the processor's design. The VEX processor System is depicted in Figure 6.9.

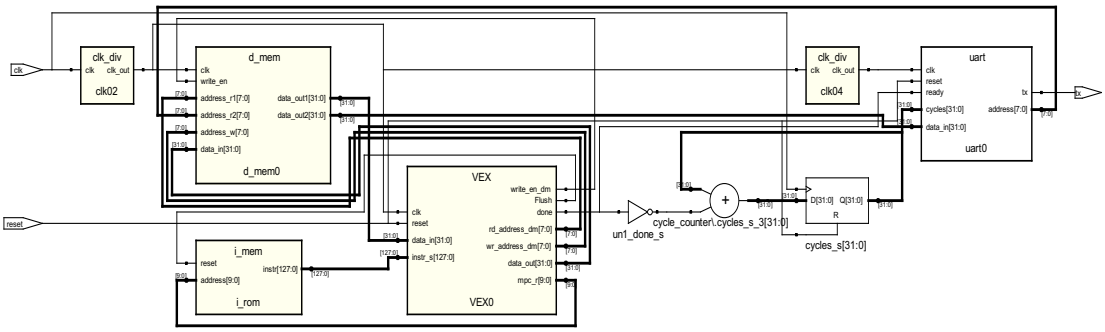

Figure 6.9: VEX processor System

6.6 Conclusion

In this chapter, we presented the discussion of how we have implemented the design of the VEX processor. During the implementation, we followed a bottum-up strategy. With this strategy, each architectural component was programmed in VHDL. Subsequently, all architectural components were separately tested and were finally connected to form a separate pipeline stage. Each pipeline stage was verified, such that we could confirm that it functioned according to the requirements of the VEX pipeline. After this step, we have connected the five pipeline stages to from the VEX processor. We have discussed how Data- and Instruction memory were implemented such as to satisfy the requirements of our VEX pipeline. At this point, the implementation of the basic processor was finished. However, no processor can function as a stand-alone device and requires I/O devices to supply or externally store data. We have presented how such devices were adjusted in our implementation such that the VEX processor system could be composed out of the VEX processor, its Data memory and Instruction memory and the I/O devices. After we implemented the processor's hardware, we move now on to the toolchain's design. In the next chapter, we see that a considerable design effort has been invested since three months of the design time were spent for the re-design of the required toolchain. In essence, what was needed was an automation of the generation of the program ROM for the VEX processor.

7 The Toolchain

The last two chapters discussed the design and implementation of the VEX processor. In this chapter, we present how we re-design and implement the Development Framework that was inherited from the ρ-VEX project. The result of this phase of the project was a toolchain that could be utilized for the instruction ROM generation of the VEX processor. In order to come up with a toolchain which is mature and useful enough for this processor, we have extended several parts of the toolchain and programmed each in C.

The organization of this chapter is as follows. In Section 7.1, we present an overview of the developed toolchain. Furthermore, this section motivates why the back-end of the VEX compiler had to be extended, and discusses why a re-design of the toolchain was required. Section 7.2 discusses how we implement extending the back-end. Section 7.3 discusses the design and implementation of the assembler which is the heart of the toolchain. Section 7.4 presents the Application Binary Interface (ABI), the low level software conventions that are required in order to understand the purpose of the linker. Section 7.5 presents our collaboration to incorporate a linker in the toolchain. Section 7.6 looks at the implementation of an additional module to realize the generation of an instruction ROM from a linked executable file. Section 7.7 concludes this chapter by establishing what we have achieved and learned in the process of developing the toolchain.

7.1 Overview

Before we start to discuss the low level details of the design and implementation of the toolchain, it is required to look at the composing software modules. We stress the fact that the inherited Development framework gave us the advantage not to start designing the toolchain from scratch. We boot-strapped the assembler from the inherited Development frame work and utilized the GNU utilities in order to develop a better toolchain. Keeping in mind, that the inherited toolchain is composed out of the following software modules:

1. **VEX compiler**: As part of the inherited Development framework, this processes a C source file and produces a VEX assembly file. This compiler is only accessible as a binary file. Consequently, there are no adjustment possible to it.

2. **ROM-generator**: This is an integral part of the inherited assembler which directly produces an instruction ROM.

This simple structure of the Development Framework was satisfactorily enough to generate the instruction ROM for the ρ-VEX co-processor. However, our sole purpose is to present the designer with an automated toolchain for the generation of the instruction memory for the VEX processor. Firstly, an embedded programmer should be able to easily port and debug his target application. Secondly, he should be able to evaluate the application's footprint on the FPGA and its performance when it runs on the VEX processor.

We debugged the inherited assembler such that we could determine its usability for the new toolchain. Moreover, during this tedious phase, we came to the conclusion that a re-design and extention of the toolchain was inevitable. We needed to do this, because we have experienced the following list of problems when attempting to utilize the inherited toolchain:

- **Instruction addressing** violated the VEX ISA definition: When generating instruction labels there were no distinction made between PC-relative instruction addresses and absolute addresses. This was as a result of the particular micro-architectural decision to implement all addressing modes with absolute addresses.

- Incorrect **indirect branch operations** implementation: The **IGOTO/ICALL** operations were not programmed correctly. It appeared that the generated binaries from the syllables were not instructing the Decode stage to schedule the required micro-operations in the execution stage and WriteBack pipeline stage.

- **Long immediate** instructions were implemented improperly: The implementation in the assembler suggested that a short immediate value was 10 bits, although its size was defined as 9 bits by the VEX ISA. The problem was traced back to the comparison that was performed with a value of 1024, which is really an unsigned 10 bits value. This seemed to be the only cause of this problem, but actually the Long immediate issues stemmed from hardware design flaws in the Decode unit of the multi-cycle implementation.

- **Pseudo-operations** were only implemented for **MOV** operations. For some strange reasons, the designer had chosen to implement this pseudo-operation as a native VEX operation. Other pseudo-operations like move from link register (**MFL**) were not recognized by the ROM-generator. This indicated that the assembler was not finished.

- **Assembly directives** were not yet implemented: These are keywords that tell the assembler how to translate an assembly file, but do not produce machine instructions. Examples of assembler directives are the **.comm** and **equals**. The assembler was not processing these directives. Furthermore, statements that were considered as comments because they carried no additional information for generating an instruction, were decreasing the readability of the generated instruction ROM.

In order to effectively deal with the above problems, we designed the structure for the toolchain as is depicted in Figure 7.1. This structure constitutes an internal design flow that is composed out of five software modules with the purpose of generating a correct instruction ROM for the pipelined VEX processor. The next sections discuss how we developed each of these software modules.

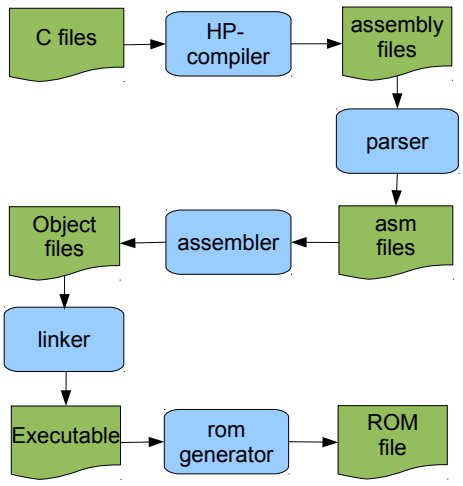

Figure 7.1: Overview toolchain

7.2 Back-end

The back-end of the VEX compiler was extended with a parser that was generated by Bison. Bison is a Yacc look-alike tool that is provided by GNU. Like Yacc, Bison works as a bottom-up parser generator but it emits ANSI C code. Bottom-up parsing considers collection of syntactic alternatives simultaneously and matches the right syntactic alternative at the latest possible moment [20]. After appending the parser to our back-end, the following steps were implemented:

- Print to cleanasm file instead of to screen.
- Extent the the rules for parsing a mop argument list.
- Adjust the common directive rules for declarations.

The first adjustment was performed in the file, main.c of the *asm* parser. The parser additionally has to filter the assembly output of the back-end. The assembly file from the compiler back-end has to be filtered in order to pass only essential assembly statements to the assembler. Furthermore, it evaluates simple expressions and prints the processed assembly file. Trying to process and filter everything in the assembler would result in a design that becomes too complex as this would also require to perform low-level parsing tasks in the assembler. Consequently, in every parsers file the output that was normally printed to the console is re-directed to an output assembly file. For this reason, we added a file pointer at the beginning of main.c. This pointer is depicted in Listing 7.1.

Listing 7.1: Writing to file

```
FILE *yycleanasm;  /* print output to file */
```

The second adjustment was to generate *mop arguments* such that the parser could print the desired terminals. The generated assembly file from the compiler had to adhere to the assembly semantics from VEX. This required adjusting three rules of which two are shown in Listing 7.2. Most of the other adjustments were performed in the parsers grammar.y file. Note the presence of a dot in front of the first grammar rule. The adjustment involved the second an third alternatives. When the back-end wants to output a "clean" assembly file it is required to print the assignment symbol as a string. For every mop_arglist, we print a comma as a separation symbol between the two arguments. This is illustrated in the last grammar rule.

