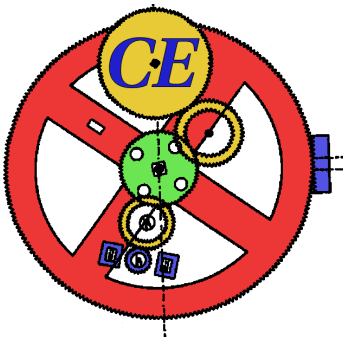


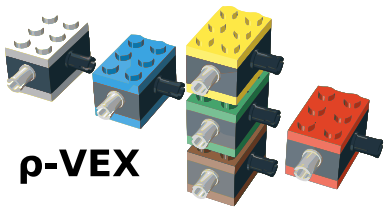
Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2008

MSc THESIS

ρ-VEX: A Reconfigurable and Extensible VLIW Processor

Thijs van As



CE-MS-2008-12

Abstract

Increasingly more computing power is being demanded in the domain of multimedia applications. Computer architectures based on reconfigurable hardware are becoming more popular now that classical drawbacks are diminishing. Field-Programmable Gate Arrays (FPGAs) are constantly improving in terms of performance and area, and provide a technology platform that allows fast and complex reconfigurable designs. The MOLEN polymorphic processor provides the possibility of executing an application-specific core in a custom generated hardware unit, which resides inside a reconfigurable fabric. This thesis presents the architectural design and implementation of a reconfigurable and extensible open source Very Long Instruction Word (VLIW) processor: ρ-VEX. In addition to architectural extensibility, our processor also supports reconfigurable operations. Furthermore, we present an application development framework to optimally exploit the freedom of reconfigurable operations. Because ρ-VEX is based on the VEX ISA, we already have a good compiler which is able to deal with ISA extensibility and reconfigurable operations.
Our processor is targeted to be a Custom Computing Unit (CCU) within a MOLEN reconfigurable computing machine. To estimate the performance gains, we present a performance analysis based on the VEX simulator. Results of benchmarks on real hardware show that different configurations of our processor in a stand-alone environment lead to considerable cycle count reductions for a selected benchmark application. 1-, 2-, and 4-issue ρ-VEX configurations were synthesized and implemented in real hardware to operate at a maximum clock frequency of 89 MHz.

PROJECT WEBSITE: `http://r-vex.googlecode.com/`

TU Delft
Delft University of Technology

Faculty of Electrical Engineering, Mathematics and Computer Science

ρ-VEX: A Reconfigurable and Extensible VLIW Processor

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Thijs van As
born in Vlaardingen, The Netherlands

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

ρ-VEX: A Reconfigurable and Extensible VLIW Processor

by Thijs van As

Abstract

Increasingly more computing power is being demanded in the domain of multimedia applications. Computer architectures based on reconfigurable hardware are becoming more popular now that classical drawbacks are diminishing. Field-Programmable Gate Arrays (FPGAs) are constantly improving in terms of performance and area, and provide a technology platform that allows fast and complex reconfigurable designs. The MOLEN polymorphic processor provides the possibility of executing an application-specific core in a custom generated hardware unit, which resides inside a reconfigurable fabric.

This thesis presents the architectural design and implementation of a reconfigurable and extensible open source Very Long Instruction Word (VLIW) processor: ρ-VEX. In addition to architectural extensibility, our processor also supports reconfigurable operations. Furthermore, we present an application development framework to optimally exploit the freedom of reconfigurable operations. Because ρ-VEX is based on the VEX ISA, we already have a good compiler which is able to deal with ISA extensibility and reconfigurable operations.

Our processor is targeted to be a Custom Computing Unit (CCU) within a MOLEN reconfigurable computing machine. To estimate the performance gains, we present a performance analysis based on the VEX simulator. Results of benchmarks on real hardware show that different configurations of our processor in a stand-alone environment lead to considerable cycle count reductions for a selected benchmark application. 1-, 2-, and 4-issue ρ-VEX configurations were synthesized and implemented in real hardware to operate at a maximum clock frequency of 89 MHz.

Laboratory : Computer Engineering
Codenumber : CE-MS-2008-12

Committee Members :

Advisor:	Dr.ir. J.S.S.M. Wong, CE, TU Delft
Advisor:	Prof. G. Brown, CS, Indiana University
Chairperson:	Dr.ir. K.L.M. Bertels, CE, TU Delft
Member:	Dr.ir. T.G.R. van Leuken, CAS, TU Delft

Dedicated to my parents,
for their love and support.

Contents

List of Figures **ix**

List of Tables **xi**

Acknowledgements **xiii**

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Project Goals . . . 2
1.3 Thesis Organization . . . 3

2 Background **5**
2.1 Related Work . . . 5
2.1.1 RISC Softcore Processors . . . 5
2.1.2 VLIW Softcore Processors . . . 6
2.1.3 Reconfigurable Processors . . . 6
2.2 ILP and VLIW . . . 7
2.3 The VEX VLIW Architecture . . . 8
2.3.1 The VEX ISA . . . 8
2.3.2 VEX Machine Configurations . . . 9
2.3.3 Cache Configuration . . . 10
2.3.4 The VEX Simulation System . . . 11
2.4 The MOLEN Polymorphic Processor . . . 11
2.5 Conclusions . . . 13

3 Performance and Configuration Analysis **15**
3.1 Lx Analysis . . . 15
3.2 VEX Analysis . . . 17
3.2.1 Simulator Benchmark Set . . . 18
3.2.2 Simulator Benchmark Results . . . 19
3.3 Conclusions . . . 22

4 Design **23**
4.1 Organization . . . 23
4.2 Instruction Layout . . . 24
4.2.1 ALU and MUL Syllables . . . 26
4.2.2 CTRL Syllables . . . 27
4.2.3 MEM Syllables . . . 27
4.2.4 Other Syllables . . . 29
4.3 Extensibility . . . 29

4.3.1 ρ-OPS . . . 29
4.3.2 VEX Machine Models . . . 30
4.4 Conclusions . . . 31

5 Implementation **33**
5.1 Bottom-Up Implementation . . . 33
5.2 Mapping the Design to an Implementation . . . 33
5.2.1 Instruction Flow . . . 34
5.2.2 Datapath . . . 34
5.2.3 Inter-Stage Control Signals . . . 36
5.2.4 Pipelining the Design . . . 38
5.3 Internal Stage Control . . . 38
5.3.1 Fetch Stage . . . 39
5.3.2 Decode Stage . . . 39
5.3.3 Execute Stage . . . 41
5.3.4 CTRL Stage . . . 41
5.3.5 MEM Stage . . . 42
5.3.6 Writeback Stage . . . 42
5.4 System Verification and Testing . . . 43
5.4.1 The Unified Verification Methodology . . . 43
5.4.2 System Wrapper . . . 43
5.4.3 Simulations . . . 44
5.4.4 Verification on Hardware . . . 45
5.5 Encountered Problems & Solutions . . . 46
5.6 Conclusions . . . 46

6 Development Framework **49**
6.1 Application Development Flow . . . 49
6.2 ρ-ASM Assembler . . . 50
6.3 UART Debugging Interface . . . 50
6.4 Hardware Development . . . 51
6.5 Conclusions . . . 52

7 Experimental Results **53**
7.1 Experimental Setup . . . 53
7.2 Fibonacci's Sequence Benchmark . . . 53
7.3 Resource Utilization . . . 54
7.4 Conclusions . . . 54

8 Conclusions **57**
8.1 Summary . . . 57
8.2 Main Contributions . . . 59
8.3 Future Work . . . 59

Bibliography **63**

List of Acronyms **66**

A Name and Logo **67**

B VEX Operations & Semantics **69**

C ρ-VEX Machine Model **73**

D Fibonacci Benchmark Assembly Code **75**
D.1 1-Issue VEX Assembly Code 75
D.2 2-Issue VEX Assembly Code 76
D.3 4-Issue VEX Assembly Code 76
D.4 4-Issue VEX Assembly Code With ρ-OPS 77

E Fibonacci Benchmark Simulation Waveforms **79**
E.1 Behavioural Simulation Waveforms 79
E.2 Post-Place and Route Waveforms 79

F Quickstart Guide **83**
F.1 Requirements 83
F.2 Deploying ρ-VEX on an FPGA 83
F.3 Assembling and Running Code 84
F.4 Using ρ-OPS 85
F.5 Running ModelSim Simulations 86
F.6 Adding Support For Other FPGA Boards 87

G Package Contents **89**

List of Figures

2.1 Structure of the default VEX cluster [1] . . . 9
2.2 Structure of a VEX multi-cluster implementation [1] . . . 10
2.3 The MOLEN machine organization [2] . . . 12
2.4 ρ-VEX integration within the MOLEN workflow . . . 13

3.1 Lx performance chart, varying cluster-width from 1 to 4 [3] . . . 16
3.2 VEX performance chart based on default cache configurations (DEF) . . . 20
3.3 VEX performance chart based on 'no cache' configurations (NO) . . . 20
3.4 VEX performance chart based on proposed cache configurations (PROP) 21

4.1 ρ-VEX organization (4-issue) . . . 23
4.2 Instruction layout . . . 25
4.3 Generic syllable layout . . . 26
4.4 Syllable layout for **ADDCG** and **DIVS** syllables . . . 27
4.5 Syllable layout for **GOTO** and **CALL** syllables . . . 28
4.6 Syllable layout for **BR** and **BRF** syllables . . . 28
4.7 Syllable layout for **RETURN** and **RFI** syllables . . . 28
4.8 Syllable layout for memory load syllables . . . 28
4.9 Syllable layout for memory store syllables . . . 28
4.10 Syllable layout for **NOP** syllables . . . 29
4.11 Syllable layout for **STOP** syllables . . . 29

5.1 Instruction flow . . . 34
5.2 Datapath . . . 35
5.3 Inter-stage control signal flow . . . 37
5.4 Moore FSM for the *fetch* stage. . . . 39
5.5 Mealy FSM for the *decode* stage. . . . 40
5.6 Moore FSM for the *execute* stage. . . . 40
5.7 Moore FSM for the MEM stage. . . . 41
5.8 Moore FSM for the *writeback* stage. . . . 42
5.9 Schematic representation of the system wrapper . . . 44

6.1 ρ-VEX application development framework . . . 49

A.1 ρ-VEX logo . . . 67

E.1 Behavioural simulation: 0 – 1330 ns . . . 80
E.2 Behavioural simulation: 21490 – 22820 ns . . . 80
E.3 Post-place and route simulation: 0 – 1330 ns . . . 81
E.4 Post-place and route simulation: 21490 – 22820 ns . . . 81

List of Tables

3.1 The Lx benchmark set taken from SPECINT'95 [3] 16
3.2 Number of clock cycles executed per system in the VEX benchmark set . 19

4.1 Opcode space distribution 24
4.2 Immediate types 25

5.1 Writeback targets in *writeback* stage 36

7.1 Results of Fibonacci sequence benchmark 54
7.2 Resource utilization for different ρ-VEX configurations 54

E.1 Syllables explained from the Fibonacci's Sequence benchmark 82

Acknowledgements

First of all, I would like to thank my supervisor Stephan Wong for guiding me through my MSc project. In addition to this, I want to thank him for the interesting discussions we had about the future of computing, proofreading the paper, and his advice concerning (academic) career opportunities.

I also want to express my gratitude to my advisor Geoffrey Brown from Indiana University. He gave valuable advices during the project, and reviewed the paper. He also helped me in my search of an international career, for which I am grateful.

I would like to thank my friends for showing interest in my work, and getting bothered with LEDs lightening binary representations of Fibonacci numbers. Especially with Kristian I had some interesting discussions concerning microprocessor designs. I'm grateful to André, Siebe, and Tamar for proofreading my thesis and giving feedback.

Last but not least, I want to thank my parents for their support during my studies.

Thijs van As ⟨t.vanas@gmail.com⟩
Delft, The Netherlands
September 1, 2008

1 Introduction

The demand for computing power in consumer electronics is increasing at a very high rate. In the age of the Internet, multimedia and 3D visualizations, people need high-performance (embedded) computers in order to cope with all modern applications. In order to achieve even higher performances, computer systems with a single general-purpose Central Processing Unit (CPU) are making place for computer systems with multiple general-purpose CPUs, or a combination of CPUs and application-specific processing units. This thesis presents an architectural alternative to current computing machines to achieve a high performance within application-specific computations in general-purpose machines.

Section 1.1 presents the motivation behind the presented work. Subsequently, project goals are identified in Section 1.2. Section 1.3 concludes this chapter with an overview of this thesis' organization.

1.1 Motivation

It was identified that the performance of application-specific computations within general-purpose computing machines lacks behind the performance of the same computations on application-specific computing machines. In order to achieve higher performances within the application-specific domain, we combined several technologically proven paradigms to provide a new architectural solution for a general-purpose computing machine.

The design of computer architectures on reconfigurable hardware is becoming more popular now that classical drawbacks are diminishing. Field-Programmable Gate Arrays (FPGAs) are constantly improving and provide a technology platform that allows fast and complex reconfigurable designs. In many cases, the utilization of FPGAs implies a large reduction in development costs, an enormous speedup of the implemented algorithm, or both. Nowadays, a broad spectrum of reconfigurable architectures are used for applications that would have been implemented in Application-Specific Integrated Circuit (ASIC) technologies or as software for a general-purpose processor [4].

The MOLEN polymorphic processor [2, 5, 6] provides the possibility of executing an application-specific core in a custom generated hardware unit, which resides inside a reconfigurable fabric. The general-purpose processor within a MOLEN machine takes care of general-purpose calculations, concurrently with application-specific calculations by the custom unit. Overall application speedups of more than 3 times have been achieved on different state-of-the-art multimedia applications [7].

Very Long Instruction Word (VLIW) processors are efficient machines for calculations that contain a lot of Instruction Level Parallelism (ILP) that can be exposed by a good compiler. Applications in the multimedia domain happen to contain a lot of ILP, because

they typically consist of many independent repetitive calculations.

By means of embedding a VLIW co-processor inside the reconfigurable fabric of a MOLEN machine, we aim to bridge the gap between the execution time of an application-specific kernel on the general-purpose processor and a custom generated hardware unit. This would result in a compromise between two fields. The first is the execution time of the application-specific kernel, and the second is the on-chip area used for the hardware. One VLIW co-processor is able to perform a large number of different calculations within a fixed area footprint, whilst a custom hardware unit is probably only able to perform one type of calculation on a fixed area footprint.

Most processor Instruction Set Architectures (ISAs) define many atomic operations. However, in many applications a custom operation[1] would result in an increase of the performance (and a decrease in power dissipation). This is why we want our processor to have support for reconfigurable operations.

In this thesis, the design and implementation are presented of an embedded reconfigurable and extensible open source VLIW processor, accompanied by a development framework. Our processor architecture is based on the VLIW Example (VEX) ISA, as introduced in [1]. The VEX ISA offers a scalable technology platform for embedded VLIW processors, that allows variation in many aspects, including instruction issue-width, organization of Functional Units (FUs), and instruction set. A software development compiler toolchain for VEX is made publicly available by Hewlett-Packard [8]. The reasons to choose the VEX ISA for this project are merely its extensibility and the quality of the available compiler. Our design provides mechanisms that allow parametric extensibility of the new processor, called ρ-VEX. Both reconfigurable operations, as well as the versatility of VEX machine models are supported by ρ-VEX. Our processor and framework are targeted at VLIW prototyping research and embedded processor design in a stand-alone environment. After some further work regarding MOLEN integration, ρ-VEX will be a scalable co-processor for the utilization within a MOLEN machine. The results of our preliminary performance analysis in Chapter 3 show that the inclusion of a VLIW co-processor within a MOLEN machine pays off in terms of performance.

1.2 Project Goals

The main goal of this project is to design and implement an extensible and reconfigurable VLIW processor according to the VEX ISA, that can be eventually used as a co-processor within a MOLEN machine. To justify the existence of such a VEX co-processor, a

[1] Throughout this thesis, the following naming conventions are used for operation, instruction and syllable:

- An operation is defined as an atomic command for the processor to be executed, e.g. the addition of two operands.
- An instruction is defined as the data fetched from the instruction memory, in which a number of operations is defined together with their operands and destinations.
- A syllable is defined as a combination of a single operation together with its operands and destination. A syllable is the same as an instruction in RISC machines.

preliminary performance analysis was performed in Chapter 3 in order to show the benefits.

The following stages have been identified in order to achieve the main goal of this project:

1. For the first stage of the processor design, a 1-issue RISC version of ρ-VEX should be designed and implemented. This processor implementation should be able to issue one operation at a time.

2. After the RISC version of ρ-VEX, a 4-issue version should be implemented, using the default VEX machine model. This version of ρ-VEX should allow parametric extensibility so that the operation issue-width, the configuration of Functional Units (FUs) and the operations could be easily adapted.

3. After the hardware platform is implemented, an assembler tool (ρ-ASM) should be created to assemble VEX assembly instructions to machine code which can be executed by the ρ-VEX processor.

1.3 Thesis Organization

The remainder of this thesis is organized as follows. A background on related work and underlying technologies as the VEX VLIW architecture and the MOLEN polymorphic processor is presented in Chapter 2. Chapter 3 discusses the results of our preliminary performance and configuration analysis of ρ-VEX serving as a co-processor within a MOLEN machine. Subsequently, the architectural design of ρ-VEX is presented in Chapter 4. Next, we present our hardware implementation and testing methods in Chapter 5. Chapter 6 discusses the ρ-VEX application development framework and design flow that we designed. Results obtained practical experiments are presented and discussed in Chapter 7. Finally, this thesis is concluded in Chapter 8 where recommendations for future work can also be found.