Listing 7.2: Adjusting grammar rules

```
.mop_arglist:
        %prec ARGS
    |   mop_arglist '=' {fprintf(yycleanasm, "=");} mop_arglist
    |   mop_arglist '=' {fprintf(yycleanasm, "=");} %prec ARGS
    |   mop_arglist;

mop_arglist:
        mop_arg
    |   mop_arglist ',' {fprintf(yycleanasm, ",");} mop_arg;
```

The third adjustment is discussed next. When the declaration of an objects was performed without initializing its attributes, the parser seemed to forget this declaration. This means that no assembly code was generated to allocate and initialize the declared object. An example is when an array gets declared without assigning its elements initial values. After an extensive debug campain, we found that the compiler was outputting the reserved assembly directives. However, after the code passed through the asm parser no assembly code was emitted by the parser. By carefully inspecting the parser files, we concluded that the grammar rule was present in the parse for such declarations, but that these rules did not print to the output file. Furthermore, we needed to print a data string in front of the object's declaration together with its initializations. This was necessary, otherwise the assembler could not distinguish between a declaration with initializations and one without initializations. To solve this issue, we extended the grammar rules in bss section of **grammar.y** file with several fprintf calls that additionally print the .comm directives. Moreover, these statements had to print the name of the declared object and its attributes.

Thus far, we have presented the important adjustments that were required in the parser after it was generated by Bison.

7.3 Assembler

In the previous section, we discussed how the parser pre-processes the assembly file, such that it becomes ready for the assembler. An assembler is a program that translates the symbolic version of instructions into their binary representation. The translation process consists of two major steps. The first step is to find the mapping such that the relationship between symbolic names and their addresses are known at compile time. The second step translates each assembly instruction by combining the numeric equivalents of opcodes, register specifiers and labels into a legal instruction [21]. An assembler that works in this way is called a **two-pass** assembler.

Unlike its predecessor we have simplified the control flow of this assembler in both passes in order to reduce the complexity of its source code. The previous assembler first generated the instruction and would then re-scan the generated assembly file in search of unresolved labels. Forward references always resulted in unresolved labels. Subsequently, the assembler would check if such labels were present. Then, during the second pass it would utilize the recorded information in the symbol table to fill in the unresolved labels. We argued that it is simpler to proceed in the reverse order. Therefore, during the first pass, we only determine the addresses of labels and store this in the symbol table without generating instructions. Then, generating instructions is done during the second pass by utilizing the mapping information in the **symbol table**.

We did not have the time to port an Operating System (OS) to the VEX processor. This is why there are additional tasks that the toolchain must perform apart from generating the instruction binaries for the target VEX processor. There is a simple **loader** incorporated into the toolchain that is programmed in assembly and gets assembled and linked with the final binary. Details about this are presented in Section 8.5. The processing heart of the assembler mainly consists out of the following parts, see the file rasm.c.

1. Data initialization
2. Label determination
3. Instruction generation

The first part parses and generates the VEX instructions in a binary file for initializing Data memory. The second part takes care of mapping all the identified labels to Data- and Instruction memory addresses in the symbol table. The third part parses and translates all assembly instructions to their binary representation. Additionally, the assembler fills in other sections in the assembled object file as this has to conform to the Unix object file format. Lets first consider this format as is given in Figure 7.2. The UNIX File Format for object files contains six distinct sections:

- **Object file header**: this specifies the size and position of other sections of the file.
- **Text segment**: this contains the machine language code for routines in the source file. These may be un-executable as they may contain unresolved references.
- **Data segment**: this contains a binary representation of all data in the file. Data may be incomplete due to unresolved references to labels in other object files.
- **Relocation information**: identifies instructions and data that depend on **absolute addresses**. These references must change if parts of the program is moved in memory.
- **Symbol table**: which is a table that matches names of labels in the source file with the addresses of the memory words that the instructions occupy. Furthermore, it lists the unresolved references.
- **Debug information**: this contains a compact description of how the source was compiled such that a debugger may know the instruction which addresses correspond to file lines, such that it can print the data structures in a readable form.

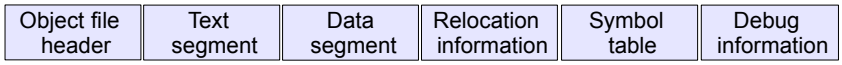

Figure 7.2: Unix assembly format

An object file typically contains two types of labels that are utilized for referencing instructions of procedures or data. A **label** is **external** if the labeled object can be referenced from files other than the one in which it is defined. A label is **local** if the referenced objects can be used only in the file in which it is defined. Subroutines and global variables require external references as they are referenced from many other files

in the program. Local labels hide names that should not be visible in other files. An example is a static function that may only be invoked from procedures within the same file. Other examples are compiler-generated names like instruction labels. This frees the compiler from the burden of having to define different names in every source file for instruction labels.

Since the assembler processes each file in isolation it only knows the addresses of local labels. Later, we will see that the assembler depends on another tool called the linker to combine a collection of object files and libraries into an executable file by resolving external labels. Subsequently, the assembler assists the linker by providing lists of labels with unresolved references. In the next section, we look at more specific details about these file sections. Furthermore, we discuss the reasons for extending the assembler with additional modules.

7.3.1 Structure

The assembler has a flat directory structure. All its header files and source files are contained within a directory called src. During the re-design, we removed some source files but have also added a number of source files and header files from the inherited Development framework. The files that printed the header and the contents of the ROM file namely vhdl.c and vhdl.h were removed. Part of their purpose is realized in the ROM-generator called elf2vhd.c. The elf2vhd module implements the generation of the instruction ROM from the final linked object file. The re-designed assembler is composed out of the files which are depicted in Listing 7.3. Of course every header file is accompanied by its equally named source file.

Listing 7.3: Constituting modules assembler

```
#include "expr.h"
#include "rasm.h"
#include "syllable.h"
#include "util.h"
#include "instr_array.h"
#include "data_array.h"
#include "relocations.h"
#include "symbols.h"
```

The assembler was extended with the file expr.c that defines functions to deals with the evaluation of expressions. Every immediate field of an instruction may be the result of an expression. In its simple form this is a constant. However, when accessing fields of structs this expression has to be evaluated at run-time to a value. The functions in expression.c implement what is required to do so. The files rasm.h, syllable.h and until.h where already present in the ROM-generator. We extended these files with additional functions to assist the functions of rasm.c. There were two additional header files that had to be included in the file rasm.c, in order to deal with interfacing the assembler to the linker's source files. We mention them here to illustrate the complete set of header files that are now included in rasm.c, see Listing 7.4. We discuss their purpose in Section 8.5. Subsequently, we look at how the assembler was re-designed by focussing on the main functions that consitute its processing hart.

Listing 7.4: Interfacing assembler with linker

```
#include <bfd.h>
#include <elf.h>
```

7.3.2 Data Initialization

In essence data initialization generates the binary code to initialize Data memory with the values of the data types that are defined in the data sections of an assembly file. We will first look at the semantics and syntax of this initialization. This is illustrated by the assembly code excerpt example in Listing 7.5. Afterwards, we discuss the algorithms that generate the required VEX binaries.

Listing 7.5: Declaration and initialization of array

```
data item: kar (global)
     init: 0 (size: 4)
     init: 1 (size: 4)
```

The data keyword is an assembly directive that tells the assembler to start operating in the data section of the Unix file. The item keyword proceeds the objects name. In this case its a globally declared array called kar with two integer elements. We mentioned that the first pass determines the mapping between labels and addresses. Moreover, it has to parse the preceding information and then it calls the initialize_data() function. This function emits the required VEX operations to actually store the array in Data memory and consists out of three parts:

1. **Prologue**: moves the immediate and its address to Data memory
2. **Body**: makes the store syllable that is required for the data type
3. **Epilogue**: stores the immediate and updates the data pointers

The *prologue* makes a move syllable that copies the immediate as a 32-bit constant to a temporary register. We treat the immediate as a long immediate, because by doing so, we do not have to handle constants that are less than a 9 bits 2's complement values differently. Subsequently, we set the syllable function type for both syllables to ALU and copy the move syllable to the instruction buffer. The function type allows us to sort the syllable to the correct issue-slot in the instruction. A similar sequence of statements is executed for moving the address to the base register. The immediate represents the target Data memory address, which is an absolute addresses. This means that we need to add a relocation entry such that the linker can have the bookeeping information available during linking separate object files. The pseudo-code in Listing 7.6 is a concise explanation of the prologue.

Listing 7.6: Prologue of Data initialization

```
clear instruction buffers
make syllable to move immediate to temporary register
```

```
make syllable to move address to sp register
set both syllables function type to ALU
add relocation entry for address
print move instruction
increment address
if no forwarding
   generate 2 nops
```

In the *body* of this function the first thing to do is to clear the instruction buffers. The data type that we can expect is either a byte, a short or a word. For each alternative data type, we make the actual store instruction. In the procedure's body we proceed by setting the syllables function type to memory (MEM). Here after, we store the instruction in a buffer, such that it is ready to be written in the instruction section of the object file. Now, we are ready to proceed with the last part of the data_initialization() function. This is presented in the epilogue part and is illustrated in Listing 7.7.

Listing 7.7: Body of data initialization

```
clear instruction buffers

switch size
   case 1:              /* byte */
      make syllable stb
      break;
   case 2:              /* short */
      make syllable sth
      break;
   case 4:              /* word */
      make syllable stw
      break;

set syllable func_type to MEM
store syllable in instruction buffer
print instruction
```

The *epilogue* represents the last part that increments the program counter address for the instruction ROM. This enables reading the next instruction during run-time. Moreover, the data memory address and the data pointer of the linker are both increased with the size of the previously stored data type. Finally, we return the next empty location of Data memory as an offset value to the caller **determine_label()**. These actions are performed during the first pass and are shown in Listing 7.8.

Listing 7.8: Epilogue data initialization

```
increment address
increase offset pointer by size
increase addend pointer by size
return offset
```

7.3.3 First Pass

During the first pass the assembler determines the mapping between labels and their addresses. For instruction labels the addresses reside in the Instruction memory, where as for data labels they reside in Data memory. One complication is that the absolute addresses will be relocated when we link this file with other files and change the pre-mapped location of the current assembly file. The function determine_ label() implements the first pass. It establishes the mapping information and stores this in the symbol table. The first formal argument of this function is a pointer to an input assembly FILE. The second formal argument is a pointer to a BFD library file descriptor. These are the main mechanism provided by the GNU project intended for a portable manipulation of object files in the different formats [27]. The implementation of the function determine_label() is explained by means of the flow chart that is depicted in Figure 7.3. We focus on the high level implementation issues through out this explanation.

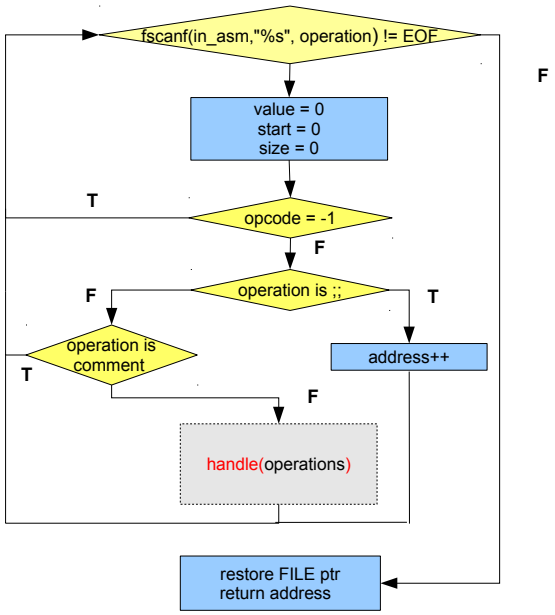

Figure 7.3: Flow chart first pass

The the while-loop is the driving loop behind the first pass that controls parsing and processing the different assembly operations and directives. We read the assembly file line by line until we reach the End Of File symbol (EOF). Within the main loop, we start by setting the variables that are used to store the value, size and name of the data types that are defined in the assembly file. Subsequently, we convert the operation to an opcode and check if the opcode is valid. If this is the case, we read the next file line. Otherwise, we decide if we have to read the symbol that marks the end of the current instruction. In the event that this is the case, we increment the address of instruction memory. Otherwise, we check if we have to read any assembly comment. When this

is true we simply *skip* it and read the next line at the beginning of the while-loop. Otherwise, we have to start handling lots of assembly operations.

For the purpose of our explanation, we have abstracted these action in the **handle(operation)** procedure. In this part of the first pass there are many assembler directives and data types processed. Consequently, we zoom into one such that we can get get an idea how they are implemented.

When we are able to parse the init: directive from the operation variable, we try to scan two values which are the expected size and the value of the data type. When this succeeds, we call the initialize_data() function and pass it the scanned information. We already discussed how this function is able to perform its task in the first part of this section. Recall that it returns the address of the current empty Data memory location as the offset value. Consequently, we can append the object at the end of the data section of the binary file that was initialized to zero. This is accomplished in the append_bss_data() function. When we are unable to scan these two values, we try to read how many bytes we need to skip next. The first pass can perform this, as the semantics allow us to read exactly the *skip* amount of bytes. Subsequently, we increase the offset variable with size bytes at the end of the data section of the binary file. Listing 7.9 is a code excerpt that shows the programming details concerning this discussion.

Listing 7.9: Parsing data labels

```
...
else if (strcmp(operation, "init:") == 0)
{
        if ((sscanf(operands, "%d(size:%d)", &value, &size) == 2))
        {   /* generate code to initialize data */
            offset = initialize_data(abfd, value, size, &addr,
                                     offset, last_name, &addend);
            append_bss_data(data, (uint8_t *)&value, size);
        }
        else if ((sscanf(operands, "skip%d", &size) == 1))
        {   /* increment offset with size bytes */
            uint8_t temp = 0;
            offset += size;
            append_bss_data(data, &temp, size);
        }
}
...
```

A similar set of statements is executed in the rest of the alternative branches within the handle(operations) part. However, as there are many possible assembly directives and labels emitted by the extended back-end which the assembler may come across during the first pass, there are many deep chains of **if else-if** statements required for determining the mapping of labels and addresses. Moreover, within each parsed alternative, there is lots of code programmed to store all mapping operations in the symbol table. When keeping track of the corresponding addresses, we perform this by correctly parsing the labels name with the aid of the lexeme in the vicinity of the data initializations or instruction labels.

7.3.4 Second Pass

During the second pass the assembler parses the file again, but now it is able to utilize the recorded mapping information from labels in the symbol table as it is composing the syllables. From the syllables it composes an instruction that is then stored in an instruction buffer. This is implemented in the function **assemble()**. As there were already instructions generated to initialize Data memory, we pass this function the program counter's offset address, such that the program instructions follow after the data initialization instructions. As the exact programming details of this function resulted in 648 lines of C code, we only discuss the high-level structure of it. We refer the reader to the source code in rasm.c for the implementation details.

Assembling Syllables

This function is composed out two major parts. The first pass takes care of initializing the syllable buffers and sets the VEX instructions so as to follow after the data initialization. Then, we enter the main loop that reads each lines of the assembly file, until we arrive at the End Of File marker (EOF). The body of the while-loop may be divided in a *pre-process* part that prepares the composition of a syllable from its composing fields and an inner-control part that flags labels, comments and the stop symbol.

The second pass process the lines in the assembly file and breaks each into its component pieces. These pieces are known as *lexeme*. Usually they are individual words, numbers and punctuation. As an example the code excerpt in Listing 7.10 illustrates a line with 7 component pieces. From left to right these are: the cluster identifier c0, the opcode add, the register specifier $r0.1, the assignment operator =, the register identifier $r0.1, a comma and the number −32.

Listing 7.10: Illustration of lexeme

```
c0      add $r0.1 = $r0.1 , -32
```

The main processing part of the loop-body parses the lexeme on the line and determines the type of the current syllable. From this information, it writes each field of the syllable with the values that were read or processed within determined alternative. Once we determined the actual alternative, we continue to parse the desired information from the lexeme that is currently being read on the line in the assembly file. Within a parsed alternative, sub-alternatives may be present due to the way that we have encoded our syllables. For example ALU syllables may be of type arithmetic or logical. Moreover, both may have immediate values encoded in their fields. This adds up to 4 possible sub-alternatives. We experienced that dealing with the sub-alternatives made programming the parser cumbersome, as the number of sub-alternatives that the assembler has to deal with are plenty and their are even exceptional cases. Once we assessed the correct sub-alternative, the assembler can generate the syllables from the fields that were set in the parsing process. Subsequently, it checks the label and comment flags which control the required actions. This is performed to stay in phase with the assembler directives and the macros. At the end of the while-loop's body, we lower the flags for labels, comment and relocation information. In this way, we process all the lexeme on each

line of the file and assemble the fields of the syllables, until we finally produce all the VEX instructions. Listing 7.11 gives a more concise description of the pseudo-code that describes the previous discussion.