A clarification of the chosen name and logo is given in Appendix A. Appendix B presents an overview of the VEX operations, their semantics and the corresponding ρ-VEX opcode and operation type. The VEX machine model for ρ-VEX is presented in Appendix C. Appendix D enlists the assembly code used for the different benchmarks on Fibonacci's Sequence. Appendix E presents waveform diagrams of behavioural and post-place and route simulations of Fibonacci's Sequence benchmark. In Appendix F, a Quickstart Guide is presented to easily deploy ρ-VEX on an FPGA development board. Appendix G presents the contents of the ρ-VEX package as it can be downloaded from the project website.

All latest hardware and software source code of this project can be found on the project website at `http://r-vex.googlecode.com`. All source code is released under the *GNU General Public License v3* [9].

2 Background

Our work is mainly based on two projects in computer architectural design, namely the VEX VLIW ISA, and the MOLEN polymorphic processor. This chapter presents a background on the underlying technologies.

Section 2.1 presents an overview of related work done by others in the past. Section 2.2 discusses exploiting Instruction Level Parallelism (ILP) and the general ideas behind Very Long Instruction Word (VLIW) processors. Subsequently, the VEX VLIW architecture is presented in Section 2.3. The MOLEN processor architecture is presented in Section 2.4. Finally, this chapter is concluded in Section 2.5.

2.1 Related Work

Different softcore approaches resulted in FPGA-based system designs that achieved notable performances. An overview of related work is presented, structured per processor type.

2.1.1 RISC Softcore Processors

Well-known Reduced Instruction Set Computer (RISC) softcore processors MicroBlaze [10] by Xilinx and Nios II [11] by Altera provide efficient sequential architectures, optimized for the reconfigurable devices of their respective designers. Both Xilinx and Altera provide a software development toolchain with an extensive library base for fast application development. The toolchains provided for MicroBlaze and Nios II are based on the GNU tools, including the GNU C Libraries.

Both softcores are widely used, and are proved to provide efficient application development cycles. However, these processors do only expose a small degree of extensibility. Some parts of the processor are parametric within their accompanying Integrated Development Environment (IDE), but the largest part of the design is fixed. As they are RISC processors, no changes can be made to e.g. the issue-width. Additionally, they are not open source and in many situations require costly licenses to be used.

A well-known open source RISC processor is OpenRISC [12]. This processor core has fully ported GNU toolchain, and extended connectivity options like a Wishbone [13] bus interface. uClinux has also been ported to run on this processor. However, internal customizability is not very high, and the issue-width can not be changed throughout the toolchain.

A 32-bit open source softcore processor by Lattice Semiconductor is LatticeMico32 [14]. The source code of the processor and software development toolchain is provided through a custom open source license. LatticeMico32 has many of the features that

OpenRISC has, like Wishbone connectivity. The same disadvantages concerning customizability can be accounted to this processor as well.

MicroCore [15] is another open source softcore processor, targeted at hardware developers that want to have full control over the microprocessor and the embedded hardware connected to it, as well as the software running on it. A software toolchain is provided, and the architecture is very customizable. However, a wider issue-width than 1 is not architecturally supported.

2.1.2 VLIW Softcore Processors

The first VLIW softcore processor found in existing literature is Spyder [16, 17]. The architecture of Spyder consists of three reconfigurable execution units, of which the compiler toolchain decides their configuration based on a library of known configurations. A developer could also add his own configurations to this library. Spyder did not evolve extensively, but its design and implementation marked the beginning of more (reconfigurable) VLIW softcore processor designs. One of the drawbacks of Spyder was that both the processor architecture as well as the compiler were designed from scratch. This implied that the designers had to work on improvements of both the processor and the toolchain.

Later, several customizable VLIW softcore projects like [18], [19] and [20] were presented. A limitation of the former architectures is mainly the absence of extensibility (like adjusting the issue-width and changing the number of functional units), or the absence of a good software toolchain.

In [21], a parametric customizable VLIW processor based on a subset of the EPIC ISA [22] is presented. This processor also supports reconfigurable operations. However, the complete support for custom operations throughout the (simulation) software & hardware toolchain and the flexible machine models that enable fast trade-off studies on functional units make our design stand out.

Another hardware implementation of a VEX machine is presented in [23]. However, in this implementation VEX assembly is used as an input to a more conventional hardware compiler. More specifically, instead of building a general-purpose VEX VLIW processor to execute code, it converts the assembly code into custom hardware.

2.1.3 Reconfigurable Processors

The Chimaera [24] architecture supports issuing reconfigurable operations, like the MOLEN architecture. The Chimaera system describes a reconfigurable functional unit within a fixed core general purpose processor. This reconfigurable unit is able to access the register files directly, as it resides within the processor pipeline. This in contrast to the MOLEN architecture, where reconfigurable computing units reside outside the core processor.

The Garp [25] system was designed specifically for accelerating loops within general-purpose software applications. A Garp machine consists of a single-issue MIPS processor, together with a reconfigurable co-processor. A custom compiler is used to compile application code for the architecture. The MIPS instruction set is augmented with a number

of non-standard instructions to load a new configuration, moving data, and starting the execution of the reconfigurable co-processor.

2.2 ILP and VLIW

A computer system embodies Instruction Level Parallelism (ILP) when it has the ability to execute multiple different operations at the same time, while a single stream of instructions is presented. Because it is an architectural technique, it is independent on changes in technology, like circuit speed [1]. Because the effects on the system performance can be notably significant, most processors exploit ILP in one way or another these days.

ILP is often compared or confused with other types of processor parallelism, as discussed in [1]. Other types of parallelism include:

- **Vector processing** – A vector processor is able to perform a single operation on a vector of operands. For example, an operation for addition can operate on a pair of 128-bit operands that both contain four 32-bit integer values. Vector processors are based on the Single Instruction, Multiple Data (SIMD) principle.
- **Multiprocessing** – A multiprocessor computer has multiple processors to execute multiple programs, or parts of programs, at the same time.
- **Multithreading** – Multithreading is a system in which multiple light independent processes on a system are alternatingly given focus of a single processor. Most of the time, these processes have their own set of registers but share the same Functional Units.
- **Micro-SIMD** – Micro-SIMD is a system in which vector operations operate on standard-sized architecture registers. This only incorporates selected operations per architecture. MMX [26] by Intel is an example of a micro-SIMD system.

These techniques may share the same kind of parallelism that is exposed in ILP. When this occurs, only one of the techniques to exploit the parallelism can be used. In other cases, multiple forms of parallelism are exposed and can be exploited separately by using the aforementioned techniques.

Current trends in exploiting ILP in single processor cores are mainly represented by architectures known as 'superscalar' and 'Very Long Instruction Word (VLIW)'. Both architectures exploit ILP by issuing more than one operation per issue slot to additional FUs. The main difference between these two architectures is that a superscalar processor issues operations from a single-operation instruction stream, while a VLIW processor issues operations from a multi-operation instruction stream. This means that a superscalar processor should have hardware that enables dynamic scheduling, while a VLIW processor can issue pre-scheduled operations (in this case, the operations are scheduled by the compiler). The big advantage of a superscalar architecture is that compiled application code for a single-issue scalar RISC processor with the same ISA can be executed directly on a superscalar processor. To execute the same application on a

VLIW processor, the original application source code should be recompiled for the new ISA. Because the scheduling logic and the logic to detect ánd omit data dependencies consume quite some area on the die of a superscalar processor as well as a lot of energy, these are less attractive for embedded applications (that require devices as small and energy-efficient as possible).

VLIW architectures require a more powerful compiler than superscalar and Reduced Instruction Set Computer (RISC) architectures, because of the scheduling of operations. As quoted in [1]:

> *"The VLIW design philosophy is to design processors that offer ILP in ways completely visible in the machine-level program and to the compiler."*

This basically means that the hardware is not allowed to perform actions that the programmer cannot directly infer.

In [27], Corporaal presents Transport Triggered Architectures (TTAs) as an alternative to (and an evolution of) VLIW architectures. In a TTA, not the FUs are specified per operation, but the data transports. This implies that unnecessary data transports do not have to take place.

Explicitly Parallel Instruction Computing (EPIC) [22] is the architecture designed cooperatively by Hewlett-Packard and Intel. EPIC strongly relies on VLIW design principles, but the architectural name was changed by Intel for marketing reasons [1] (because enough innovative changes were applied to differentiate the architecture). The first processor that was designed on this architecture was the Intel Itanium. The architectural name was called IA-64 (for Intel Architecture with 64 bit word length, as opposed to IA-32). When the Itanium processor was released, the name was changed to IPF (Itanium Processor Family), again for marketing reasons.

2.3 The VEX VLIW Architecture

The VEX (VLIW Example) ISA [1] is loosely modeled on the ISA of the HP/STMicroelectronics Lx [3] family of embedded VLIW processors. The VEX ISA supports a multi-cluster implementation, where each cluster provides a separate (possibly different) VEX ISA implementation. Each cluster has the ability to issue multiple operations in the same instruction (that is, each cluster acts as a separate VLIW core). A VEX multi-cluster processor shares one instruction fetch unit and one memory controller. The extensibility of the instruction set enables the definitions of special-purpose instructions in an organized way. VEX does not support floating point operations.

2.3.1 The VEX ISA

Figure 2.1 depicts the structure of a default VEX cluster, with an instruction issue-width of 4. By default, a VEX cluster has 4 ALU units, 2 multiplier units, 1 branch control unit and 1 memory access unit per cluster. Also, an instruction- and data-memory cache of 32 kB is present. A VEX instruction consists of one or more syllables, depending on the issue-width. A syllable can be seen as a single 'RISC-style' instruction.

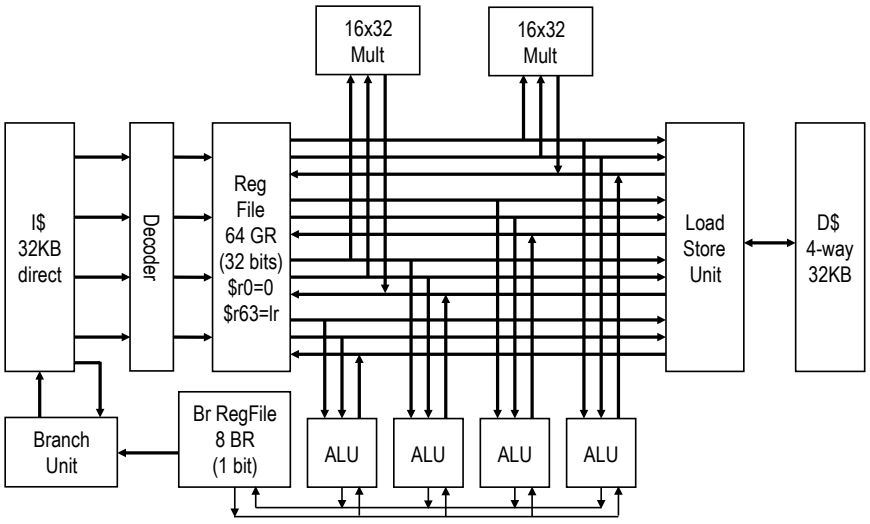


Figure 2.1: Structure of the default VEX cluster [1]

A publicly available VEX software toolchain is provided by Hewlett-Packard Laboratories [8], which offers a VEX C compiler and a VEX simulator. The compiler allows the user/designer to easily adjust parameters of the VEX processor (like the number of clusters and the issue-width). The VEX simulator offers an architecture-level simulator that comes with a set of POSIX-like libraries, a cache simulator, and an Application Programming Interface (API). The simulator is able to output many statistical run-time data of simulated applications.

The VEX C compiler is a derivation of the Lx/ST200 C compiler, itself a descendant of the Multiflow C compiler. It uses trace scheduling as its main scheduling method. Trace scheduling implies that operations will be restructured in order for large 'traces' to appear without branches. Profiling of compiled applications is supported via the GNU Profiler *gprof*.

2.3.2 VEX Machine Configurations

As described in [1], a VEX machine is a highly customizable. Figure 2.2 depicts the structure of a VEX multi-cluster implementation. A well-defined VEX machine should have at least one cluster (cluster 0). For each cluster, the number of resources per instruction (Arithmetic Logic Units (ALUs), Multiplier units (MULs), issue-width and memory ports) can be indicated, as well as the delay per computational element and the number of registers. Figure 2.1 depicts the default structure of a VEX cluster. We can distinguish some basic VEX machine configurations:

- **1-cluster VEX** – A 1-cluster configuration is a default configuration. The machine has one VEX cluster (cluster 0), for which the various parameters can be altered.
- **multi-cluster VEX** – On a multi-cluster machine, cluster configurations can be altered per cluster. This implies that cluster 2 and cluster 3 can both have a

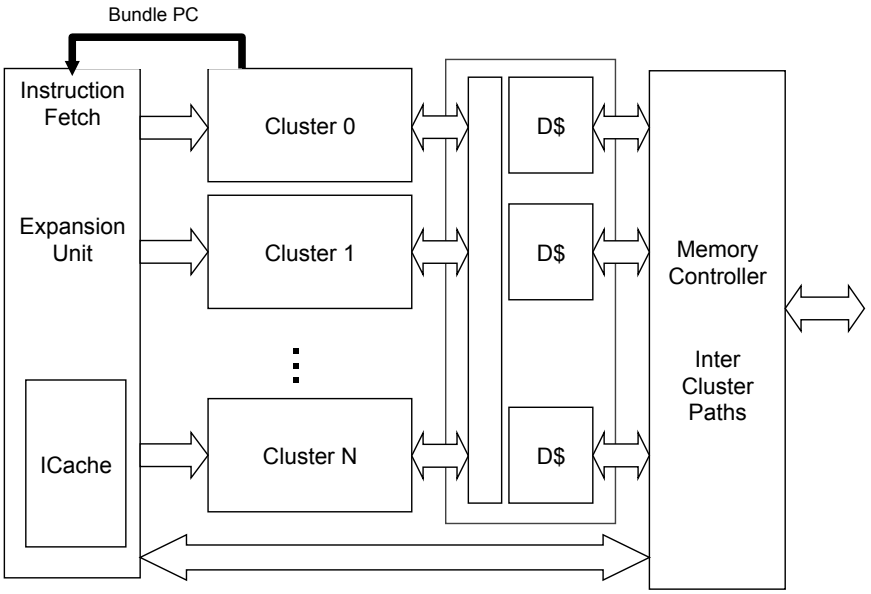


Figure 2.2: Structure of a VEX multi-cluster implementation [1]

different ISA.

- **1-issue (RISC) – VEX** This machine variation is actually a subset of the 1-cluster VEX machine models. The issue-width is 1 by default, so only one ALU is available, one MUL, etc.

2.3.3 Cache Configuration

As well as per-cluster configurations, global configurations like cache sizes and miss penalties can be defined. These configurations are passed as directives to the simulator. Instruction- and data-cache sizes are of significant importance for the performance of a processor. Cache misses can heavily influence the number of executed cycles by introducing stall cycles. The following equations from [28] show us what the influences of cache misses are:

$$\text{CPU time} = (\text{CPU cycles} + \text{Memory stall cycles}) \times \text{Cycle time} \tag{2.1}$$

$$\text{Memory stall cycles} = \text{Instruction count} \times \frac{\text{Memory accesses}}{\text{Instruction}} \times \text{Miss rate} \times \text{Miss penalty} \tag{2.2}$$

It is clear that for applications with many memory accesses, or with badly constructed cache mechanisms (resulting in a high miss rate), the memory stall cycles can account for a significant part of the total number of execution cycles.

As for our research on the ρ-VEX processor, we suggest three configurations for which simulations should be done, in order to define a good design space:

- **Configuration with default cache sizes (DEF)** A default VEX machine has both an instruction- and data-cache of 32 KB. The cache mechanism is organized 4-way set associative.
- **Configuration without cache (NO)** To gain insight in VEX performances where no cache is available, simulations should be done on VEX machines where no cache is available.
- **Configuration with cache trade-off (PROP)** Because memories are expensive to implement in reconfigurable fabric, we want to propose a design that uses less cache memory than the default 32 KB.

These configurations are used in the performance benchmarks in Chapter 3.

2.3.4 The VEX Simulation System

The VEX simulator is a 'compiled simulator', contrary to an 'interpreted simulator'. An interpreted simulator is the most straightforward, but slowest, way of simulating an architecture. The interpreter mimics the target processor, and performs the same actions as the target processor performs on the instructions. Because of the interpretation overhead this is a slow solution (but relatively easy to implement). A compiled simulator translates the target executable binary code to a binary executable that can run on the host system. This removes a lot of the interpretation overhead in a interpreted simulator, and is thus a lot faster.

2.4 The MOLEN Polymorphic Processor

The MOLEN paradigm provides a solution to the growing processor hardware design challenges, by reconfigurable processors (processors that adapt their micro-architecture according to the application's requirements). This is being achieved by Custom Computing Units (CCUs) and reconfigurable microcode ($\rho\mu$-code) [6].