Listing 7.11: Second pass pseudo-code

```
determine_label(bfd, offset)
{
        initialize syllable buffers
        instruction follow after data

        while operation that was read from this line is not EOF
        {
              get cluster number and continue
              fetch line (operands), strip comments and spaces
              clear fields and flags
              handle directives and labels
              set the function type for each syllable in instruction

              when current operation is not comment nor label nor stop
                determine syllable type and set fields
                otherwise print Error
                assemble syllables from their type and fields
                check if label or comment flag were detected
                    ...
                check if label was detected
                    ...
                lower label, comment and relocation flags
        }
        return 0;
}
```

Pseudo-operation	Native VEX operations	Semantics
MTB b = {s2,im}	CMPNE b = $r0.0, {s2,im}	MOVE s2 or im to b
MFB t = b1	SLCTF t = b1, $r0.0, 1	MOVE b1 to t
MFB b1 = b2	??	MOVE b2 to b1
MTL lr = s	ADD $r.63 = $r0.0, s	MOVE s to lr
MFL t = lr	ADD t = $r0.0, $r.63	MOVE lr to t
MOV t = s2, im	ADD t = $r0.0, s2, im	MOVE s2 or im to t

Table 7.1: Pseudo-operations

Pseudo-operations

Besides the native operations, the VEX compiler also emits a number of ALU operations that are only documented in the parser's header file vexopc.h [15]. We have considered

these operations as *pseudo-operations*. This means that they do not have to be implemented in hardware. For the toolchain's software one advantage was that we did not have to utilize the extra opcode values to encode these operations in the assembler. In hardware this implied that we do not need to extent the set of operations that the decoder handles. Consequently, this means the second stage becomes less complex. The idea was to implement each pseudo-operation with a set of native VEX operations. This is accomplished by enabling the assembler to detect the opcode strings of the pseudo-operation, overload this with native operations, and set the VEX operands to the pre-defined values that are presented in Table 7.1. This table presents the pseudo-operations along with the necessary native VEX operations and the required predefined values to obtain the desired semantics of a pseudo-operation. By setting the operands to these values, we are able to receive the right semantics from the pseudo-operations. One exception was the **MFB** operation, as this operation has a branch register as the source register. However, it is still unclear whether this operation can have a branch register as its write destination. For this reason, we decided to implement it as a native operation in the Decode Unit such that both register types are possible as destination register.

The bit level operations to set or test bits e.g. **SBIT/SBITF** and **TBIT/TBITF**, that were utilized in the older versions of the VEX compiler are obsolete as from version 3.41. These operations are somewhat redundant, as they can be generated with **AND** and **OR** operations and a long immediate as the second source operand.

7.4 Application Binary Interface

The low level interface between the application and the Operating System (OS) that controls the hardware resources of a processor system is called the Application Binary Interface (ABI). In this interface the applications code and the processors run-time system coexist and visibly connect [9]. Part of the ABI constraints of the application's code, is that the run-time system is able to safely cooperate with it. When doing so, it includes items of a larger scope such as functions, complation units, libraries and even complete programs. The ABI may also be considered to be part of the run-time system. On the otherhand, one must keep in mind that the ABI affects **all stages** of the development **toolchain**, especially compilation and and linking [9]. The ABI can be considered as part of the OS. Although porting a complete OS to our VEX processor was out of this project's scope, we have devised additional modules that provide the low-level services that an OS can offer the run-time system of the VEX processor.

It was clear from the beginning that we did not have the time or the resources to port an OS, like for example (μ-Clinux), to our VEX processor. We reflected on this dilemma and came up with the idea to run at least the bear essentials that an OS would provide in terms of a linker and loader. Because in the last stages of implementing the assembler, it became clear that for most application even a mature assembler would not be able to generate the necessary instruction ROM, when the application's source code consited out of multiple source and header files. This may be understood by keeping in mind that an assembler by definition processes each file from the application in isolation. So it can not know the addresses of external labels. Having performed its part, the assembler

depends on the linker to combine a collection of object files and libraries into a larger executable. An important constraint is that our assembler was programmed in a way that simplified interfacing to a linker. So after the assembler satisfied its part of the ABI, the linker was interfaced as the missing module of the toolchain. There were two file formats that the assembler must obey when it produces the required executables. Both are important since we did not want to program our linker from scratch. These formats are:

1. **E**xecutible and **L**inkable **F**format: elf
2. **B**inary **F**ile **D**descriptor Format: bfd

The first format is a common format for executables, object codes and shared libraries. Unlike many proprietary formats it is flexible and extensible as it is not tied to a particular processors architecture. This allowed us to tailor it for the purpose of producing the specific binaries for the VEX processor. Moreover, in a future project it would be possible to adopt it such that a desired OS could be ported. Thus, producing our binaries in this format would ensure that the linker can read its contents in the next module of the toolchain.

The second format is from the GNU's project. It is a software library that is able to abstract different file formats like elf and coff, such that these fromats can be easily maninupulated in software. As an application opens a target object file a file pointer to a structure called bfd is returned. By convention this pointer is called a BFD and instances of it within the application (rasm.c) are called abfd. The BFD format utilizes the abstraction that an object file is structured as is presented in Figure 7.2 of Section 8.3. The operations are applied to the object file as methods to the BFD object. Mapping the object file to the BFD format is accomplished in the header file bfd.h by a set of macro's that all start with bfd_. When we write the BFD object it is converted to the chosen elf format. The programming details that explain how the linker and the assembler interface are implemented are found in the file rasm.c.

7.5 Linker

A linker enables separate compilation allowing an application to be programmed in separate C files. Every source file or header file contains a logical related collection of functions and data structures that together form a module as a larger set of files. This set is called a program or an application. From a compiler perspective, we obtain the advantage of not having to compile the entire application when a file is changed. We have not implemented the linker alone, instead its design and implementation were realized as a collaboration within the ERA project. Our VEX processor and the new toolchain served this project's goals. Therefore, we will not discuss the programming aspects of the linker here. Instead we only exemplify the most important design principles.

One of the claims that were made in the ρ-VEX project is that it had produced a Development framework. Consequently, when the implementation of the assembler finalized, we started investigating if their was any linker present. As this was not the case, we came up with the idea for linking VEX binaries by porting the GNU's binutils

and their associated tools, bfd and gld. Furthermore, we could bootstrap the linker that was provided by ST-micro distribution. However, we decided to utilize samples of other provided architectures for porting the bfd and ld tools. After porting, the ld tool provides the linker facilities for our toolchain. After this step, we placed intervening calls in the assembler such that the linker could interfact to it. The linker merges the separate object files that were produced by the assembler. There are several tasks that a linker may perform in a standard Application Programmer Interface (API). Firstly, the linker should get access to libraries to perform its task. Secondly, it is required to indicate to the linker which libraries it must access when it links the binaries to an executable. Furthermore, once this is performed the linker must do the following:

1. Search through the program's libraries to find referenced routines.
2. Asses the memory location that a module occupies and **relocate** its instructions by adjusting the absolute addresses.
3. Resolve references across the constituting files of the target application.

The first task of the linker is to match external references with unresolved symbols in the application's source files. External symbols in a file resolve a reference from another file when both refer to a label with the same name. Unmatched symbols are the result of using a label that was not defined any where. After matching the symbols in the source code, the linker searches the systems libraries to find pre-defined functions and data structures that the application references. A program that references an undefined symbol that is not in a library, contains an error and can not be linked. On the other hand, when a program utilizes a library routine, the linker extracts the library code and incorporates this into the text segment. This routine may in turn depend on other library routines, consequently the linker continues to fetch other library routines, until their are no more unresolved external references or such a routine can not be found.

At some point all external references are resolved, so the next step for the linker is to determine the memory locations that each module occupies. Recall that the assembler assembled the files in isolation so it has no idea where a modules instruction and data will be placed relative to other modules in Data memory. The linker relocates the absolute references to a module, when this is placed in memory such that its true location is reflected. To accomplish this the linker has access to relocation information that it utilizes to identify all relocatable references. This enables the linker to efficiently find and backpatch all references to absolute addresses.

7.5.1 Loader

We designed a simple loader in the form of a _startup() routine that is linked with the final executable of the linker. We programmed this routine in assembly. This means that it first has to be compiled before the object code is available for the last link step. The loader performs the following tasks:

1. **Initialize** the **stackpointer** register: We have to set the initial value of the stack to the maximal frame size. This is the size of Data memory as this is where the

activation records of each function will be kept. Of course we take care that after the function main() is invoked the value of the stack address is aligned to a 32-bit memory address.

2. **Call** the **main()** function of the application: This will invoke the target application to run on our VEX processor. Note that from the previous step, we can set the stackpointer to the non-existent value DATA_MEM_SIZE. Once main is called it subtract a value that is a multiple of 32. This aligns the resulting stackpointer.

3. **Stop** the processor: This is accomplished by printing the **STOP** syllable after the **CALL** syllable. After the application returns it will execute the stop instructions. In hardware, this causes the processor to stall. However, as there is a delay cycle in the VEX pipeline (*stop-synchronization*), we need to execute an extra **NOP** operation so that the pipelines state remains the same. Therefore, we need to insert an extra **NOP** syllable after the **STOP** operation in the loader.