Figure 2.3 presents an overview of the MOLEN machine organization, including a General-Purpose Processor (GPP) with a fixed instruction set, and a reconfigurable co-processor. Application code is executed on the GPP, except for selected functions that were identified to have a very efficient hardware implementation. Those are executed within the reconfigurable processor, by a CCU. The $\rho\mu$-unit takes care of the control flow within the reconfigurable processor.

The process of application generation and execution for a MOLEN machine is as follows [5]:

1. From a given application source code, one or more pieces should be determined and isolated for execution on a hardware CCU. These are being determined by high-level to high-level instrumentation within the Delft Workbench [29] toolchain, and benchmarking. This results in a set of candidate code pieces.
2. From these code pieces, it has to be determined which pieces are suitable for hardware execution, by means of the effort it takes to map them onto hardware.

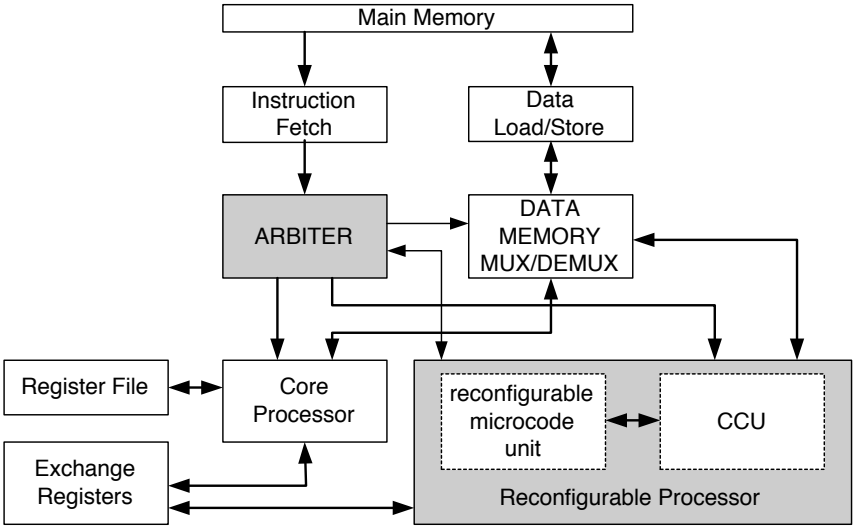


Figure 2.3: The MOLEN machine organization [2]

3. A hardware implementation in an Hardware Description Language (HDL) should be generated either automatically, or manually (for timing-critical parts).

4. Calls to these particular hardware mapped functions should be replaced in the software code by calls to the hardware unit. Data exchange between the hardware units and the general-purpose processor is done by using exchange registers (XREGS).

5. Upon execution of the application, the reconfigurable fabric should be configured (or reconfigured) with the corresponding functions in time.

The design and implementation of new concepts for the MOLEN paradigm are very active topics at the Computer Engineering Laboratory of Delft University of Technology, ever since its conceptual introduction in 2001 [6].

A prototype [7] of the MOLEN processor is currently available, based on the Xilinx Virtex-II Pro platform. This FPGA platform features two PowerPC 405 general-purpose CPU cores embedded in the reconfigurable fabric. One PowerPC 405 core running at a clock speed of 300 MHz serves as the GPP in the current prototype. This prototype is the platform at which the ρ-VEX co-processor is targeted at. ρ-VEX should be implemented as a CCU for the MOLEN processor, as a part of the reconfigurable processor. The application code which is targeted for execution by ρ-VEX will be directed to the VEX compiler, where binary executable code will be generated. Figure 2.4 depicts the workflow for a MOLEN machine featuring an ρ-VEX co-processor. An application kernel identifier as part of the Delft Workbench toolchain identifies a code fragment to be executed by a CCU (ρ-VEX). This code is then passed to the VEX compiler by Hewlett-Packard, instead of being transformed into an application-specific hardware solution. The resulting VEX assembly code will be assembled by ρ-VEX' accompanying assembler, ρ-ASM. The ρ-VEX executable is then loaded into the instruction memory

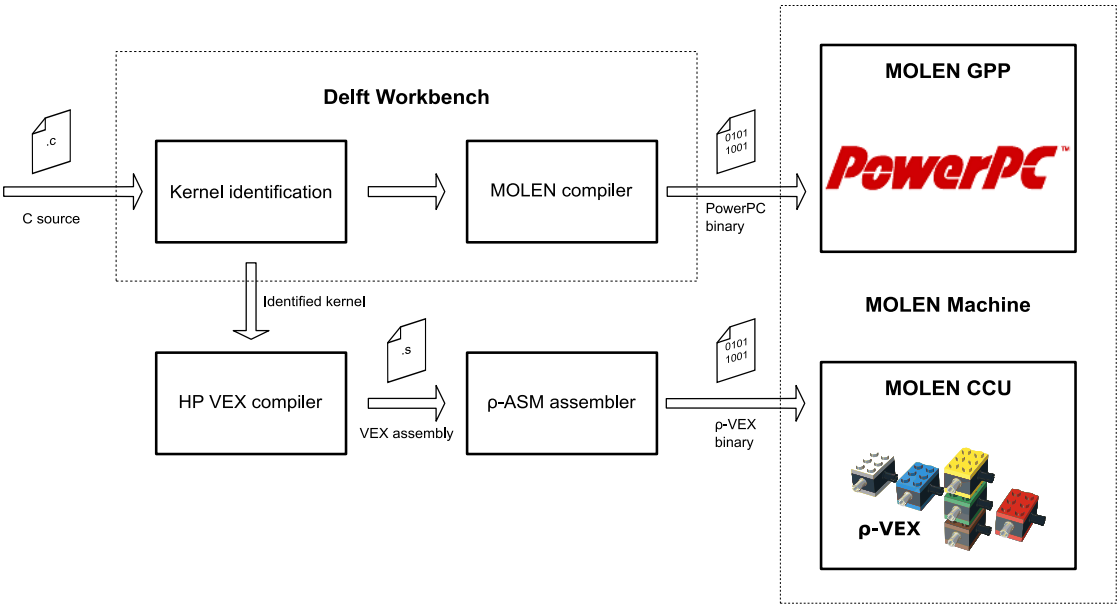


Figure 2.4: ρ-VEX integration within the MOLEN workflow

of ρ-VEX. When ρ-VEX receives that CCU 'start' signal, it will execute the application kernel. ρ-VEX raises CCU 'finished' signal when the execution has finished.

2.5 Conclusions

This chapter presented a background on the technologies used in our research. Different softcore approaches by others resulted in designs and implementations of several processors. These approaches vary between relatively fixed designs with extensive toolchains, to flexible designs with weak or no toolchains. Some approaches provided support for reconfigurable operations, through different mechanisms.

Instruction Level Parallelism (ILP) is exposed within an application when a computer system is able to execute multiple different operations, when a single stream of instructions is presented. Superscalar and VLIW are architectures that are able to exploit ILP by issuing more than one operation per issue slot. These operations are issued to additional Functional Units (FUs) available within the processor. Superscalar machines demand dynamic operation scheduling logic in hardware, while operations are pre-scheduled by the compiler for VLIW machines.

The VEX architecture is designed at Hewlett-Packard, based on the Lx VLIW processor architecture. The VEX ISA supports multi-cluster computing machines with a variable issue-width. A compiler and simulator toolchain are made freely available by Hewlett-Packard.

The MOLEN paradigm presents a reconfigurable processor architecture, consisting of a General-Purpose Processor with a fixed instruction set and a reconfigurable co-processor. Custom operations are placed in the co-processor as Custom Computing Units. The co-processor is instructed and configured by reconfigurable microcode ($\rho\mu$-

code) operations. An arbiter decodes the instructions to decide whether to target the reconfigurable processor or the fixed core processor.

Performance and Configuration Analysis

3

In order to give more insight in the performance gain the ρ-VEX co-processor would give to a MOLEN machine, we performed a preliminary performance and configuration analysis. We also analyzed what a good ρ-VEX configuration would be when it serves as a co-processor within a MOLEN machine.

Our performance analysis is divided in two parts. The first part consists of an analysis of the commercially available Lx processor, since the VEX architecture is a descendant of the Lx architecture. The second part consists of an analysis of custom benchmarks executed on different architectures: inside the VEX simulator using different machine models, on a PowerPC 405 processor and on a MOLEN hardware CCU when possible.

In Section 3.1, the performance of the Lx processor is evaluated. Section 3.2 describes the benchmarks and results obtained from our own simulations on the aforementioned architectures. Conclusions are drawn in Section 3.3.

3.1 Lx Analysis

Because the VEX ISA is very similar to the Lx ISA by Hewlett-Packard and STMicroelectronics, we considered the performance measurements of Lx based processors with different ISA configurations to be valuable. In [3], performance measurements have been done on different Lx machine configurations against a cycle-accurate simulator. The benchmarks used are a subset of the SPECINT'95 suite, consisting of application-specific benchmarks, as well as general-purpose benchmarks. Table 3.1 presents an overview of the performed benchmarks. All application-specific benchmarks (except *adpcm*) were optimized with source-level compiler pragmas, as well as code restructuring to expose more Instruction Level Parallelism (ILP). Figure 3.1 depicts performance charts for an Lx processor running on a clock frequency of 300 MHz[1]. The cluster-width of the Lx processor varies between 1,2 and 4 clusters (corresponding to an issue-width of 4, 8 or 16 operations). Performances are compared to an Intel Pentium II running at 333 MHz, with a performance scale of 1.00.

These results clearly show that specializing for an application domain pays off in terms of performance gain in contrast to general-purpose applications. This is expected behaviour, as general-purpose applications are not able to expose a lot of ILP. The general-purpose results are of less interest for our research, as ρ-VEX is intended to be an embedded co-processor within a MOLEN machine, dedicated to execute application-specific kernels. The general-purpose code will be executed by the GPP inside the

[1] In [3], performance of 200 and 400 MHz Lx processors was also measured, and all results were compared to a 275 MHz StrongARM processor. These have been left out because of no significant importance.

Application-specific		**General-purpose**	
Name	**Description**	**Name**	**Description**
bmark	Printing imaging pipeline	*boise*	Printing rendering pipeline
copymark	Color copier pipeline	*dhry*	Dhrystone 1.1 and 2.1
crypto	Cryptography code	*gcc*	GCC
csc	Color-space conversion	*go*	GO
mpeg2	MPEG-2 decoder	*li*	LISP interpreter
tjpeg	JPEG-like codec	*m88ksim*	M88000 simulator
adpcm	ADPCM audio codec	*gs*	Ghostscript PS interpreter

Table 3.1: The Lx benchmark set taken from SPECINT'95 [3]

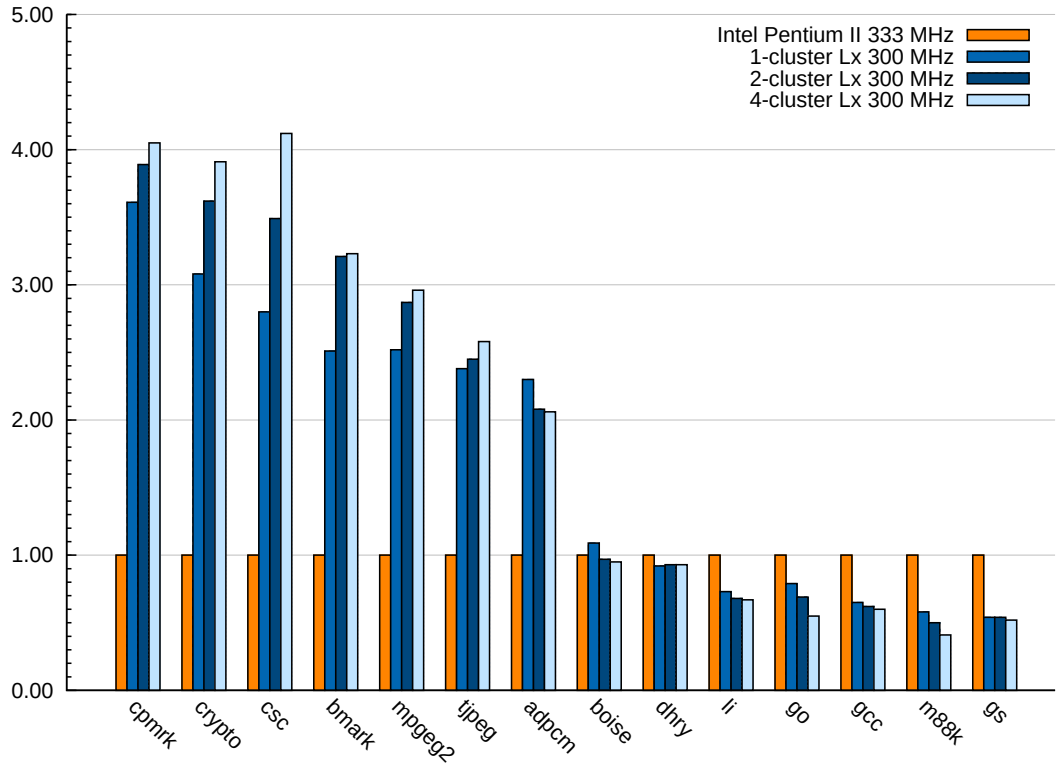


Figure 3.1: Lx performance chart, varying cluster-width from 1 to 4 [3]

MOLEN machine. However, these benchmarks show the negative impact of general-purpose applications on a VLIW architecture when ρ-VEX serves as a GPP itself (in a stand-alone environment).

Except for the *adpcm* benchmark, increasing the issue-width of the Lx processor improves the performance. This heavily depends on the application. Both the largest and the smallest performance gain can be found in the *bmark* benchmark: going from 1

to 2 clusters (issue-width increases from 4 to 8) yields a performance gain of about 28%. Contrary, the gain from 2 clusters to 4 clusters can be neglected. Generally speaking, it can be concluded that the increase from 1 to 2 clusters has a bigger influence on the achieved performance than scaling from 2 to 4 clusters.

3.2 VEX Analysis

To gain more insight in the performance of ρ-VEX processor we performed benchmarks on the VEX ISA. To investigate the performance and real-world pay-off of different VEX configurations we performed the benchmarks on different machines:

- **RISC-VEX machine** – A VEX machine in RISC configuration equals a VEX machine with an instruction issue-width of 1. By default, a VEX cluster has an issue-width of 4. We performed the benchmarks on a RISC-VEX configuration, because this would clearly show the advantage of a higher issue width when the results are compared to a 1- or 2-cluster VEX machine.
- **1-cluster VEX machine** – The single cluster in this machine has a default configuration concerning computation resources.
- **2-cluster VEX machine** – Both clusters of this machine have default resources configured. Because our original intentions were to design either a 1- or 2-cluster ρ-VEX processor, this machine is included in the set.
- **PowerPC 405 at 300 MHz** – The PowerPC is one of the direct 'competitors' of ρ-VEX, because it is the GPP inside the current MOLEN prototype. ρ-VEX has to perform at least better than this processor to expose advantages.
- **MOLEN hardware CCU** – The other 'competitor' of ρ-VEX is a MOLEN hardware CCU. The performance of ρ-VEX is targeted to be in between the performances of the PowerPC and a CCU.

It is clear that we chose not to benchmark a 4-cluster VEX machine. There mainly are two reasons for this: a 4-cluster VEX machine is beyond the scope of the ρ-VEX project, and the performance gain of a 4-cluster compared to a 2-cluster Lx machine is minimal, as was shown in Section 3.1. We did not perform benchmarks including custom operations as it was initially not planned to have support for custom operations in ρ-VEX.

We performed all benchmarks with the VEX machines in three different configurations, the ones presented in Section 2.3.3: the default (DEF) VEX cluster configuration (32 KB 4-way set associative instruction- and data-cache), the 'no cache' (NO) configuration, and the proposed (PROP) configuration. Currently, in the 3.41 version of the VEX toolchain, we could not de-activate the cache system totally for simulation purposes. We were however able to bring it down to a 16 byte, direct mapped cache. This resulted in a 16 B direct mapped instructio- and data-cache organization for the 'no cache' configuration. For te proposed configuration, a 4 KB 4-way set associative instruction- and data-cache seemed to be a good trade-off, performance-wise.

Except for performance, ρ-VEX also competes area-wise with other CCUs, because they are both configured in the same reconfigurable fabric. A user may opt for an instantiation of the ρ-VEX co-processor in the reconfigurable fabric of the MOLEN machine, or save area to have more CCUs. ρ-VEX is supposed to perform 'reasonably' on many application-specific kernels, whereas a CCU is supposed to perform 'considerably' on one specific task. When making this trade-off, the user has to take the number of application-specific kernels in the particular application into account, as well as the relative speedup for a CCU compared to ρ-VEX and to the PowerPC 405 processor. It should be noted that the current MOLEN prototype is configured without I- and D-cache for the PowerPC 405 processor, because of limitations in the used FPGA technology.

3.2.1 Simulator Benchmark Set

Our benchmark set consisted of four benchmarks that will be discussed below. Two of our benchmarks are modified examples from the MOLEN prototype example set [30]. This means that a representable hardware CCU is available and we can have benchmark results from all machines. The other two kernels are chosen within the (media) application domain, because ρ-VEX is meant to excel in this domain.

- **G.723 Audio Encode** The G.723 [31] audio encoding technique is used for the encoding of (voice) audio in Voice over IP (VoIP) telephony applications. For this benchmark, an 8-bit A-law input audio signal of 928 bytes is encoded to a 24 kbps G.723 encoded signal. The G.723 implementation used in our project was released to the public domain by Sun Microsystems [32]. This benchmark was chosen because it is a real-world application in the domain where the computing machine is targeted at.

- **Matrix Multiplication** We created an application which multiplies two 64×64 semi-randomly integer-filled matrices. We created this application because both ALU parallelism would be exposed, as well as prefetching possibilities.

- **Min/Max from Array** This application resides within the MOLEN prototype examples set. It determines the minimum and maximum values of a semi-randomly filled array with 16 integers (the unmodified demo application operates on an array with 8 integers). This benchmark exposes a lot of branches and comparisons.

- **Moving Filter** This last application also originates from the MOLEN prototype examples set. It transfers a signal represented by a semi-randomly filled array of 32 integers through a filter (the unmodified demo application operates on an array with 16 integers). This benchmark was chosen because it showed a lot of parallelism in terms of ALU usage.

The VEX simulator generates extensive log files after each simulation, of which we were able to obtain the cycle counts of the different VEX machines. To obtain the number of execution cycles for the PowerPC 405, we used the internal cycle counter of the PowerPC. We obtained the cycle count with inline assembly lines. The number of execution cycles for the MOLEN CCU are relative to the PowerPC. We let the PowerPC

	G.723			Matrix Multiplication		
	DEF	**NO**	**PROP**	**DEF**	**NO**	**PROP**
VEX RISC	2108515	10782609	3363355	2320266	24803442	12337487
VEX-1	1148287	9871235	2442257	1277970	22492006	11431185
VEX-2	1118928	10314585	2244582	1023833	22201002	11413672
PPC-EDK	17213803			27015717		
PPC-MOLEN	N/A			N/A		
CCU	N/A			N/A		

	Min/Max from Array			Moving Filter		
	DEF	**NO**	**PROP**	**DEF**	**NO**	**PROP**
VEX RISC	1017	3789	1332	1944	14458	2349
VEX-1	940	3878	1255	1517	14468	1922
VEX-2	976	3797	1291	1347	12929	1783
PPC-EDK	2170			6942		
PPC-MOLEN	2754			7530		
CCU	360			558		

Table 3.2: Number of clock cycles executed per system in the VEX benchmark set

start counting cycles at the call to a CCU function, and we stopped the counter after obtaining a result.

All benchmarks have been performed with the compiler set to medium optimization. Because the MOLEN prototype *GCC*-based compiler 2.0α (PPC-MOLEN) currently only compiles with low optimization, we repeated those benchmarks with the *GCC*-based PowerPC compiler that is bundled with the Xilinx Embedded Development Kit (EDK) 8.1i (PPC-EDK).

3.2.2 Simulator Benchmark Results

The results of our benchmarks are presented in Table 3.2. Since the different cache configurations only apply to the VEX machines, the PowerPC and MOLEN CCU results are only presented once per benchmark. Figure 3.2 depicts the performance chart based on default VEX configurations. The performance in clock cycles of the PowerPC 405 processor (with application code compiled by the *GCC* compiler included with EDK) is used as reference. Subsequently, Figure 3.3 depicts the performance chart based on the 'no cache' configurations. Finally, Figure 3.4 depicts the performance chart based on the proposed cache configurations.

It can be seen from the charts that cache size heavily influences the results of the VEX machines. The biggest differences can be seen in the Matrix Multiplication benchmark. With default cache sizes, a 2-cluster VEX machine outperforms the PowerPC processor

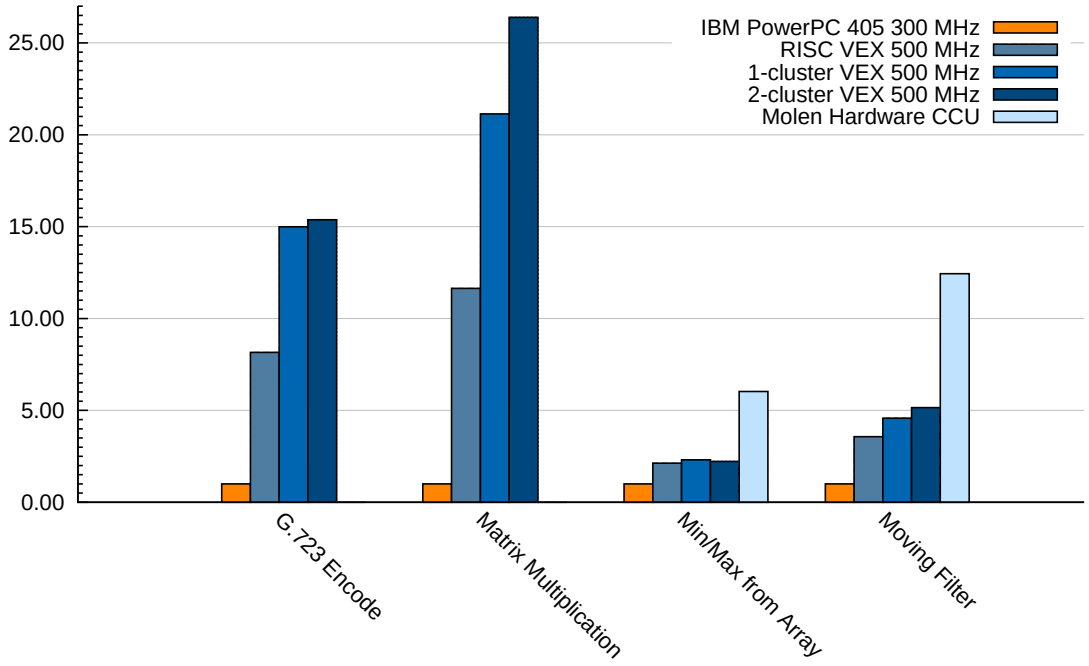


Figure 3.2: VEX performance chart based on default cache configurations (DEF)

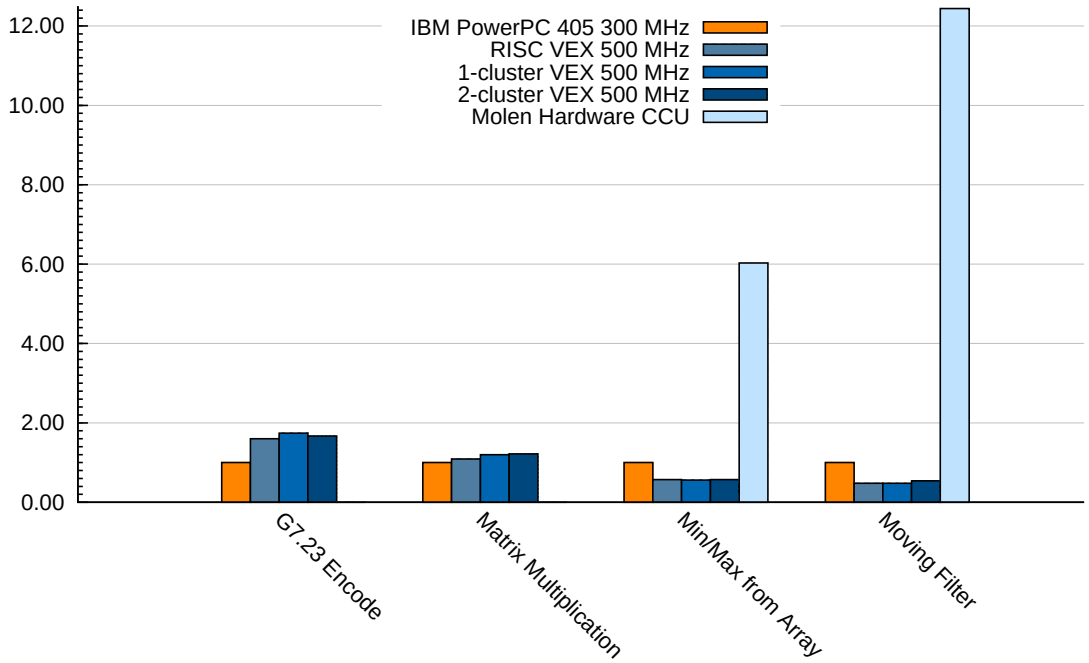


Figure 3.3: VEX performance chart based on 'no cache' configurations (NO)

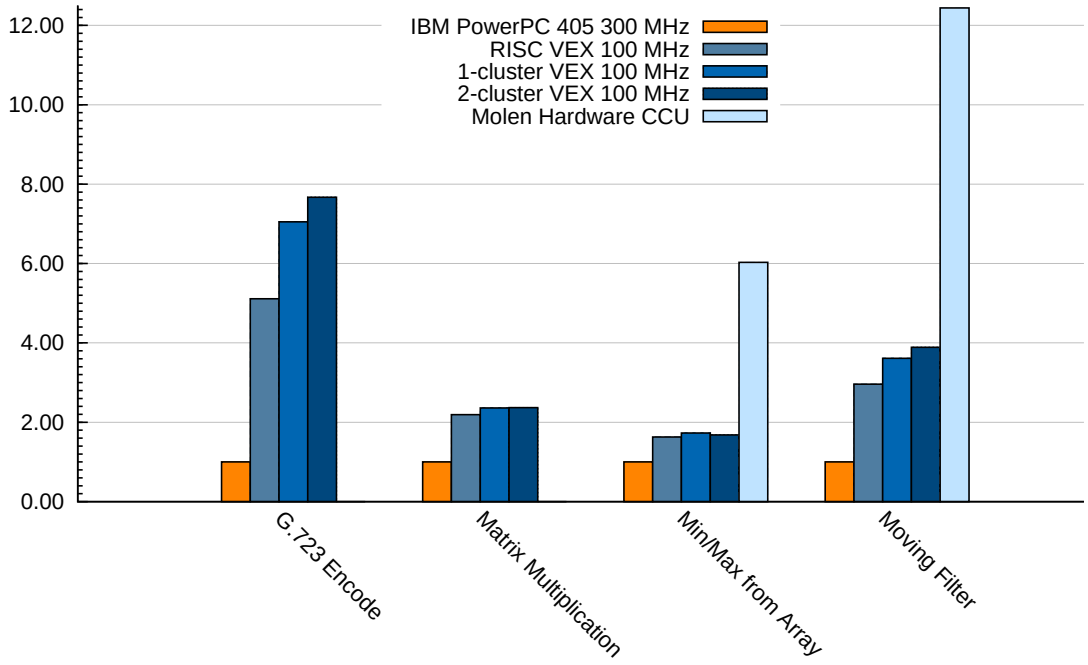


Figure 3.4: VEX performance chart based on proposed cache configurations (PROP)

by a factor of 27. In contrast, when data- and instruction-cache is used, the PowerPC and the VEX machines perform equally. When the proposed cache sizes of 4 KB are applied, the VEX machines outperform the PowerPC by a factor 2 approximately.

When looking at the performance difference between a VEX machine in RISC configuration and a 1-cluster VEX machine (in the default cache configuration benchmarks), we see performance improvements ranging from 10% to 100%. When the differences between a 1-cluster VEX machine and a 2-cluster VEX machine are observed, we see improvements ranging from -5% to 20%.

When no cache is used, we can see that VEX machines perform worse than the PowerPC in the Min/Max and Moving Filter benchmarks. The proposed configuration performs well, considering its relative small cache sizes. The 1-cluster VEX machine and the RISC VEX machine even outperform their counterparts in the default configuration with 8 times the amount of cache memory available in the Matrix Multiplication benchmark. Generally speaking, the performance gain of the VEX machines with 4 KB instruction- and data-cache ranges between 80% and 700% compared to the PowerPC.

It should be kept in mind that in particular the Matrix Multiplication and the Min/-Max benchmarks only represent a very small part of a real-world application. A speedup of the full application is heavily influenced by Amdahl's law:

$$S_i = \frac{T}{T - T_{PPCi} + T_{VEXi}} = \frac{1}{1 - (a_i - \frac{a_i}{s_i})} \tag{3.1}$$

In (3.1), S_i is the total speedup of the application, T is the total amount of execution cycles needed of the application in software only, T_{PPCi} is the amount of execution cycles of the application kernel in software, T_{VEXi} is the amount of execution cycles

of the application kernel on a VEX machine. a_i is the percentage of time used by the application kernel in software ($\frac{T_{PPCi}}{T}$), and s_i is the speedup of the application kernel on a VEX machine compared to software execution ($\frac{T_{PPCi}}{T_{VEXi}}$).

3.3 Conclusions

In this chapter we first presented performance benchmarks done in earlier work for the Lx processor family. Benchmarks were performed with 1-, 2- and 4-cluster Lx machines running at 300 MHz compared to an Intel Pentium II running at 333 MHz. The results showed significant performance gains in application-specific benchmarks for the Lx configurations compared to the Pentium II configuration. For general-purpose benchmarks, the performance was worse for Lx configurations than for the Pentium II configuration. When looking at the differences per cluster-configuration, we see that increasing the cluster-width from from 1 to 2 results in performance gains of about 28%. Increasing the cluster-width to 4 has a negligible effect in most benchmarks.

We performed benchmarks on three different VEX machine configurations, a PowerPC 405 processor (within a Xilinx Virtex-II PRO FPGA), and a MOLEN hardware CCU (when applicable). For the VEX machine configurations we used a 1-issue, 1-cluster RISC machine, a standard 1-cluster machine, and a 2-cluster machine. We used a custom coded matrix multiplication benchmark, two benchmarks from the MOLEN example set, and a G.723 audio encoder. To measure performances on the VEX machines, the VEX simulator by Hewlett-Packard was used with different cache configurations. The benchmarks showed that the VEX machines scored in-between the PowerPC 405 and a MOLEN CCU performance-wise, as we expected. Furthermore, the performance gain from a 1-cluster to a 2-cluster configuration was again negligible in many cases. These benchmarks resulted in the decision to first implement a 1-issue RISC ρ-VEX configuration, followed by a 1-cluster 4-issue standard configuration ρ-VEX.

4 Design

ρ-VEX was designed to be an extensible and parametric processor from the ground up. This led to a modular design and an instruction layout that exposes enough freedom to extend the standard set of VEX operations. The design is based on a 1-cluster VEX machine with a default configuration.

Section 4.1 presents the ρ-VEX organization. The used instruction layout is presented in Section 4.2. Subsequently, the extensibility of ρ-VEX is discussed in Section 4.3. This chapter is concluded in Section 4.4.

4.1 Organization

The design of ρ-VEX is based on a Harvard architecture, which defines physically separated memories for program instructions and data. This implies that the widths of data busses may differ per memory type. This is especially useful for VLIW architectures, because we want to issue very wide words from instruction memory. This contrasts a von Neumann architecture that defines one memory structure where both program code and data reside.

A four-stage design consisting of *fetch*, *decode*, *execute*, and *writeback* stages was used for ρ-VEX. The standard configuration of a 1-cluster VEX machine was used for the default configuration. This implies the availability of four Arithmetic Logic Units (ALUs), two Multiplier units (MULs), one Control unit (CTRL), one Memory unit

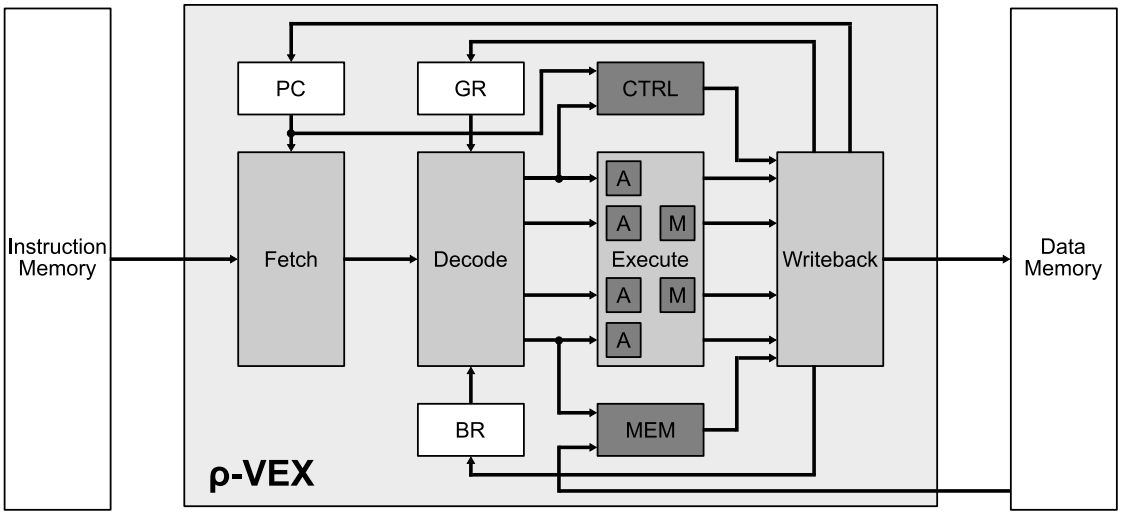


Figure 4.1: ρ-VEX organization (4-issue)

Opcode	Class	VEX	ρ-OPS
1------	ALU	42	10
000----	MUL	11	4
010----	CTRL	11	5
001----	MEM	9	5

Table 4.1: Opcode space distribution

(MEM), a General-purpose Register (GR) file with 64 32-bit registers and a Branch Register (BR) file with 8 1-bit registers. We decided not to design instruction- and data-memory caches for our prototype, because the memory would be on-chip. Appendix C presents a more detailed overview of the ρ-VEX machine model.