These services are what we have considered to be the minimum low-level requirements of what a loader must do in order to run the application without necessitating a separate phase of assembly programming. We illustrate the start-up routine as it was written in VEX assembly.

```
bss align: 32
data align: 32

++ begin function: _start (global)
            c0    add $r0.1 = $r0.0 , 1024
;;      1]  ------------------------
            c0    call $l0.0 = main
;;      2]  ------------------------
            c0    stop
;;      3]  ------------------------
            c0    nop
;;
++ end function: _start
```

Once the startup routine is linked with the applications binary, the resulting executable file forms the basis for creating the processor's instruction ROM. This file has the same format as the object file from the assembler, but the difference is that it contains no unresolved references or relocation information. In the next section, we see that we need to generate a VHDL ROM description from this executable, such as to represent the true instruction ROM of the VEX processor. The module that is responsible for this translation is called elf2vhd.

7.6 ROM Generator

The C program in the file elf2vhd.c implements the generation of the instruction ROM from the resulting object file that was produced by the linker. In principle this modules reads the data and text sections, combines them and writes the instruction ROM with

the same syntax of the file i_mem.vhd. Note that the semantics that is stored in both these files are the same as these represent the instruction ROM. The module elf2vhd.c mainly consits out of the following functions:

- print_section()
- create_data()

Once we obtained the address width and the the data size of the VEX instruction, we initialize the bfd file descriptor. Utilizing this, we can open the executable and read the contents of each of its section. Subsequently, we call bfd_map_over_sections() function such that it reads each section of the executable to translate these to the syntax of the instruction ROM file. During this process bfd_map_over_sections() is assisted by print_sections(). This function in turn takes care of the low-level details of reading the text and data sections of the executable. The function create_data() actually goes through these sections as it reads in the bytes and prints out the instructions address and the contents of the instruction to the output ROM file. In a nut shell, these actions describes how the final instruction ROM is created. The exact low-level implementation details are given in elf2vhd.c.

7.7 Conclusion

We have reached the goal that our design and extended toolchain aimed for. This is true, as we have produced a toolchain which is both maturer and better to be utilized for generating the instruction ROM for the VEX processor. We have motivated why it was not possible to utilize the inherited toolchain without re-designing it.

First of all, we extended the back-end of the VEX compiler with a parser such that developing the assembler could be exercised without a heroic programming effort. Second, we have discussed the extention of the missing data initialization capabilities from the inherited assembler and its redesign. We motivated why we needed to simplify its control-flow. Moreover, we showed how we allowed the assembler to deal with pseudo-operations. The final implementation produced the correct VEX binaries and is still easier to understand. Furthermore, we have added an ABI to the assembler such that we could extend it with a linker. This was added as a collaboration within the ERA project since the toolchain proved to be more work then we had initially expected. When we benchmark the VEX processor, we will experience the consequences of this for our fingerprint application. During the last implementation phase of the assembler a simple loader was also designed, such that it could be linked along with the final executable. Once this proved satisfactory, we added the new ROM Generator to the toolchain such as to produce the final instruction ROM.

There were many problems encountered and solved of which the most important ones were discerned in this chapter. In retrospect, although it was not our initial plan to extent the toolchain to the current level, it was necessary. Altogether, implementing it gave us a better understanding of how the ABI fit into the processor's run-time system. The additional modules that were added (parser, loader, assembler, linker and

ROM generator) to the toolchain, made it maturer and possible to be utilized from a programmer's perspective. We have tested this toolchain on no less then 30 test-files, to verify that it produces the expected results.

8 Experimental Results

To measure the performance of our VEX processor, we have used four different bench marks. First of all, we have assessed if there was any performance advantage when executing an application on the VEX processor, instead of the ρ-VEX processor. Once this was determined, we executed three other benchmarks on the VEX processor.

This chapter is structured as follows. Section 8.1 explains the experimental setup used for our benchmarks. Section 8.2 presents each of their execution. The third benchmark represents the limitations and the results that we have observed when running the Fingerprint application on the VEX processor. Section 8.3 discusses the resource utilization of our VEX processor. Section 8.3 compares several key performance metrics of the realized VEX processor with it's predecessor. Finally, we draw this chapters to a conclusion in Section 8.5.

8.1 Experimental Setup

We have described the design of the VEX processor in VHDL and it's implementation was simulated with Mentor Graphics ModelSim SE 6.5e. The synthesis was perfromed with the XST from Xilinx ISE version 12.1.2. We decided not to choose the Xilinx Virtex -II PRO chip as the target FPGA. When this FPGA is chosen, the size of the Register file may not exceed 32 registers. Firstly, this implies that in the machine configuration files of the VEX compiler, we have to reduce the number of registers from 64 to 32, so as to utilize the toolchain for instruction ROM generation. However, when we adjusted the number of general-purpose registers, we observed that the compiler still targets register number 33 and above. This necessitates an embedded programmer to be forced to program in assembly, which is very complex and time consuming as he is targeting a multiple-issue processor. Hence, we decided to target the Xilinx Virtex-VI FPGA on the ML-605 board as our target platform. The experiments were performed on the VEX processor System with 64 general-purpose registers. The processors Data memory was implemented as a Block RAM of 1 Kb with additional read through capabilities for storing and accessing data in a pipelined processor. The VEX processor issue-width has four operations. This implies that there are also four ALU units. Within this processor the MUL units were relocated in issue-slot number 1 and 3 to deal with long immediate issues.

8.2 Benchmarks

We have managed to run four benchmarks on the VEX processor. The first benchmark was the assembly program which calculates the 45-th Fibonacci number from Fibonacci's sequence. This is a famous mathematical sequence where the first two numbers are defined as 0 and 1, and each subsequent number is computed as the sum of the previous two. It was the same assembly program used for benchmarking the ρ-VEX processor [4]. The reason we chose this benchmark was that it could prove that the VEX processor gave better performance then the ρ-VEX processor for a certain base benchmark application. It was problematic to use other benchmark applications for this purpose. Several attempts to build the required instruction ROM for the ρ-VEX processor kept showing more bugs in the previous Development Framework and in the ρ-VEX processor's hardware.

The second benchmark was chosen to demonstrate the effect of frequent memory accesses on a target application. In fact the same recurrence relation was programmed in assembly, but additional load and store operations were added to calculate and write all the forty-five results of the recurrence relation in data memory. Thus, this benchmark computed all of the 45 Fibonacci numbers.

The third benchmark was on the Fingerprint minutiae extraction. We executed the DFT bottleneck of it on the VEX processor. This proved to be the only bottleneck that was left in the application as we have seen from the results of porting it to the VEX simulator. However, due to limited time we have not managed to port the different libraries that our applications references, to the linker. This is required to execute the entire application on the VEX. Apart from accessing it's self-defined libraries the

application accesses the following libaries:

- math.h
- stdlib.h
- stdio.h
- string.h

Again, additional make-files have to be written, such that the linker can traverse the directory tree of the application and link each object file from the procedures that are referenced within each of the above libraries. Although, we are still optimistic there can be bugs that will show up when the linker is tested with the mentioned libraries.

Our fourth benchmark is an ADPCM application which performs voice compression and decompression of speech samples. We have adapted the source code of its codec, such that we could limit the number of speech samples that were processed. The encoder of this codec compresses speech samples of 16-bits to nibbles (4 bits). The decoder reverses this process by decompressing the compressed samples back to a 16-bits. We have limitted the number of samples that are processed to 10 but it is straight forward to make this number larger by increasing DATASIZE parameter of the sample buffer.

The number of executed cycles and the execution time for running the benchmarks on the VEX processor is depicted in Table 8.1. Executing the first benchmark on the ρ-VEX processor required 537 clock cycles. Executing it on the VEX processor requires 162 clock-cycles, which is only 30% of the original execution cycles. This shows that our VEX processor is better than the ρ-VEX processor in the number of execution cycles. When the performance is compared of both processors, one must also consider improvements to the clock frequency. This is discussed in Section 8.4. The number of execution cycles for executing the recurrence relation, the DFT and the ADPCM benchmark on the VEX processor are presented in Table 8.1. These three benchmarks could not execute on the ρ-VEX processor. We have not considered reconfigurable operations (ρ-OPS). Although, these can provide more performance they strongly depend on the target application, as they are not an intrinsic characteristic of the processor's micro-architecture. In addition there were also 20 other testfiles simulated on the VEX processor. The purpose for these test files was for verifying the proccesor and its tool-chain.

Benchmark	**Cycles**	**Time (μs)**
Fibonacci	162	1.62
Recurrence	580	5.80
DFT	27947	279
ADPCM	1211	12.11

Table 8.1: VEX Benchmarks results

8.3 Resource Utilization

We have performed synthesis of the VEX processor System, including the Instruction and Data memory. The resource usage of the VEX System was determined on target FPGA. The measured results are presented in Table 8.2. Next to the Slices, our design also requires Look-up table paired with flip-flops (LUT-FF). This is why we present this information together with the percentage from the VIRTEX-VI that is utilized for each of the resource types. The VEX implementation was synthesized to run at a clock speed of 117.45 MHz. This information was taken from the POST-PAR Static Timing report section for the following Device, Package, Speed: xc6vlx240t, ff1156, -1. The synthesis results indicate that the VEX processor can run at a higher **synthesis** clock-frequency compared with the ρ-VEX processor synthesized to run at 89.44 MHz. We have run the VEX processor at a **realization** clock frequency of 100 MHz on the FPGA. We present the details in Section 8.4.