Figure 4.1 depicts the organization of a 4-issue ρ-VEX processor. The *fetch* unit fetches a VLIW instruction from the attached instruction memory, and passes it on the *decode* unit. In this stage, the instruction is being split into syllables. Also, the register contents used as operands are fetched from the register files. The actual operations take place in either the *execute* unit, or in one of the parallel CTRL or MEM units. ALU[1] and MUL operations (respectively, A and M in Figure 4.1) are performed in the *execute* stage. This stage is designed parametric, so that the number of ALU and MUL functional units could be adapted. ρ-VEX should have exactly one (by definition of the VEX ISA) CTRL and MEM unit, so these units are designed outside the parametric *execute* unit. All jump and branch operations are handled by the CTRL unit, and all data memory load and store operations are handled by the MEM unit. To ensure that all results to the GR and BR registers, external data memory and the internal Program Counter (PC) are written at the same time per instruction, all write activities are performed in the *writeback* unit.

4.2 Instruction Layout

The standard set of VEX operations consists of 73 operations (excluding **NOP** – no operation). Opcodes for the two inter-cluster operations (**SEND** and **RECV**) described by the VEX ISA are reserved, but not used as ρ-VEX (currently) supports only 1-cluster VEX machine configurations. We complemented this default set of operations with two extra operations: **STOP** and **LONG_IMM**. The former operation tells ρ-VEX when to stop fetching instructions from the instruction memory. The latter is used when *long immediate* operands are handled. To assign a unique opcode to every operation, at least 7 bits are needed to encode the opcodes. Table 4.1 presents the opcode space distribution per functional unit, and the number of operations the VEX ISA describes. A few exceptions to this scheme are discussed later, as well as the meaning of the values in the ρ-OPS column.

As VEX supports up to 64 GR registers, 6 bits are required to address them all. Up to 8 BR registers are supported, so 3 bits are needed to address them. To be able to

[1]Both integer arithmetic operations as well as logical, compare and select operations are performed by ALU units.

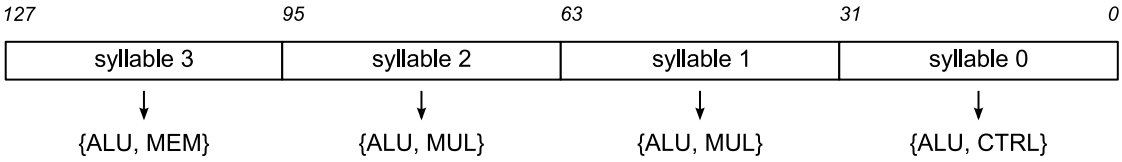


Figure 4.2: Instruction layout

Immediate type	Size	Immediate switch
no immediate operand	N/A	00
short immediate	9 bit	01
branch offset immediate	12 bit	10
long immediate	32 bit	11

Table 4.2: Immediate types

fit opcode bits, register addresses bits and syllable meta-data bits in one syllable, 32 bit syllables are used.

An instruction, by default consisting of four syllables, is a concatenation of all syllables (even **NOP** syllables), with syllable 0 starting at bit 0. Figure 4.2 shows the instruction layout. So the default instruction size is 128 bit. This is not very efficient, and there are many possible instruction packing/compression techniques to reduce the instruction size. As this was not our primary concern for the current ρ-VEX design, we left this unoptimized. Even VEX' **XNOP** operation, which issues multiple **NOP** operations, is implemented as a regular **NOP** operation.

Figure 4.2 also shows what syllables are able to issue operations on the various functional units. As a standard VEX cluster contains 4 ALU units, all syllables are able to issue an ALU operation. To use the functional units optimally, the other operation classes are distributed evenly among the syllables. Syllable 0 is able to issue CTRL operations, syllables 1 and 2 are able to issue MUL operations and syllable 3 is able to issue MEM operations.

Appendix B presents an overview of all VEX operations together with their semantics.

The VEX standard defines the use of three types of immediate operands: 9 bit *short immediate* operands, 24 bit *branch offset immediate* operands and 32 bit *long immediate* operands. The first two types are embedded in a single syllable, but the last one is spread over more syllables. For ρ-VEX, we decided to change the size of a *branch offset immediate* operand to 12 bit. This was done in order to use our syllable layout templates more efficient. Every syllable has an *immediate switch* field consisting of 2 bits that describe the type of immediate operand that the operation affects. Every VEX ALU/MUL operation is overloaded to support both register operands as well as immediate operands by default. Table 4.2 presents an overview of all immediate types and the corresponding value of the *immediate switch*.

Figure 4.3 depicts the syllable layout templates that we designed for ρ-VEX. The shaded bit-fields shows the content of the *immediate switch*. All variations of this bit-field are depicted in the figure. Not all fields are evaluated in all cases. For example,

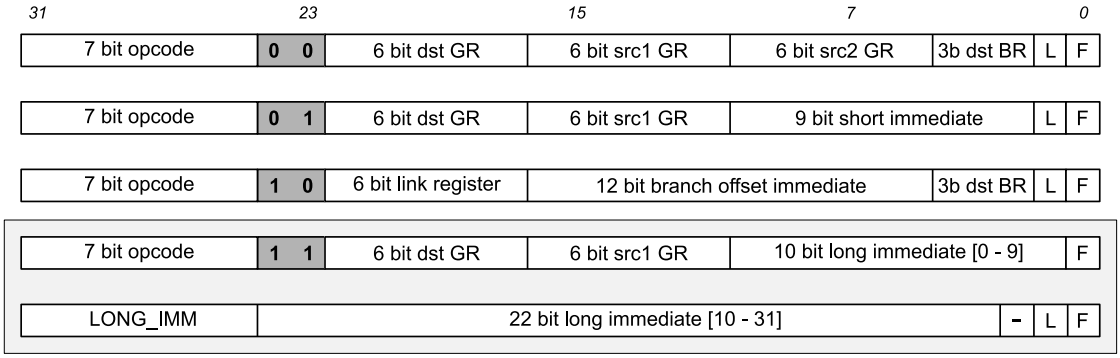


Figure 4.3: Generic syllable layout

when an ALU operation has no BR destination operand, the *3 bit dst BR* field holds 'don't care' values.

ρ-VEX syllables include two bits with syllable meta-data, the L and F bits. The L bit denotes whether a syllable is the last syllable of an instruction and the F bit denotes whether it is the first syllable in an instruction. The VEX standard defines such meta-data to be available in every syllable. In the current ρ-VEX prototype these fields are not used, but these bit-fields allow the implementation of a more sophisticated syllable packing mechanism (instructions with variable length can be evaluated this way).

Special attention should be paid to the *long immediate* syllables. As a *long immediate* operand is always spread over two syllables, a syllable with opcode **LONG_IMM** could not occur without a preceding syllable where the *immediate switch* is set to `11` (nor can a *long immediate* operand be issued in a 1-issue configuration). As the first syllable of a *long immediate* operand could not be the last syllable in an instruction, this syllable does not provide an L field. It should be noted that the current implementation of ρ-VEX does not support the utilization of *long immediate* operands.

4.2.1 ALU and MUL Syllables

In principle, all ALU and MUL syllables fit in the syllable templates in Figure 4.3. All logical and select ALU operations (types II and IV as defined in Appendix B) can have a GR register or a BR[2] register as a destination operand. To decide which destination register should be targeted, the GR destination address field is evaluated in the *decode* unit. When the GR destination address equals `$r0.0` (which is hardwired to zero/ground), the BR destination address is used to store the result of the operation.

The **ADDCG**, **DIVS**, **SLCT** and **SLCTF** ALU operations operate on three source operands: two GR register operands, and one BR register operand. Because some of these

[2]In the current ρ-VEX implementation, immediate operands are not supported when the operation is targeting a BR register. This is caused by the fact that part of the immediate operand field is used by the BR destination address field. A possible way to solve this could be the utilization of the *immediate switch* for *branch immediate* operands when a BR register is the target destination, and for *short immediate* or *long immediate* operands when a GR register is the target destination. All destination addresses (GR and BR) could then be placed in the GR destination address field. In case of a *long immediate* operand in a BR-targeting operation, one of the extra bits in the GR address field could be set.

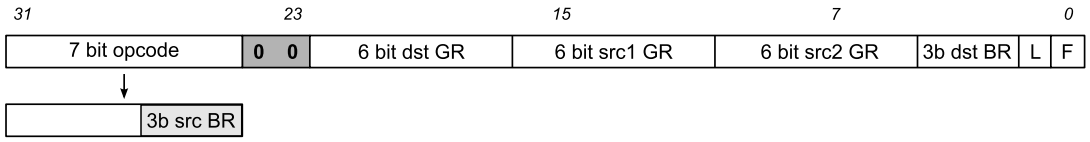


Figure 4.4: Syllable layout for **ADDCG** and **DIVS** syllables

operations are also able to operate on either a GR register or BR register as destination, a new location for the BR source register address should be assigned. We chose opcodes for these operations, so that the 4 most significant bits are unique. The least significant bits of the opcode field are used in this case to pack the BR source address. Figure 4.4 depicts this technique of source address packing as well as the syllable layout for **ADDCG** and **DIVS** syllables. Because these operations do not support immediate operands, there is no problem that all bit-positions are always in use.

SLCT and **SLCTF** have similar layouts, but do not need a destination BR address. These operations do support immediate operands. These are handled in the same way as immediate operands are handled by other operations according to Figure 4.3. Because there was not enough space left in the ALU opcode region (Table 4.1), **SLCT** and **SLCTF** have opcodes outside the standard ALU region.

4.2.2 CTRL Syllables

Figure 4.5 depicts the syllable layout for the overloaded[3] **GOTO** and **CALL** syllables. Either one of the layouts is issued, depending on the assembly usage. As **GOTO** and **CALL** are overloaded with **IGOTO** and **ICALL** functionality, the assembler should issue one of these latter operations when the operation is called with a link register address. An unconditional absolute indirect jump to the *branch offset immediate* (or link register contents) is performed when a **GOTO** (or **IGOTO**) operation is issued. When a **CALL** or **ICALL** operation is issued, the same jump is being performed, but the link register is updated with the current PC value incremented by 1.

Figure 4.6 depicts the syllable layout for **BR** and **BRF** syllables. When the branch condition evaluated on *3 bit src BR* are evaluated as true (**BR**) or false (**BRF**), the Program Counter will be updated with the *branch offset immediate*.

Figure 4.7 depicts the syllable layout for **RETURN** and **RFI** syllables. **RETURN** and **RFI** pop the stack frame (the stack pointer residing in `$r0.1`) and jump to the contents inside the link register.

4.2.3 MEM Syllables

Figure 4.8 depicts the syllable layout for memory load syllables, and Figure 4.9 for memory store syllables. It should be noted that the source address for the source GR

[3]According a to a message on the VEX forum, the **CALL** & **ICALL** and **GOTO** & **IGOTO** operations are overloaded in the latest VEX compiler, so **ICALL** and **IGOTO** are obsolete. These are still supported in its original form to stay compliant to the original VEX ISA.
See `http://www.vliw.org/vex/viewtopic.php?t=52` for more information.

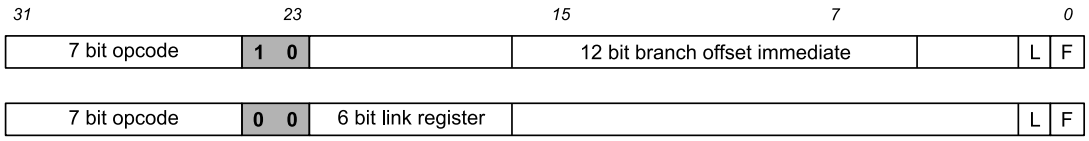


Figure 4.5: Syllable layout for **GOTO** and **CALL** syllables

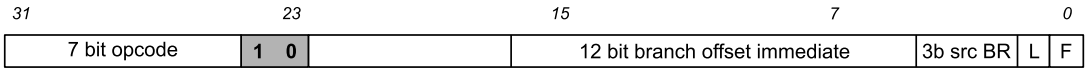


Figure 4.6: Syllable layout for **BR** and **BRF** syllables

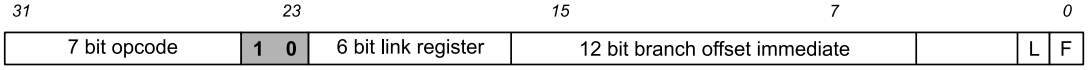


Figure 4.7: Syllable layout for **RETURN** and **RFI** syllables

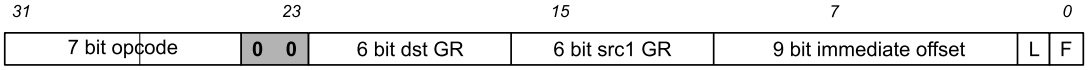


Figure 4.8: Syllable layout for memory load syllables

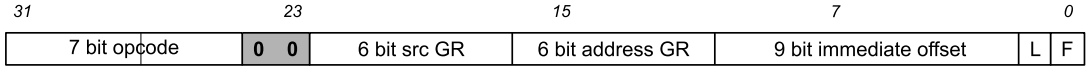


Figure 4.9: Syllable layout for memory store syllables

register in a store operation is located in the field where normally the destination GR register address resides. The GR register that contains the address in memory where the source register should be written to, resides in the *6 bit address GR* field.

It should be noted that both load and store operations use an *immediate switch* that indicates no immediate operand, although there is an immediate operand present. This originates from some implementation issues when dealing with two source GR register addresses, as well as an immediate operand.

4.2.4 Other Syllables

Figure 4.10 depicts the syllable layout for **NOP** syllables, and Figure 4.11 for **STOP** syllables. Apart from the opcode and first/last syllable bits, the other bits are of no importance for these operations.

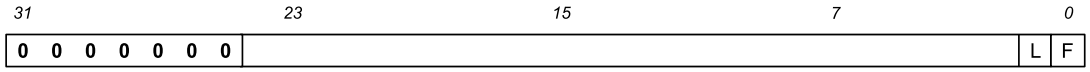


Figure 4.10: Syllable layout for **NOP** syllables

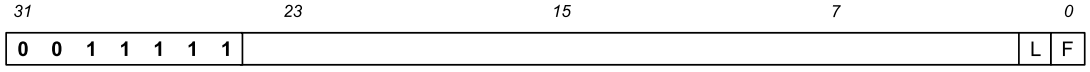


Figure 4.11: Syllable layout for **STOP** syllables

4.3 Extensibility

ρ-VEX extensibility is provided by two mechanisms, namely ρ-OPS and VEX machine models. ρ-OPS are custom reconfigurable operations that can be easily added to the execution model of ρ-VEX. VEX machine models define among others the configuration of FUs, register files, and the issue-width.

4.3.1 ρ-OPS

The VEX software toolchain supports the use of custom instructions via pragmas inside the application code, as described in [1]. With ρ-OPS we provide a mechanism to execute these operations on the ρ-VEX processor. The good thing about ρ-OPS is that they are not restricted combinatorial operators. Sequential ρ-OPS consisting of multiple atomic operations are also allowed, as long as the design still gets properly synthesized and routed. This depends strongly on the reconfigurable implementation technology used, and is thus not architecturally bounded. Our benchmark in Chapter 7 also uses sequential ρ-OPS.

In the current ρ-VEX implementation, it takes about 5 lines of VHDL code to add a custom operation to the architecture. One of the available ρ-OPS opcodes as listed in Appendix B should be chosen, and a template VHDL function should be extended with the functionality of the operation. The Quickstart Guide in Appendix F presents a practical example of using ρ-OPS.

Because 4 ALU operations used extra opcode space for BR address packing (as explained in Section 4.2.1), there are 24 opcodes available for custom reconfigurable operations, ρ-OPS. Table 4.1 presents the number of ρ-OPS available per operation

class in the last column. Syllables for a reconfigurable ρ-OPS operation should use the same layout as standard VEX operations within the corresponding class for an easy implementation. It is possible to implement ρ-OPS within a certain opcode space that does not use the corresponding functional unit. Then, some extra logic should be added in the *decode* stage implementation to recognize these operations and handle the correct syllable layout.

4.3.2 VEX Machine Models

A large amount of parametric options from the VEX machine models used by the VEX compiler can be used to parameterize ρ-VEX. Currently, the following properties of ρ-VEX are parametric:

- Issue-width (should be a power of 2)
- Number of ALU units
- Number of MUL units
- Number of GR registers (up to 64)
- Number of BR registers (up to 8)
- Types of accessible Functional Units (FUs) per syllable – Figure 4.2 depicts the default configuration for FU accessibility. This configuration could be changed according to demands and available FUs.
- Width of memory busses – By default, the width of the data bus from the instruction memory has a width of 128 bits. Because we used a Harvard architecture, this width could be different than the data memory bus width (which is in fact 32 bits by default). Because the instruction memory bus width is inherent to the global issue-width of the processor, this should be scalable.

In principle, the number of CTRL and MEM units could also be changed. However, the VEX standard only allows one CTRL unit per VEX (multi-cluster) machine (which makes sense, because otherwise multiple CTRL operations could change the program flow by changing the Program Counter to a different value). The acMEM unit is responsible for memory load and store operations, as discussed earlier. The VEX standard defines separate units for these actions, which can have multiple instances. This can be also achieved by instantiating multiple MEM units in our design, but the memory interfacing should support multi-port read/write access in this case.

To change the different parameters, some changes to the VHDL source code of ρ-VEX have to be made. Depending on the parameter, this differs from changing a constant signal declaration to adding or removing some logic (i.e. to change the issue-width). In the current ρ-VEX implementation these changes have to be made manually. For a future release a scripted parameterization based on the VEX machine model file format (`.fmm` files) is planned, for example by using a file preprocessor.

Because we present ρ-VEX as an open source platform, the whole architecture could in principle be adjusted. However, with the parametric properties we provide a fast and efficient way to adjust common configurations.

4.4 Conclusions

The ρ-VEX design as discussed in this chapter uses a Harvard-based architecture with physically separated instruction- and data-memories. Four main stages can be identified in the processor architecture: a *fetch*, *decode*, *execute*, and *writeback* stage.

The standard set of 73 VEX operations is supported by ρ-VEX. A VEX instruction consists of four syllables (in case of a standard configuration), which can be seen as separate RISC instructions. Three types of immediate operands are supported. A syllable layout template was designed for all syllable configurations. For some operations, address packing was applied within the opcode space to be able to stay compliant to the defined templates.

Extensibility of ρ-VEX is provided by two mechanisms: ρ-OPS and VEX machine models. ρ-OPS use the free opcode space to provide opcodes that can be used freely to implement extra functionality. Both sequential and combinatorial operations are supported, as long as the design get properly synthesized. ρ-OPS can be easily added to the existing VHDL code base by adding a few lines. Most of the parametric options within VEX machine models are supported by ρ-VEX. Parameters like issue-width, number of FUs, and the number of registers can be altered easily.

5 Implementation

This chapter discusses the way we implemented the design presented in Chapter 4. Different types of signal flows were identified (those of stage control signals, the datapath and the instructions) and discussed separately. For the different processor stages, a Finite State Machine (FSM) is presented so that one is able to understand the design ρ-VEX, and even extend it. A 4-issue ρ-VEX implementation is used in the explanations.

Section 5.1 presents the bottom-up implementation approach that was taken to implement our design. Section 5.2 discusses how our design is mapped to an implementation, together with the different signal flows that can be identified in our implementation. FSMs for all stages are presented in Section 5.3. Section 5.4 discusses the used hardware verification and testing methods. Encountered problems are presented in Section 5.5. This chapter is concluded in Section 5.6.

5.1 Bottom-Up Implementation

A bottom-up implementation was used for ρ-VEX. The first goal was to implement a 1-issue RISC version of the processor. We started with the implementation of the ALU. After the design of the ALU was verified, the MUL unit was implemented. When this design was also verified, an *execute* stage wrapper was created, using the ALU and MUL units as sub-entities. From this point on, every new implementation was added to the bottom-up design. This made it relatively easy to detect errors and bugs in an early stage.

Because the processor was meant to be parametric from the design stage on, as much as possible was implemented in a parametric way from the beginning (even when the RISC version did not require many things to be parametric). This helped a lot at the point where the RISC version was working, and the VLIW version had to be implemented. In many cases, it was just copying implemented parts to other issue-slots, or adding new instances of implemented Functional Units. To ensure a system-wide parametric implementation, all VHDL was hand-coded without the use of an Integrated Development Environment (IDE) with automated design tools.

More about the used verification and testing methods used while implementing the design can be read in Section 5.4.

5.2 Mapping the Design to an Implementation

To clarify the mapping from ρ-VEX' architectural design to our implementation, we identified three different (main) signal flows. The instruction flow shows how a fetched VLIW instruction is distributed in the system as syllables, and later opcodes. The

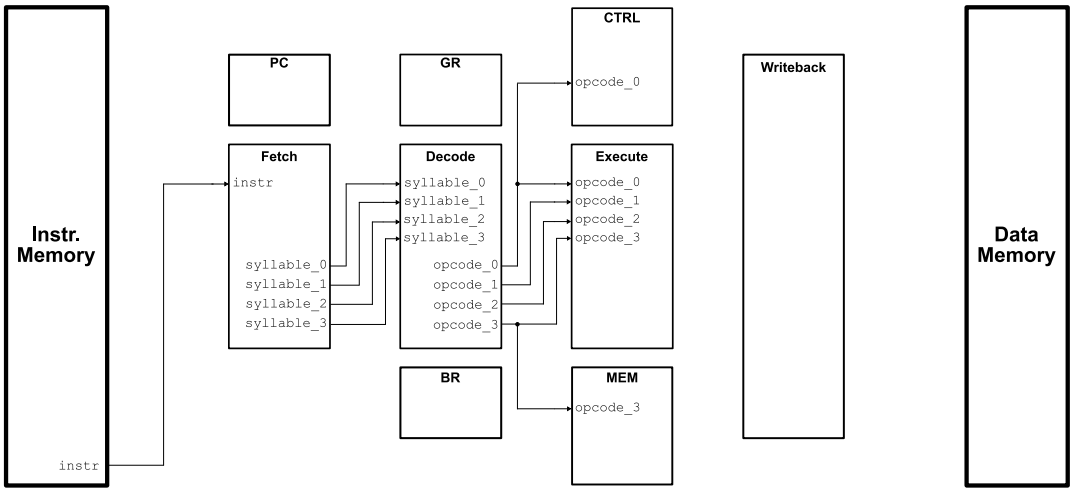


Figure 5.1: Instruction flow

datapath shows how data is being processed by the system. Finally, the inter-stage control flow shows how the different stages are dependent on each other.

By convention, we are using **bold** signal names for a bundle of signals designated for each issue slot (so `data_r1_` equals the separate mentioning of `data_r1_0`, `data_r1_1`, `data_r1_2` and `data_r1_3`, where the last suffix indicates the issue slot).

5.2.1 Instruction Flow

Figure 5.1 depicts the instruction flow as it is implemented in the current prototype. After a wide instruction is fetched from the instruction memory by the *fetch* stage, the instruction is split into 4 VLIW syllables. Those are passed to the *decode* stage, where opcodes are distilled. The opcodes are passed to the functional units inside the *execute*, *CTRL* and *MEM* stages. Because the latter two can only operate on certain syllables (see Figure 4.2), they only receive one opcode.

5.2.2 Datapath

The datapath is depicted in Figure 5.2. In this diagram all GR, BR and memory transactions are presented as well as the Program Counter (PC) updates and readouts, because all these transactions are treated similar.

5.2.2.1 Fetch Stage

The only 'data' that the *fetch* stage handles, is the current PC value (`pc`). It is used to determine which instruction to fetch from the instruction memory. This value comes directly from the Program Counter unit, and aligned to the used VLIW word size.

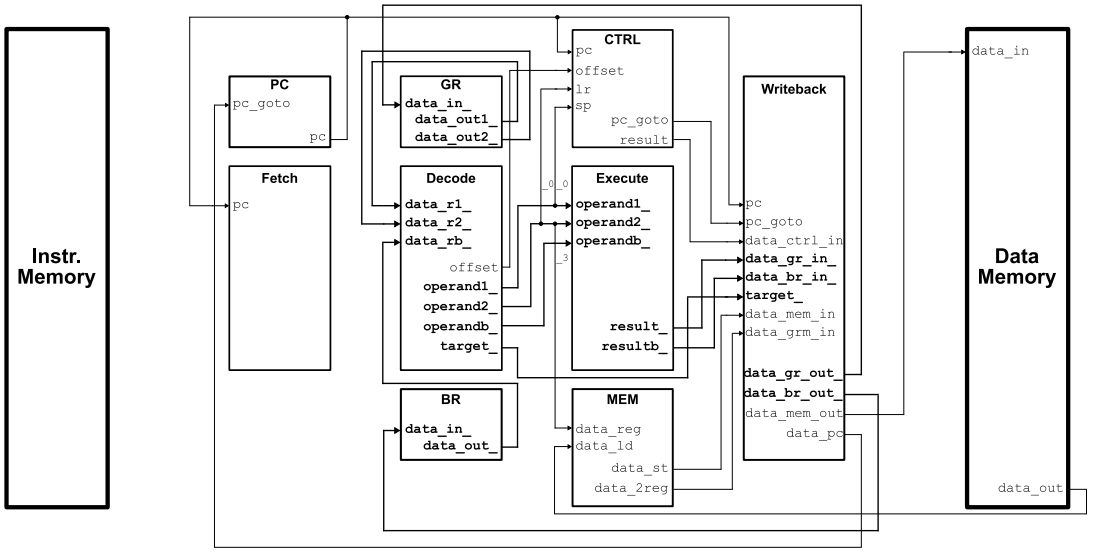


Figure 5.2: Datapath

5.2.2.2 Decode Stage

The *decode* stage receives the register values used as operands via the signal bundles **data_r1_**, **data_r2_** and **data_rb_**. The operands are presented to the *execute* stage by means of the signal bundles **operand1_**, **operand2_** and**operandb_** (respectively GR operand 1, GR operand 2 and BR operand). When an operation operates on an immediate operand instead of a register operand, the result from the GR file is not used. The *branch offset immediate* operand is passed via the `offset` signal. To determine the write targets per syllable for the *writeback* stage, the **target_** signal bundle is assigned. This signal will be discussed in Section 5.2.2.4 about the *writeback* stage.

5.2.2.3 Execute, CTRL and MEM Stages

The next parallel stages are the *execute*, CTRL and MEM stages. The *execute* stage has 4 ALU and 2 MUL units in the standard VEX configuration, so the complete bundle of operands is passed to this stage. Results of this stage are presented in the **result_** (ALU and MUL results) and **resultb_** (carry-out values) signal bundles. CTRL and MEM operations are only allowed to be issue in one issue-slot, so only one set of operands are passed to these stages. In the CTRL unit, the operands could represent the current values of the *link register* (`lr`) and the *stack pointer* (`sp`). The CTRL unit is only able to operate on the first syllable (Figure 4.2), so these signals originate from `operand1_0` and `operand2_0`. The CTRL unit also needs the current PC value (`pc`) to operate, and a possible *branch offset immediate* (`offset`). The CTRL unit passes the next PC value (`pc_goto`) in case of a branch, and the new *stack pointer* or *link register* value (`result`) as results. The MEM unit receives its operand either from the data memory (`data_ld`) in case of a memory load operation, or from the GR register file (`data_reg`, via `operand2_3`)

target_i	Writeback target	Mnemonic
000	Don't writeback	WRITE_NOP
001	General-purpose Register (GR)	WRITE_G
010	Branch Register (BR)	WRITE_B
011	GR and BR	WRITE_G_B
100	Program Counter (PC) (branch)	WRITE_P
101	PC and GR	WRITE_P_G
110	Data memory	WRITE_M
111	GR (memory load)	WRITE_MG

Table 5.1: Writeback targets in *writeback* stage

in case of a memory store operation. The `data_st` output signal represents the data to be stored in data memory in case of a store operation, and `data_2reg` embodies the data to be stored in the GR register file in case of a load operation.

5.2.2.4 Writeback Stage

In the *writeback* stage, all results from the previous stages are written back to one of the accepting targets. Different write targets could be the GR register file, the BR register file, data memory or the Program Counter. Table 5.1 presents all different writeback targets, together with the value of a `target_i` signal, which is presented for every issue slot in the signal bundle **`target_`**. GR and BR results are presented in the signal bundles **`data_gr_out_`** and **`data_br_out_`**. Data to be written to the data memory in case of a memory store operation is presented in the `data_mem_out` signal. The new PC value is presented in the `data_pc` signal. Even when no `WRITE_P` or `WRITE_P_G` `target_i` signal is present, the PC will be written by the *writeback* stage. In this case, the old PC value (`pc`) is increased by the size of 1 VLIW word, and written back to `data_pc`.

5.2.3 Inter-Stage Control Signals

Figure 5.3 depicts the inter-stage control signals, as used in our implementation. A high-level description of these control signals will follow, organized per stage. The control signals are used to control internal state machines that define the stage behaviour, which are discussed at a lower level in Section 5.3.

5.2.3.1 Fetch and PC Stages

The *fetch* stage is the first stage within the chain of processor stages. The two control signals that reach beyond the boundaries of the ρ-VEX top-level entity are both evaluated and driven in this stage. The two signals are `run` and `done`, which are respectively being routed to `start` and `stop_out` in the *fetch* stage. When `start` is driven high by a module residing outside the ρ-VEX boundaries, ρ-VEX is supposed to start processing instructions from the instruction memory (starting at address 0x0). When processing is finished (thus when a **STOP** operation has been decoded), `stop_out` is being driven

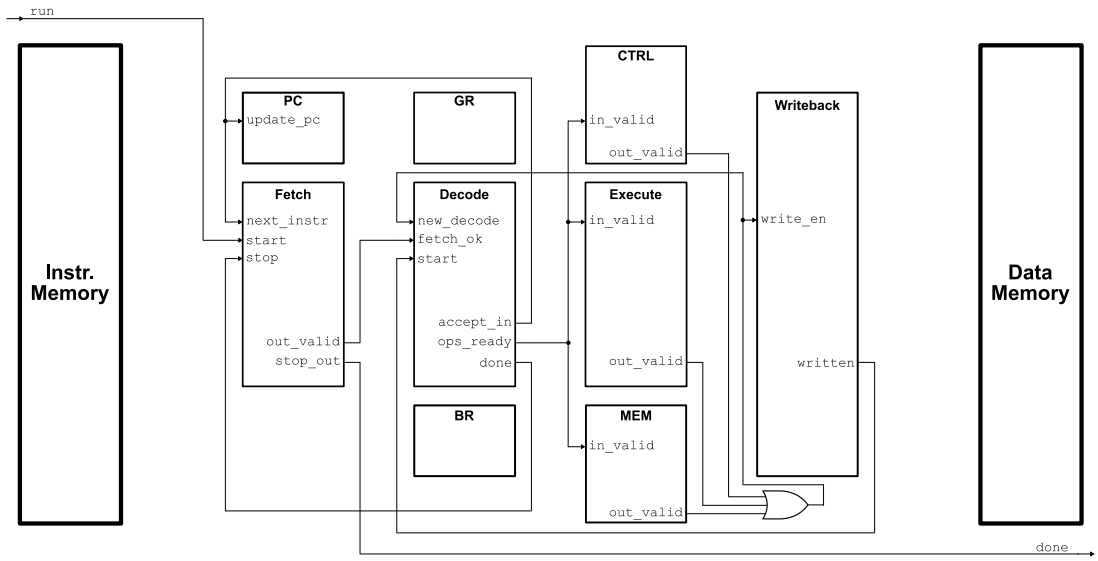


Figure 5.3: Inter-stage control signal flow

high by the *fetch* stage. Consecutively, the *decode* stage will notify the *fetch* stage by means of the `stop` bit. When input `next_instr` is high, the *fetch* stage is supposed to fetch a new instruction from the instruction memory. After an instruction has been fetched successfully, `out_valid` will be driven high.

5.2.3.2 Decode Stage

A new instruction is being decoded after `fetch_ok` is raised by the *fetch* stage, and operands are fetched from the GR and BR register files. The new operands are presented to the *execute* stage, and `ops_ready` will be driven high. The `new_decode` bit indicates whether (one of) the *execute*, CTRL and MEM stages are done with processing the current syllables. When the *writeback* stage is done writing the back results to their targets, `start` is asserted. As long as `start` is low, the *decode* stage presents the destination addresses used for writing back the results to the GR and BR register files continuously. `accept_in` is driven high when the *decode* stage accepts a new instruction from the *fetch* stage.

5.2.3.3 Execute, CTRL and MEM Stages

The *execute*, CTRL and MEM stages all have an `in_valid` bit that tells the stage whether the opcodes and operands are ready to be processed. When processing finished, these stages dive an `out_valid` bit high. The three `out_valid` bits are then passed through a logical OR-gate, to create a single signal from them.

5.2.3.4 Writeback Stage

When `write_en` is driven high by the combined `out_valid` signals from the previous stages, the *writeback* stage is supposed to write back the results to their desired targets. When everything is written back, the `written` bit is asserted.

5.2.4 Pipelining the Design

In the current implementation, one operation uses 6 (for ALU operations) or 7 (for MUL operations) cycles to be fetched, decoded, executed and written back. To decrease this number and potentially increase the clock frequency, the implementation could be pipelined. ρ-VEX is designed to be easily pipelined, as can be seen from the various stage diagrams with the signal flows. To have a pipelined implementation, these things have to be done:

- Pass destination GR and BR addresses through the *execute* stage to the *writeback* stage with pipeline registers, instead of 'directly' driving these addresses from the *decode* stage to the corresponding register files.
- Pass the **`target_`** signal bundle through the *execute* stage to the *writeback* stage with pipeline registers, instead of bypassing it.
- Let consecutive stages only be dependent on control signals driven by their direct neighbour stages, instead of 'skipping' stages.
- The `stop` bit for the *fetch* stage should be driven by the *writeback* stage, in order not to lose operations inside the pipeline.
- The Program Counter should be updated by the *fetch* stage upon non-CTRL operations (possible branches).
- The current PC value (for the syllables in a certain stage) should be registered between each stage.
- When a CTRL syllable is decoded, a register `flush_pipeline` should be set, in order to flush the pipeline before branching.

5.3 Internal Stage Control

To clarify the inner workings of the different stages, a Finite State Machine (FSM) is presented for every processor stage. The inputs of the FSMs are the input signals in the control signal diagram in Figure 5.3, the outputs are the corresponding output signals in the same diagram. The presented FSMs only depict control signals, no data or operational signals. These are inherent to the description. Upon power-on, all FSMs are initialized in the **reset_state**.