Distribution	**Freq. (MHz)**	**Slice/LUT**
Slice GR	117.45	1900 (1%)
Slice LUTs	117.45	3573 (2%)
LUT-FF	117.45	1167 (25%)

Table 8.2: Resource utilization VEX

8.4 Comparing Performance

We would like to measure the speedup that an application would experience when it would be executed on the VEX processor compared to when it would run on the ρ-VEX. Initially, it should be possible to run it on both processors. This could create some issues for the the ρ-VEX processor as we have found that additional debugging and/or adjustments in its design are required. However, as we have already seen the Fibonacci benchmark formes an exception to our findings. This benchmark could run on both the ρ-VEX processor and the VEX processor.

Recalling Equation (5.1) which expresses the execution time of a processor, we like to note that the product of the **instruction count** for a program and the average clock-cycles per instruction (**CPI**) is equal to the **total clock-cycles** that are required for executing that program. When this is substituted in (5.1), the next (8.1) results. We fill in the realization clock frequency at which the processor was executed on the target FPGA. We have run the design at 100 MHz on the Virtex-VI FPGA. Subsequently, on the ρ-VEX processor the benchmark required 537 clock-cycles at a realization clock-rate of 50 MHz [4]. Therefore, the ρ-VEX processor execution time becomes 10.74 μs. However, our VEX processor seen in Table 8.1 requires 162 clock-cycles to execute the first benchmark. With the same computation, we see that the execution time becomes 1.62 μs. Then, the other execution time of the other three benchmarks were computed in the same manner and are presented in the last column of Table 8.1. Plugging both execution times into the speedup (5.1), we could assess what performance gain the new architecture provides.

$$CPU_{time} = \frac{CPU_{clock-cycles-program}}{Clock_{rate}} \tag{8.1}$$

The speed up is defined as the factor between the old computation time and the new computation time [22]. We formulate the relation as:

$$S = \frac{T_{original}}{T_{improved}} \tag{8.2}$$

The Speedup is calculated by filling in the execution time in (5.1): 10.74/1.62 = 6.64. Thus, a speedup of 6.64 is obtained for running the Fibonacci application on the VEX processor. Even if we run the VEX at a lower clock frequency of 50 MHz, we will still achieve a speedup of 3.32 for the Fibonacci benchmark.

8.5 Conclusion

This chapter presented the experimental setup that was utilized in order to measure the execution time and the resource utilization of the VEX processor. We have run the DFT kernel from the Fingerprint application and three other benchmarks on the processor. The results from these experiments were discussed. Before the entire Finger Print application can be executed on the processor, the new toolchain will require even more implementation time for porting the necessary libraries. In Chapter 4, we saw that the DFT kernel is the only remaining bottleneck when executing our application on the processor. Despite the experienced difficulties we were able to successfully run three other benchmarks in hardware. From these results we have argued that the toolchain allows the processor to run a broad range of applications. From the Fibonacci benchmark, we concluded that the VEX performs better than it's predecessor. The resource utilization when realizing the VEX processor on a VIRTEX-VI FPGA was presented. We have seen that this processor requires a small part of the FPGA resources.

When an embedded programmer needs to port his application in an automated way, it does not seem to make sense to target the VIRTEX-II PRO chip, as he will be forced to program in assembly. However, targeting the VEX running on a VIRTEX-VI FPGA will allow him to utilize the entire toolchain. In fact, he will obtain more performance than the ρ-VEX processor. When this FPGA is used to realize the VEX processor the application obtains a speed-up of 6.64 compared to an execution on the ρ-VEX processor.

9 Conclusion and Future Work

In this dissertation, we have presented the research that was performed in order to develop the VEX processor System. The purpose of the fingerprint minutiae extraction application was for evaluation, debug and proof of concept. More specifically, our focus has been on the following three objectives. First, to analyze the application such as to determine its bottlenecks and parallelize its execution on a the VEX simulator. Second, to develop a VLIW processor that could execute the parallelism that is extracted by the VEX compiler. Third, to re-design and extend the processor's toolchain such that it could generate the instruction ROM of the application for evaluating, debugging and proof of concept purposes.

In the first section of this chapter, we summarize the work that was performed. Section 9.2 specifies the main contributions of this work. Finally, in the last section we suggest future research directions.

9.1 Summary

In Chapter 2, we have discussed the fingerprint minutiae extraction application. We have argued that eventhough it implements a highly complex biometric technique, it may be understood when it is modeled as an image pipeline. This view model produces an image pipeline with six processing stages. The result of this pipeline is a list of the detected minutiae. In between the input image and the minutiae list, the following processing stages were described:

- Contrast enhancement: boosts the gray-scale value of regions in the input image with a blurry ridge structure. Such regions can be analyzed as their composing blocks have little dynamic range in the intensity distribution of their pixels.
- Quality analysis: computes and records the low-flow and high-curvature areas in the image. Moreover, the direction of ridges in a block is also computed based on the DFT algorithm.
- Binarization: converts the gray-scale image to black and white, as the actual detection can only work on a binary image.
- Minutiae detection: finds the minutiae based on a pattern recognition heuristic that scans and compares pairs of pixel to some pre-defined pixel pairs. When a similarity has been determined, a minutu is detected and its location is stored.
- False minutiae removal: corrects and tries to remove the incorrectly introduced minutiae that were detected by previous stages.
- Quality assessment: assigns a quality factor to all detected minutiae. This factor is based on the quality map and the intensity distribution of the pixels within the direct neighbourhood of the detected minutiae.

In addition, we have motivated our choice for the fixed-point version of the fingerprint application. The arguments were that it was faster, needed less storage and gave an output comparable with the original MINDTCT version. In fact the ISA of our target VEX processor had no support for floating-point operations.

We identified three specific objective in order to archive our goal in Chapter 3. The first objective was to prepare the application such that it could be ported to the VEX simulator. Porting it would allow us to analyse and measure the number of execution clock-cycles for different potential processor organizations. The second objective was to design the VEX processor according to architectural parameters that would be specified in the simulator. The aim was that the design had to yield the same performance as was predicted by the simulator results. The third objective was to further develop the inherited toolchain such that it could be utilized for porting the target application. The hardware design strategy for implementing the VEX processors was also presented. As we were designing a processor, an overview of the different available processor technologies was discussed. We have concluded that a costum VLIW processor satisfied all the important technology characteristics as performance/cost, time-to-market and flexibility.

Finally, we have reasoned that our application would require more of a stand-alone processor instead of a co-processor. Moreover, from the simulator results, we estimated that we required more performance than the ρ -VEX co-processor could provide us. As for the concerns of the toolchain, we reflected on the necessity for making it mature, design it correct, and extend it enough such that it could automate the generation of the instruction ROM for the target VEX processor.

The first objective of this thesis is met in Chapter 4, where we investigated how to parallelize the application on the VEX simulator. This is the cycle accurate processor simulator that was utilized for the VEX processor. We started by investigating the fingerprint application directory structure. Indeed this had a big impact on porting the application to the simulator.

We experienced that this even had consequences on how the rest of the toolchain was further developed. Before the application could run on the simulator, it was clear that there were several issues that had to be resolved. First, we had to write make files that could handle the tree structure of the application, such that it could be compiled, linked and executed on the simulator. Second, we had to adjust swapping the byte ordering when reading the image file. Despite the fact that the endianess assumption of the host and the VEX ISA are different, changing the byte order was incorrect as the simulator runs a compiled simulation code. Third, we had to add support for emulating 64-bits arithmetic as these computations were utilized by the fixed-point computations throughout the entire application. Lastly, we had to add exception detection capabilities to the arithmetic module in order to be sure that it was functioning correctly when perfroming 64-bits arithmetic for the application. Once these and other issues were resolved, we ported the application to the VEX simulator. This enabled us to analyze its bottlenecks, extract ILP from the code and optimize it for different VEX processor organizations. It was shown that the only remaining kernel was the DFT operations as was utilized by the directional analysis stage of the image pipeline. Based on the simulation results, we observed that a multi-cluster processor was not the best design option for this application. Whereas a single cluster with a issue-width of 4 looked more promising. This was explained by the fact that the application's execution time scaled better in the issue-width of a cluster than in the actual number of clusters.