We can distinguish two different FSM types: a Moore [33] FSM and a Mealy [34] FSM. In a Moore FSM, the output of the machine only depend on the current state in which the machine resides. In a Mealy machine, the output depends on both the input and the current state.

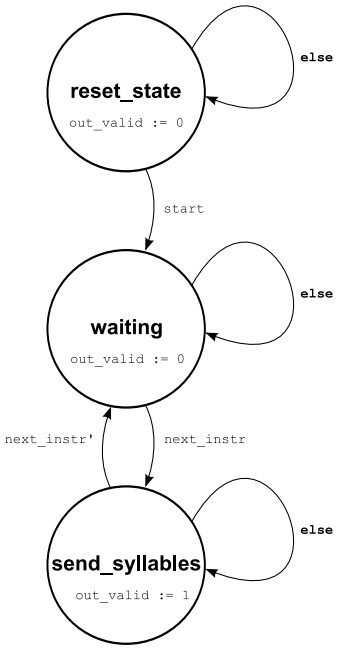


Figure 5.4: Moore FSM for the *fetch* stage.

5.3.1 Fetch Stage

Figure 5.4 depicts the FSM for the *fetch* stage, which is of the Moore type. This stage stays inside its **reset_state**, until `start` is raised, which is connected to the `run` pin of ρ-VEX. When in the **waiting** state, the *fetch* stage gets a VLIW word from the instruction memory (the address comes directly from the Program Counter) containing the instruction to be executed. When new syllables are demanded by the *decode* stage, `next_instr` is high, and the **send_syllables** state is reached. In this state, the split syllables are presented to the *decode* stage, and `out_valid` is raised. When `next_instr` becomes low again, the stage will fall back to the **waiting** state.

It should be noted that next to this state machine, the *fetch* stage has another small mechanism which triggers the `stop` signal from the *decode* stage. When a low-to-high transition is detected of this signal, `stop_out` is raised, and the state machine in Figure 5.4 is disabled. ρ-VEX halts.

5.3.2 Decode Stage

A Mealy-type FSM for the *decode* stage is depicted in Figure 5.5. After landing in the **reset_state** directly after power-on, the machine will directly fall into the **waiting** state where `accept_in` is raised. Upon a positive transition of `fetch_ok`, the stage starts fetching register contents from the GR and BR register files. This occurs in the **fetch_regs** state. In the next cycle, a state-transition is made to **send_operands**, where `ops_ready` is raised. When a **STOP** operation has been decoded, `done` is raised. When the execution stages and the *writeback* stage are finished (by sensing high signals on `new_decode` and `start`), the state machine falls back in the **waiting** state.

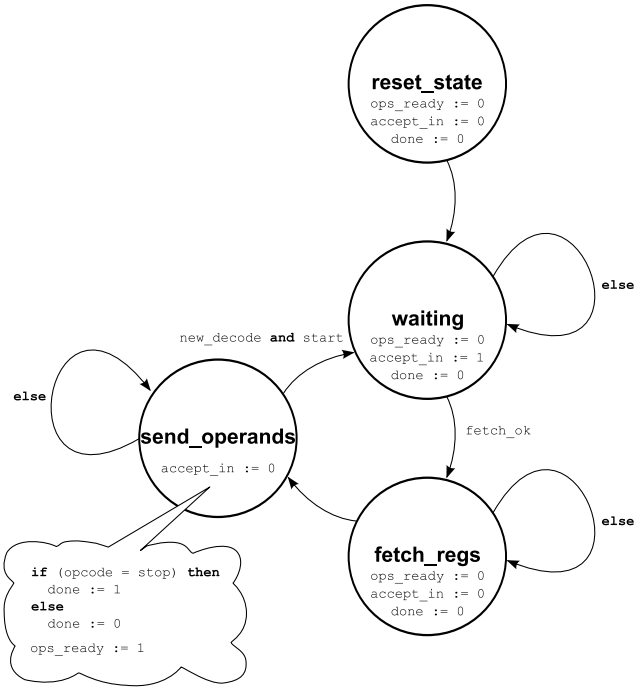


Figure 5.5: Mealy FSM for the *decode* stage.

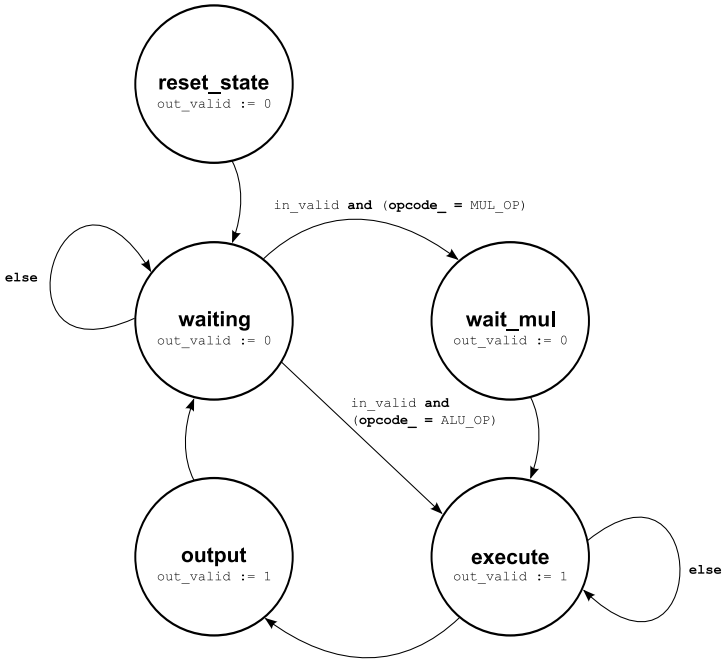


Figure 5.6: Moore FSM for the *execute* stage.

5.3.3 Execute Stage

The FSM for the *execute* stage is depicted in Figure 5.6. Again, this is a Moore machine. After power-on or a system reset, the machine directly makes a transition to the **waiting** state, where it is waiting for decoded opcodes and operands from the *decode* stage. When a `in_valid` becomes high, a state transition is made. Depending on the kind of operation (ALU or MUL), a transition is made to **execute** or **wait_mul**. Because the Multiplier unit has a delay of 2 cycles as defined by the VEX standard in [1], an extra single-cycle waiting stage is implemented for these operations. In the **execute** state, `out_valid` is raised, and output is kept valid for one more cycle in the **output** state.

5.3.4 CTRL Stage

The CTRL stage is not implemented as a state machine. This is a synchronous design which raises `out_valid` upon receiving a valid opcode at its input, and performing the corresponding processing.

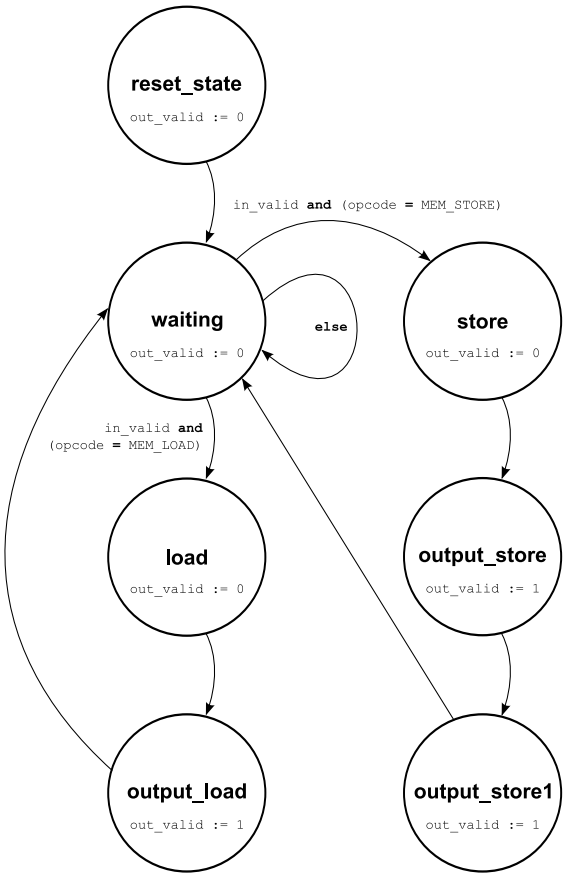


Figure 5.7: Moore FSM for the MEM stage.

5.3.5 MEM Stage

Figure 5.7 presents the Moore FSM for the MEM stage. Like the previous state machines, this machine makes the transition to the **waiting** state directly after power-on. When `in_valid` is high, and a memory load or store opcode is detected, a state transition occurs to either the **load** or **store** state. As memory store operations take two extra delay cycles in our implementation, two single-cycle delay stages are implemented for a store operation: **output_store** and **output_store1**. A memory load operation has only one extra delay cycle, in the **output_load** state.

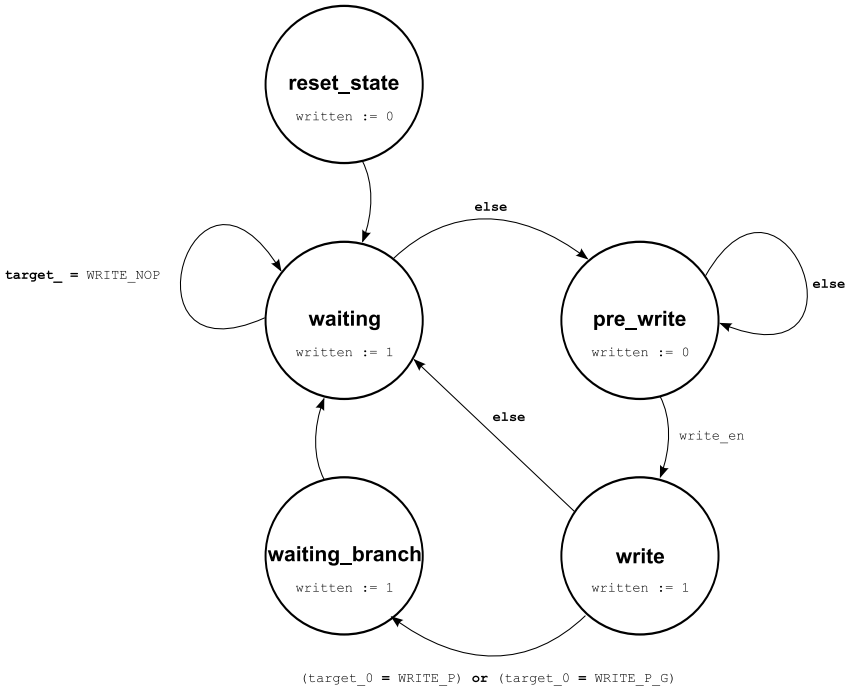


Figure 5.8: Moore FSM for the *writeback* stage.

5.3.6 Writeback Stage

The last state machine, describing the *writeback* stage is presented in Figure 5.8. Again, a transition to the **waiting** state is being made after power-on. `written` is raised, in order to tell the *decode* stage that decoding can start (as far as the *writeback* stage concerns). If neither of the write target signals in the `target_` bundle is `WRITE_NOP` (which is the default value when the *decode* stage does not have a valid output), a transition is made to the **pre_write** state. When `write_en` is high, a transition to **write** takes place. Here the actual writeback process takes place. When a CTRL operation was issued, an extra transition to the single-cycle delay state **waiting_branch** is made, in order for the Program Counter to adjust.

5.4 System Verification and Testing

Although the process of designing and developing strictly according to a very extensive methodology is quite time-consuming, we did recognize its importance. Especially considering the relatively short time span and the limited availability of human resources of a MSc project, we had to define some custom verification trajectory. The methodology we used is loosely modelled on the Unified Verification Methodology (UVM), developed by Cadence Design Systems [35] in 2002.

To be able guarantee the correct operation of ρ-VEX, simulations and tests of the design already started at an early stage. After pre-synthesis simulations with CAD/EDA software were performed, post-place and route simulations were performed to ensure the correct working on real hardware. After all simulations were successfully performed, real-world tests were done on (multiple) FPGA development boards. A system wrapper was created in order to use the same environment in all simulations and tests.

5.4.1 The Unified Verification Methodology

The UVM trajectory is based on four phases:

1. **System-Level Design** – The product is defined and an Functional Virtual Prototype (FVP) is created. An FVP is a complete functional representation of the design and its testbench.

2. **Subsystem Verification** – The design and verification of separate subsystems is done.

3. **System Integration** – The final system is integrated efficiently because of testbench reuse and common models. Integration is done one subsystem at a time.

4. **System Verification** – The system is verified under real-world operating conditions.

During the first phase, we defined all system constraints. Because the VEX ISA is reasonably well documented (read more in Section 5.5), this went fairly fluently. During the next phase, we implemented all identified subsystems separately (using the bottom-up approach as described in Section 5.1). All functionality of the FUs (i.e. the ALU, MUL, MEM, CTRL) was fully verified by simulations. As the UVM prescribes, the final system was integrated one subsystem at a time. Because of the limited human resources, we could not develop all subsystems concurrently. Therefore, each subsystem was directly integrated in the final system after it was verified. The final system was verified by means of behavioural simulations, post-place and route simulations and real-world tests on FPGA platforms. To verify the final system, selected testbenches were used that ran test applications from instruction memory.

5.4.2 System Wrapper

To be able to perform the different kinds of simulations and real-world tests in the same environment, a system wrapper was designed. This system wrapper encapsulated a

larger set of subsystems until the *System Integration* phase was reached from the Unified Verification Methodology.

Figure 5.9 shows a schematic representation for the final iteration of the system wrapper. Within this system wrapper, an instruction memory, a data memory, a UART interface, and the ρ-VEX processor reside. The UART interface was designed and implemented especially for real-world hardware tests, to easily obtain feedback results from our system.

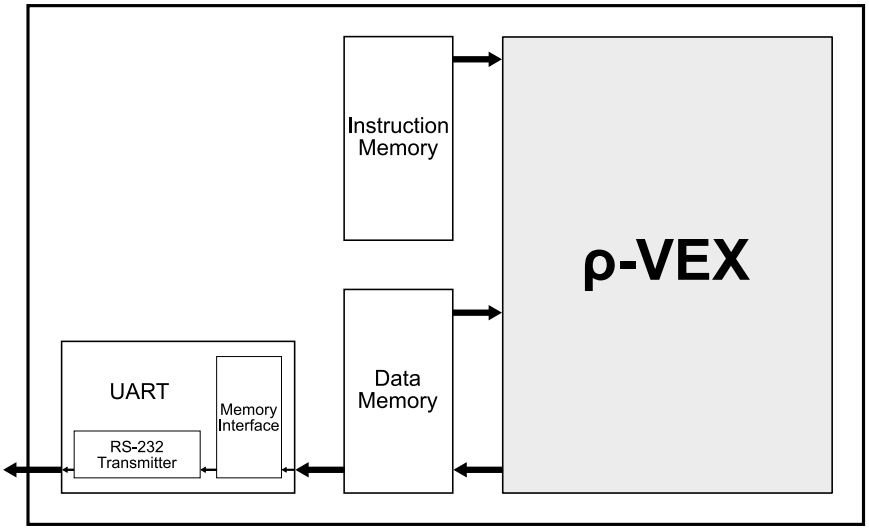


Figure 5.9: Schematic representation of the system wrapper

5.4.3 Simulations

Our simulations were mainly performed for functional testing and for limited manufacturing testing [36]. This is because the (research) nature of the project that does not directly demand a product ready to be mass-produced. Still, the post-place and route simulations and real-world hardware tests did help to eliminate technology-driven issues.

5.4.3.1 Pre-Synthesis Simulations

Every time a new subsystem implementation was finished, pre-synthesis behavioural simulations were performed to verify the subsystem's functionality. Simulations were performed using Mentor Graphics ModelSim SE 6.3d.

A simulation testbench with a stimulus was used to verify the subsystem. When these simulations resulted in the expected results, the subsystem was integrated with the other verified parts and simulated with a higher-level behavioural testbench. Appendix E presents waveform diagrams of the benchmark used in Chapter 7. Waveform diagrams of pre-synthesis simulations are presented in Section E.1.

5.4.3.2 Post-Place and Route Simulations

After behavioural simulations gave expected results, post-place and route simulations were performed. To perform these simulations, the Xilinx-specific `simprim`, `unisim`, and `XilinxCoreLib` libraries had to be compiled. These libraries include well-modelled heterogeneous Xilinx FPGA parts, so that the correct timing constraints and delays (including routing delays) were used.

Section E.2 presents waveform diagrams of post-place and route simulations of the benchmark used in Chapter 7.

5.4.4 Verification on Hardware

Real-world tests on different FPGA platforms were performed to ensure an implementation as technology-independent as possible. During the course of the project, different hardware debugging methods were used.

5.4.4.1 Target Technology

Initially, two different hardware platforms were chosen to verify our implementation on: a platform based on the Xilinx Spartan-3E (XC3S500) FPGA and one based on the Xilinx Virtex-II Pro (XC2VP30) FPGA. For both hardware platforms, we had development boards from Digilent available.

As the Spartan-3E device we had only featured 500K gates, our implementation could not be placed and routed correctly anymore after the 1-issue RISC implementation of ρ-VEX. From this point on, we exclusively used the Virtex-II Pro platform as implementation target.

To synthesize the hardware, Xilinx Synthesis Technology (XST) from the Xilinx Integrated Synthesis Environment (ISE) 8.1.03i suite was used. Chosen was for this somewhat outdated (2006) release of the suite, because the current MOLEN prototype was only successfully implemented with this toolchain at the time we started.