In Chapter 5, we explained the design of a novel VLIW processor called the VEX processor. We studied, analyzed and discussed its ISA. In order to get a general overview of how the target hardware must execute the VEX operations during run-time, the processors execution model was investigated. The architectural state was considered in order to understand how architectural components must be designed such that they fit in the processors microarchitecture. An architectural classification of the ρ-VEX processor was analyzed. This analysis enabled us to make better and wiser decisions when composing our own microarchitecture. Furthermore, it gave us a general strategy how to implement the processors pipeline. Subsequently, we have proposed a novel organization of the processor as to provide a basis for the microarchitectural components, such as the branch architecture and the number of execution stages in the processors datapath. The former has been designed without offsetting the scheduling assumptions that are made when the VEX compiler schedules branch operations. The latter was designed such that it could be mapped to 2 pipeline stages. The miss-penalty was minimized to a single cycle. This was perfromed add the cost of more and complicated forwarding logic. Moreover, we regulated the syllable formats which provided us with an interface for designing the pipelines micro-architecture. In retrospect, we saw that almost all operation types were re-designed or enhanced in order to overcome the previous encoding problems. Here after, our focus was on the pipeline's micro-architecture. We have explained each of its 5 stages. This explanation resulted in a detailed description of the entire processor design. Furthermore, a motivation behind each of the made design decisions was given. Finally, in the last phase we devised three architectural improvements to maximize the performance that will be delivered by our VEX processor.

The second objective is met in Chapter 6. We explained how the design of the VEX processor was implemented. A bottum-up strategy guided us during its implemention. We set out by programming the architectural components of the first pipeline stage in VHDL. Once this was finished we moved on to test its results and when these were verified, we connected the components to form the Fetch pipeline stage. In the same way every subsequent pipeline stages was seperatly implemented and verified until all pipeline stages were implemented. At last, we connected the verified pipeline stages to form the pipeline of the VEX processor. We explained how Data- and Instruction memory were implemented such that they would satisfy the requirements of the pipeline. At this point the implementation of the basic processor was done. However, it can not function alone and needs I/O devices to supply data or to externally store data. We presented how these were adjusted in our implementation. Once this was performed the VEX CPU System

composed out of the VEX processor, the data- and instruction memories and the adjusted I/O devices, came into existence. After the implemention of the VEX processor was performed, we moved on to the processor's toolchain.

The third objective is met in Chapter 7. We have reached the goal that our designed toolchain aimed for. More specifically, we produced a maturer and better tool-chain to generate the instruction ROM of the VEX processor. We motivated why it was impossible to utilize the inherited toolchain without re-designing it. First of all, we extended the back-end of the VEX compiler with a parser such that designing the assembler could be performed with a modest programming effort. Second, we have discussed the implementation of data initialization capabilities and that of assembly directives in this assembler. Furthermore, we discussed why wee needed to simplify its control-flow. Moreover, we explained how we allowed the assembler to deal with VEX *pseudo*-operations. The final implementation produced the correct binaries and was easier to understand. An ABI was added to the assembler such that we could extend it with a linker. This part of the toolchain was incorporated as a collaboration within the ERA project. The reason was that the toolchain proved to be much more work than we had initially expected. The linker still requires external libraries and platform specific libaries to be ported to finalize its implementation. During the last implementation phase of the assembler, a simple loader was designed and implemented such that it could be linked along with the final executable of the application. Once this proved satifactory, we added a novel ROM Generator module to the toolchain, such as to produce the final instruction ROM from this executable.

There were many problems encountered and solved of which the most important ones were discerned. In retrospect, although it was not our initial plan to extent the toolchain to the current level, it was necessary for future applications. Altogether, implementing the the toolchain gave us a good understanding of how the ABI fit into the processors run-time system. From a programmer's perspective, we argued that the additional modules that were added (parser, assembler, loader, linker) to the toolchain made it maturer and better to utilize. We tested this toolchain on not less then 25 testfiles, as to verify that it produced the correct instruction ROM for the VEX processor.

In Chapter 7, we presented the experimental setup that was utilized in order to measure the execution time and the resource utilization of the VEX processor. We were able to run the DFT kernel from the fingerprint application and three other benchmarks on the VEX processor. The results of the performance and resource usage were measured and have been presented. We argued that before the entire Finger Print application will correctly execute on the VEX processor, it will require more implementation time for finishing the linker. The main reason for this is that the application reads libraries that the linker can not yet handle. In chapter 4, we showed that the DFT kernel is the only remaining bottleneck when executing our target application on the VEX simulator. Despite these difficulties, we managed to successfully run this kernel and three other benchmarks on the VEX processor in the target FPGA. In particular, the ADPCM application showed that other applications can also run on the VEX processor. From these results, we have argued that the toolchain allows the processor to run a modest range of applications. This range will broaden as the linkers implementation is finalized. Moreover, from the Fibonacci benchmark we concluded that the VEX performs better than its predecessor. The second benchmark was to investigate the effect of frequently addressing Data memory. The resource utilization when realizing the VEX processor on a VIRTEX-IV FPGA were presented. We saw that the VEX requires only a small part of the VIRTEX-IV resources.

An embedded programmer that needs to port his application to either the ρ -VEX or the VEX processor in an automated way should not target the VIRTEX-II PRO. The reason is that he will be forced to program in assembly. As both processors rely on the compiler to extract the avaliable ILP and because both are multiple-issue machines, programming will become a tedious task. However, targeting the VEX processor running on a VIRTEX-IV FPGA, allows the

programmer to utilize the entire toolchain. More importantly, he will obtain more performance. We showed that when this FPGA is utilized to realize the VEX processor, the application obtains a speed-up of 6.64 compared to the ρ-VEX processor.

9.2 Contribution

In this section, we highlight the main contribution of the work that was performed in this project.

- **Arithmetic Module:** applications that require 64-bits arithmetic can utilize the arithmetic module for providing this in the form of an emulation library.
- **Developed Application:** we have developed the application such that it could be ported to the VEX simulator. We analyzed, parallelized and profiled the application such as to derive the parameters of a VEX processor instance.
- **VEX processor:** we designed and implemented a novel VLIW processor to execute the extracted ILP. This processor is able to do so as it has a a micro-architecture which is able to exploits both temporal and spatial ILP from the target application. From the ρ-VEX processor, the VEX processor has inherited the characteristics of design time reconfiguration and extensibility.
- **Toolchain:** Delivered a toolchain for instruction ROM generation of the VEX. We have extended the back-end of the VEX compiler, re-designed the assembler, added a loader and a separate ROM generation module. Furthermore, through collabartion the toolchain was extended with a linker.
- **Performance analysis:** Benchmarking the VEX processor and comparing it with the ρ-VEX utilizing 4 benchmarks was perfromed. Among these, we were able to execute the DFT kernel of the fingerprint application on the VEX. A comparison was performed between the VEX and its predecessor. The second benchmark investigated the effect of frequently addressing Data memory. An ADPCM benchmark was executed to support our claims that the VEX processor is applicable for other applications.

9.3 Future Work

The future work that is recommended in this thesis are provided in this section. The suggestions are organized according to each research objectives:

Application

- Investigate whether adding support for custom VEX instructions to the arithmetic module will result in performance benefits. The VEX compiler has the ability to define and incorporate assembly intrinsics in the code sections that are frequently executed. It would be interesting to investigate whether the applications execution could improve from this.
- Investigate whether implementing the Cooley-Tukey decomposition for the DFT algorithm results in a significant performance increase [23]. This can allow the DFT kernel to be perfromed efficiently in software by utilizing a faster FFT algorithm.

VEX Processor

- Add support for clustering. This may be advantageous for other applications when the VEX simulator shows that the performance scales with a clustered architecture.
- Add support for the processor's exception behaviour. In VEX interrupts are currently decoded and executed. However run-time exchange of data between the core and other I/O devices is limitted. This could be solved by providing improved I/O designs with a System bus.

- Replace functional units with faster and smaller components. The ALU design should be aimed at reducing its area. Whereas the MUL units should aim at increasing their speed. This can be accomplished by utilizing the Baugh-wooley algorithm for the multipliers. The assumption that the VEX ISA makes is that the multipliers logic is divided over two pipeline stages. As the MUL Units perform 32×16 multiplication the resulting partial product three has half the three height of a standard array 32×32 multiplier.
- Add an extra pipeline stage between the Fetch stage and Decode stage for executing pre-decode operations for performing *horizontal nop folding*. In this way the fetched instruction stream for the VEX Core is compressed and the code size of ROM can be decreased.
- Add a DMA unit and its interface for off loading the VEX Core when streaming data is accessed. This could be efficient when the VEX is performing stream processing on video applications.
- Add partial reconfiguration of functional units to the VEX processor. This can allow the VEX processor to take better advantage of the fine grain parallelism that is available in some applications. Subsequently, this could be executed on Custom Computing Units.
- Interface to other VEX processors. When applications will be executed in a multi-core environment the VEX processor should have an interface that simplifies connecting it to a Network on Chip.

Toolchain

- Add platform- and external libraries to the toolchain. This will solve problems when the instruction ROM is generated for applications that reference such libraries.
- Investigate porting an Embedded Operating System to the VEX processor. This will require close collaboration with other proposed hardware oriented research directions. Hence, collaboration between adding support for the the processor's exception behaviour and porting an Embedded Operating System to the processor will be necessary.

Bibliography

[1] E. van Dalen, S.G. Pestana, *HiveFlex ISP2100: An Integrated, Low-Power Processor for Image Signal Image Processing*, white paper version 1.2, Silicon Hive B.V., February 2007.

[2] S. Aqeel, R. Seedorf, *VLIW Architectures: Current Trends and Techniques used in the Embedded domain.* student paper Computer Architecture special topics (ET4078), Delft University of Technology, August 2008.

[3] M. van der Net, *A SoC Solution for Fingerprint Minutiae Extraction*, Masters's thesis, Delft University of Technology, July 2008.

[4] T. van As, *ρ-VEX: A Reconfigurable and Extensible VLIW Processor*, Masters's thesis, Delft University of Technology, September 2008.

[5] G. Goede, *Accelerating the XviD on DAMP*, Masters's thesis, Delft University of Technology, December, 2004.

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

[7] C.I. Watson, M.D. Garris et al., *User's Guide to NIST Biometric Image Software (NBIS)*, National Institute of Standards and Technology (NIST).

[8] C.J. Lee, S.D. Wang Maltoni, *"Fingerprint Feature Reduction by Principal Gabor Basis Function"*, *Pattern Recognition*, vol.34, no. 11, pp 2245-2248, 2001.

[9] J. Fisher, P. Faraboschi, and C. Young, *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools.* Morgan Kaufmann, 2004.

[10] J.S.S.M. Wong, *Reconfigurable Embedded processors*, PhD thesis, Delft University of Technology, December 2002.

[11] M. Sima, *The ρ- Trimedia Processor*, PhD thesis, Delft University of Technology, February, 2004.

[12] P. Faraboschi, G. Brown, J.A. Fisher, G. desoli, F. Homewood, *Lx: A Technology Platform for Customizable VLIW Embedded Processing*, Hewlett Packard Laboratories, Cambridge MA, STmicroelectronics Cambridge MA., ACM 2000.

[13] W. Wolf, *Computers as Components: Principles of Embedded Computing System Design*, Academic Press London, 2001.

[14] P.G. Lowney, S.M. Freudenberger, T.J.Karzes, W.D. Lichtenstein, R.P. Nix, J.S. O'Donell, J.C. Rutttenberg, *The Multiflow Trace Scdeduling Compiler*, Kluwer Academic Publishers, Boston, 1993.

[15] Hewlett-Packard Laboratories. VEX Toolchain. [Online]. Available: http://www.hpl.hp.com/downloads/vex/

[16] B. Parhami,*Computer Arithmetic Algorithms and Hardware designs*, Oxfords University Press, New York

[17] Xilinx Corporation. *XST User Guide*, version 11.3. [Online]. Available: http://www.xilinx.com/support/documentation

[18] R. Nouta, *VHDL Synthesis support of the program Synplify Pro*, Course: VLSI Systems Design (ET4066), March, Delft University of Technology, 2003.

[19] R. Nouta, *Field Programmable Gate Array's (FPGA's) and doucmentation of 3 modern Xilinx FPGA's*, Course: VLSI Systems Design (ET4066), March, Delft University of Technology, 2003.

[20] D. Grune, H.E. Bal, C.J.H. Jacobs, K.G. Langendoen, *Modern Compiler Design*, Wiley, New York, 2002.

[21] D.A. Patterson, J.L. Henessy, *Computer Organization And Design: The Hardware/Software Interface*, Third edition, Morgan Kaufmann, 2005.

[22] J.L. Henessy, D.A. Patterson, *Computer Organization And Design: A Quantitative Approach*, Third edition, Morgan Kaufmann, 2003.

[23] B. Porat, *A Course In Digital Signal Processing*, John Wiley & Sons, 1997.

[24] G. Wijnen, P. Storm, *Projectmatig Werken: Over samenwerking, risico's aanpak en procesmanagement*, Spectrum, 2007.

[25] Wikipedia, the free encyclopedia. [Online]. Available: http://en.wikipedia.org

[26] Wikipedia, the free encyclopedia. [Online]. Available: http://en.wikipedia.org/wiki/Dilation_(morphology)#Grayscale_dilation

[27] Wikipedia, the free encyclopedia. [Online]. Available: http://en.wikipedia.org/wiki/Binary_File_Descriptor_Library

Machine Configuration File

A

Below is the machine configuration file which specifies how to execute the MINDTCT application on the VEX simulator. The target architecture is a 4-issue VEX default cluster with bypass hardware and register file forwarding

```
CFG: Debug 0
RES: IssueWidth 4
RES: MemLoad 1
RES: MemStore 1
RES: MemPft 0
RES: IssueWidth.0 4
RES: Alu.0 4
RES: Mpy.0 2
RES: CopySrc.0 1
RES: CopyDst.0 1
RES: Memory.0 1
DEL: AluR.0 0 DEL: Alu.0 0
DEL: CmpBr.0 1 # expose branch latency to compiler
DEL: CmpGr.0 0
DEL: Select.0 0
DEL: Multiply.0 1
DEL: Load.0 1
DEL: LoadLr.0 1
DEL: Store.0 1 # default latency
DEL: Pft.0 0
DEL: Asm1L.0 0
DEL: Asm2L.0 0
DEL: Asm3L.0 0
DEL: Asm4L.0 0
DEL: Asm1H.0 1
DEL: Asm2H.0 1
DEL: Asm3H.0 1
DEL: Asm4H.0 1
DEL: CpGrBr.0 0
DEL: CpBrGr.0 0
DEL: CpGrLr.0 0
DEL: CpLrGr.0 0
DEL: Spill.0 0
DEL: Restore.0 1
DEL: RestoreLr.0 1
REG: $r0 63
REG: $b0 8
```

Manual B

This appendix is the manual that describes how to utlize the toolchain. We assume that the user has already generated the required assembly files by compiling the source files of the application with the VEX compiler. Hereafter, the toolchain consiting of the following modules can be utilized:

- Extended back/end: asm
- Assembler: r-asm
- linker: ld-new
- loader: _start.s
- Rom-generator: elf2vhd

To utlize the toolchain a sequence of five steps have to be taken. Each of these steps invokes a particulair module from the toolchain. The final instruction memory for the VEX processor is build from the final excutible of the entire application. This memory file is called **prom.vhd**, and contains the instruction ROM of the VEX processor. The next section explains how to invoke each of the modules.

B.1 Parser

First, it is required to pull the generated assembly files through the parser. This is performed by invoking the following commands:

```
cd r-tools/asm
./vexasm ../demos/asm.s ./r-asm/tests/asm_out.s
```

The output is an assemblly file that has been pre-processed and stripped from all irrelefant information for the assembler

B.2 Assembler

First, it is required to generate the VEX binary by pulling the output assembly file through the assembler. This is performed by invoking the following commands:

```
cd ../src
./rasm ./test/asm_out.s
```

The first line causes the shell to change to the directory where the assembler resides. The second call invokes the assembler on the file that was generated by the parser. The output is an object file called asm_out.o that has been assembled and written to the ./test directrory. When the application consits of multiple source file the previous two steps have to be repeated untill all object files are produced.

B.3 Linker and loader

Now that all object files were generated we can link them such as to create the final executible for the application. This is accomplished by invoking the following commands:

```
cd ./tests      r-tools/src/tests ../../binutils/ld/ld-new _start.o asm_out.o
```

What happens is that we link the final executible with the object file of the loader. By default the loader can handle a data memory size of 2048 instructions (2Kb). When larger applications have to be compiled the user has to adjust the assembly file of the loader e.g. _start.s. Subsequently, he needs to pull the resulting object file through the assembler, such that the adjusted loader can be linked as is described here.

B.4 Rom generator

The last step is to generate the instruction Rom for the VEX processor, which is called prom.vhd. To perform this, we copy the executible of the application to the directrory of the rom-generator called elf2vhd. Then we change directory to rom-generator. Finally, we pass the address width of instruction memory and the final executible as the parameters to the rom-generator.

```
cp a.out ../../elf2vhd/
cd ../../elf2vhd/
./elf2vhd -a 11 a.out
```

The reader can copy the prom.vhd to the source directory of the processor's vhdl files. At last he will need to include this file into the processor's source files. This allows him to perform a synthesis with xilinx for the target FPGA.

Curriculum Vitae

Roël Seedorf was born on 7 september, 1977 in Paramaribo Suriname. He graduated with honours from the W.Ritfeldschool Mulo B in 1993. From 1994 until 1996 he attended the AMS VWO school in Suriname. After coming to holland he attended from 1996 till 1998 the Scholen gemeenschap Reigersbos, where he obtained his Atheneum diploma.

In september 1999 he started his studies at the Delft University of Technology. After obtaining his Bachelor of Science degree in Electrical Engineering, he continued his studies at the Computer Engineering Laboratory where he hopes to receive the Master of science degree. His research interest are: Embedded System design, Computer Architecture, Digital Signal processing, Embedded Application Development and Microprocessor System Design.

Besides his academic studies he likes to play basket ball, cycling and listening to music. Furthermore, he enjoys playing chess and reading good books.