5.4.4.2 Hardware Debugging Methods

During the first hardware tests, we used a combination of LEDs and switches on the FPGA development boards to present debugging feedback. With the switches, one was able to select the GR register of which the content should be represented by the LEDs (in binary). This worked fine when doing small tests, but later on, testing became rather complex.

To ease debugging, an RS-232 compatible Universal Asynchronous Receiver/Transmitter (UART) transmitter interface was designed implemented, and added to our system. A small interface to control the UART transmitter and obtain contents from the data memory was placed in-between the transmitter and ρ-VEX. After ρ-VEX' `done` bit is raised, the UART interface starts transmitting the contents of the data memory together with the number of clock cycles used, in ASCII representation. As this debugging interface appeared to be a useful addition to the system, it was incorporated in the development framework. More details about the UART interface can be read in Section 6.3.

5.5 Encountered Problems & Solutions

The biggest issues that were encountered had to do with the conversion of VHDL code that simulated well to synthesizable code. Before using the UVM, a relatively naive verification 'approach' was used. Behavioural simulations were performed extensively, but no post-synthesis simulations or real-world hardware tests were performed. At about 60% of the 1-issue RISC implementation with our initial approach, the first hardware tests were performed (without any accompanying post-synthesis simulations). Synthesization of the code went without errors, but a lot of warnings were generated by the synthesizer. The resulting implementation did not give any positive results when programmed on an FPGA. After harvesting through the VHDL code (that simulated well) for a couple of days, it was decided to start implementing the design all over again using the Unified Verification Methodology. Of course most of the code could be reused after some small modifications. It seemed that a lot of gated clocks were inferred, as well as inefficient state machines. After doing more research about VHDL coding style [36] for synthesizable designs, Xilinx specific code directives [37], and following the UVM, increasingly better results were obtained.

Some smaller issues arose from the definition VEX ISA that was not always as strict as possible. It was 'accidentally' found on the VEX Internet forum that the **GOTO/IGOTO** and **CALL/ICALL** pairs were overloaded in the latest releases of the VEX toolchain.

Some places in the VEX documentation refer to GR `$r0.63` as the link register used in some CTRL operations. However, it is undetermined what the link register will be when the GR register file is scaled down to for example 32 registers.

The **MTB** (move GR to BR) operation was totally undocumented. It was found by seeing an unrecognized in the assembly code generated by the VEX compiler. The **MOV** (move GR to GR) was only half documented; this operation is allowed to operate on immediate operands but this was not clear from the VEX documentation [1]. The latter one was only found recently, while assembling compiler-generated assembly with ρ-ASM. The current release of ρ-VEX does not support this, this is an open issue.

Additionally, a large part of the VEX semantics (Appendix B) had to be figured out from compiled code. Because of some very recent discoveries, some (small) semantical parts still have to be added to the semantical recognition part of ρ-ASM.

An issue tracker on the project website at `http://r-vex.googlecode.com` is used to organize all issues and report about the status.

5.6 Conclusions

This chapter discussed different aspects that had to do with the implementation of ρ-VEX. We used a bottom-up approach for the implementation of ρ-VEX. Starting by implementing small Functional Units, we worked towards an implementation of a 1-issue RISC ρ-VEX. When this model was verified, the design was extended to a 4-issue processor.

To be able to map the design to an implementation, we identified three signal flows within the processor. An instruction flow, a data flow, and an inter-stage control signal

flow helped the implementation process. Every processor stage was implemented as a small state machine. These state machines control the behaviour of the inter-stage control signals.

For the implementation trajectory, the Unified Verification Methodology (UVM) was used as a guideline. By dividing the implementation phase in multiple sub-stages UVM seemed a structured methodology to ensure correct results. For each identified sub-system, behavioural simulations as well as post-place and route simulations were performed. After simulations presented correct results, verifications were done on real hardware.

During the implementation of the processor, several problems were encountered. The most notable issues had to to with the conversion of VHDL code that simulated well to synthesizable code. Other problems arose from the fact that the VEX ISA definitions were not always very consistent.

6 Development Framework

To be able to efficiently perform experiments and develop applications for the ρ-VEX platform, a framework was developed. This framework consists of a design flow, an assembler (ρ-ASM), and a UART debugging interface.

Section 6.1 discusses the presented application development flow. The ρ-ASM assembler is presented in Section 6.2. Next, the UART debugging interface is discussed in Section 6.3. A description of the hardware development flow is presented in Section 6.4. Section 6.5 concludes this chapter.

6.1 Application Development Flow

Figure 6.1 depicts a schematic representation of the ρ-VEX application development framework that we present. It basically consists of two stages:

1. Compile an arbitrary piece of application code with the VEX C compiler [8]. When targeting a non-standard ρ-VEX configuration (as discussed earlier), a VEX machine model should be present as a compiler directive (in the form of an `.fmm` file). When ρ-OPS are used, the application code should be augmented with the correct pragmas that define the ρ-OPS.
2. The VEX assembly file generated by the compiler should be the input for ρ-ASM, which generates ρ-VEX object code. ρ-ASM should also receive the VEX machine model ant ρ-OPS definitions by means of a mapping between operation and opcode/class.

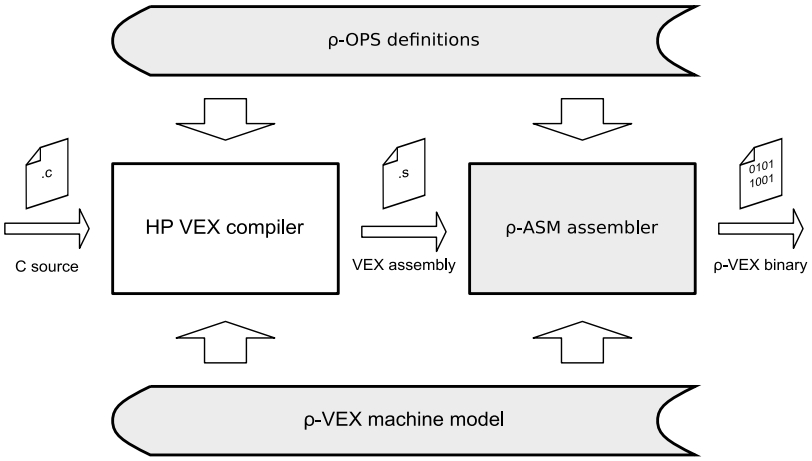


Figure 6.1: ρ-VEX application development framework

The framework model we present, currently has some limitations in accepting assembly code generated by the VEX compiler. The current version of ρ-ASM does not support initializing the data memory, and some undocumented (but where discovered by coincidence) VEX assembly semantics as generated by the compiler are not recognized by ρ-ASM. Also, within the compiler-generated assembly code, calls to functions within the host's system libraries are made. These are handled by the VEX compiled simulation system, as no hardware implementation of a VEX processor is provided.

As ρ-VEX is mainly targeted to compute (small) application-specific kernels, it might even not be desirable to execute the compiler-generated assembly directly. The VEX assembly generated by the compiler could be used to extract the assembly code for a particular function, and use this code as the input for ρ-ASM.

6.2 ρ-ASM Assembler

The assembler/instruction ROM generator for ρ-VEX is called ρ-ASM. It is a 2-pass assembler written in C that is able to assemble an assembly file with VEX assembly (as described in Appendix B) to an instruction ROM. The current release of ρ-ASM has the following features:

- Assemble a VEX assembly file to an instruction ROM VHDL file which can be synthesized directly together with the other parts of ρ-VEX.
- Support for ρ-OPS. However, currently ρ-OPS definitions should be manually added to `syllable.h` of the application code. ρ-ASM should be recompiled afterwards to support the added operations.
- Support for different VEX `.fmm` machine model parameters. These could currently only be applied in the `rasm.h` and `syllable.h` files currently, so recompilation is needed.

The current version of ρ-ASM is not able to initialize the data memory with contents. A future release should enable this, as well as the support of parametric input via configuration files instead of direct source code modifications.

ρ-ASM always adds a **STOP** operation together with 3 **NOP** operations in an instruction after the last instruction defined. This is to make sure program execution ends.

Open issues regarding the ρ-ASM implementation can be found in the issue tracker on the project website at `http://r-vex.googlecode.com`.

6.3 UART Debugging Interface

As described in Section 5.4.4.2, we developed a debugging UART interface to transmit data via the serial RS-232 protocol, at a data rate of 115200 bps. This interface invokes a transmission of the hexadecimal representation of the data memory contents, as well as the contents of the internal ρ-VEX cycle counter register. All data is transmitted

using American Standard Code for Information Interchange (ASCII) encoding, so that the output could be captured and evaluated on a terminal application on a Personal Computer. An example of a data memory listing is presented below:

```
r-VEX
-----
Cycles: 0x0000002D

Data memory dump

addr | contents
-----+-----------
0x00 | 0x000000FF
0x01 | 0x0000000F
0x02 | 0x00000EF1
0x03 | 0x00000000
0x04 | 0x00000000
0x05 | 0x00000000
0x06 | 0x00000000
0x07 | 0x00000000
0x08 | 0x00000000
0x09 | 0x00000000
0x0A | 0x00000000
0x0B | 0x00000000
0x0C | 0x00000000
0x0D | 0x00000000
0x0E | 0x00000000
0x0F | 0x00000000
```

The interface can be easily disabled, as it resides in the system wrapper and not within ρ-VEX itself.

For a future release, we might want to add the support to upload an executable program file for ρ-VEX via the UART, and invoke a boot loader (so load the instruction memory with the appropriate data, and soft-reset ρ-VEX).

6.4 Hardware Development

To ease the hardware development of ρ-VEX, a `Makefile` for GNU's *Make* tool was created. This `Makefile` works in any Linux environment where the Xilinx ISE suite is installed, and in any Microsoft Windows environment with ISE and Cygwin installed (the basic Cygwin installation that comes with Xilinx' EDK is supported). The `Makefile` is based on a Xilinx 8.1 environment, but should work (or at least be easily adapted to work) with any later versions. By using the `Makefile`, one is able to quickly synthesize, place and route the hardware design by entering just 1 command. Other possibilities are running a hardware behavioural simulation using Mentor Graphics ModelSim and

programming the synthesized design to an FPGA. All log-files generated by the different invoked Xilinx tools, are structurally concatenated into one new logfile in ASCII text.

The different *Make* targets as outputted when running the `make` command without any arguments inside the source code directory of ρ-VEX is presented below:

```
---------------------------------
r-VEX stand-alone system deployer
---------------------------------

Usage: make <target>

Targets:  v2p      | Xilinx Virtex-II Pro VP30 Development Board

          modelsim | Run modelsim simulation
          vcom     | Compile files for Modelsim

          fpga     | Download bitstream to FPGA
          unlock   | Unlock download cable

          clean    | Clean generated project files
```

A Quickstart Guide presenting a short user manual to easily deploy an instance of ρ-VEX on an FPGA development board is presented in Appendix F. This guide also provides information about adding ρ-OPS to your implementation, as well as on how to extend the `Makefile` with targets for other development boards than the Xilinx Virtex-II Pro VP30 board by Digilent.

6.5 Conclusions

An application development flow consisting of two phases was presented in this chapter. The first phase consists of compiling an arbitrary application kernel described in C code. The second phase converts the generated assembly code into an executable program for ρ-VEX. Both phases support extensibility through ρ-OPS and machine model configurations.

An assembler/instruction ROM generator ρ-ASM was created, to translate VEX assembly applications into an executable application within the ρ-VEX instruction memory. ρ-ASM also supports ρ-OPS custom operations and is parametric in order to support multiple ρ-VEX configurations. Additionally, a UART interface was designed and implemented to present the contents of the data memory in a human-readable (ASCII) way. This allows easy debugging and prototyping. It can be easily disabled, as the interface resides within the system wrapper (not in the processor itself). A *Make* environment for extending the ρ-VEX hardware design was created. Our Quickstart Guide provides information on how to start working with this environment.

7 Experimental Results

To measure the performance of ρ-VEX, we did benchmarks based on Fibonacci's Sequence. The setup used for our benchmarks is discussed in Section 7.1. In Section 7.2, the Fibonacci benchmark we used is presented, as well as the performance results. Resource utilization results of the ρ-VEX processor are discussed in Section 7.3. Conclusions are drawn in Section 7.4.

7.1 Experimental Setup

The ρ-VEX organization has been described in VHDL and simulated with Mentor Graphics ModelSim SE 6.3d. Synthesis was performed with XST from the Xilinx ISE 8.1.03i suite. As the target reconfigurable technology, the Xilinx Virtex-II PRO (XC2VP30) FPGA was chosen, embedded on the XUP V2P development board by Digilent. All experiments were performed on a non-pipelined ρ-VEX system with 32 general purpose registers (GR). A data memory of 1 kB implemented using Block RAM was connected to ρ-VEX to store results. The issue width of ρ-VEX was varied between 1, 2 and 4. All configurations had the same number of ALU units as their issue width. The 2- and 4-issue ρ-VEX configurations had 2 MUL units.

7.2 Fibonacci's Sequence Benchmark

We hand coded an assembly program that calculates the 45th Fibonacci number from Fibonacci's Sequence:

$$F_n = \begin{cases} 0 & \text{if } n = 0; \\ 1 & \text{if } n = 1; \\ F_{n-1} + F_{n-2} & \text{if } n > 1. \end{cases}$$

We created the code for 1-, 2- and 4-issue ρ-VEX configurations. The assembly code for all configurations can be found in Appendix D. We created two ρ-OPS, **FIB3** and **FIB4**. These operations calculate, respectively, 3 and 4 Fibonacci iterations in one cycle. The operations are done in a naive sequential way, to show the support for sequential operations within ρ-OPS. More information on how we implemented the ρ-OPS in an ρ-VEX processor can be found in Section F.4 of Appendix F.

We adapted our 4-issue code to use the ρ-OPS as a final benchmark. The resulting numbers of executed clock cycles are presented in Table 7.1. Because the code was efficient to parallelize, we can see that the number of clock cycles almost halves when the issue width doubles. After using ρ-OPS, we see the expected speedup of almost 4. Because the core of the application consists of only 2 VLIW instructions, we are able to achieve such a high speedup by only adding 2 ρ-OPS.

Configuration	Cycles
1-issue	1906
2-issue	1080
4-issue	537
4-issue, ρ-OPS	141

Table 7.1: Results of Fibonacci sequence benchmark

Configuration	Freq. (MHz)	Slices	Slices GR
1-issue	89.44	1895 (13%)	1 (0%)
2-issue	89.44	5105 (37%)	3370 (24%)
4-issue	89.44	10433 (76%)	3927 (28%)

Table 7.2: Resource utilization for different ρ-VEX configurations

7.3 Resource Utilization

The aforementioned ρ-VEX configurations were synthesized without any connected memories, to check the resource usage. Table 7.2 presents the results of the measurements. Next to the number of slices, the percentage of slices used from the total available slices on the FPGA is presented. Large increases in the usage of slices on the FPGA can be seen when increasing the issue width of ρ-VEX. This can be mainly ascribed to the growing GR register file. Because the GR register file in the 1-issue ρ-VEX can be totally implemented in dedicated Xilinx 2-port Block RAM, the implementation uses no slices. The 2- and 4-issue ρ-VEX configurations, however, require 4- and 8-port register files, respectively. These memory configurations can not be instantiated as Xilinx primitive elements, therefore they need to be formed by slices. This consumes a large number of slices. An interesting trade-off might be a multi-cluster machine configuration consisting of single-issue ρ-VEX cores instead of one single-cluster, multi-issue ρ-VEX machine. As the VEX compiler has the ability to schedule data moves across VEX clusters, we already have architectural support for this. All configurations were synthesized to run at the same clock speed of 89.44 MHz.

7.4 Conclusions

This chapter presented the results of the benchmark we did on different ρ-VEX configurations. We hand coded an assembly application to calculate the 45th number from Fibonacci's Sequence. Different versions were created, to run on 1-, 2- and 4-issue ρ-VEX processors. A version using ρ-OPS custom operations was also created for a 4-issue ρ-VEX processor. The results showed speedups of factor 2 by doubling the issue-width and adding ρ-OPS.

We synthesized the different ρ-VEX processor configurations without memories and system wrapper to obtain the area utilization on the Xilinx XC2VP30 FPGA. A 1-issue ρ-VEX configuration consumes 13% of the available logic slices on a XC2VP30

device, a 2-issue configuration consumes 37%, and a 4-issue 76%. The largest impact on these differences is caused by the GR register files. This is caused by the fact that an n-issue ρ-VEX processor should have n-port register file memories, which can not be instantiated as Xilinx primitive elements. All ρ-VEX configurations were synthesized to work correctly with a maximum clock frequency of 89.44 MHz.

8 Conclusions

This chapter presents the conclusions that could be drawn from the performed research. Section 8.1 presents a summary of this thesis, organized per chapter. The main contributions of our work are discussed in Section 8.2. Finally, recommendations for future work are proposed in Section 8.3.

8.1 Summary

In this thesis, we presented ρ-VEX, an open source reconfigurable and extensible VLIW processor based on the VEX ISA [1]. Various architectural aspects like the syllable issue-width, the number of functional units and the sizes of register files are parametric. Reconfigurable operations are also supported in the form of ρ-OPS. Because of the freely available VEX compiler by HP [8], which supports the same extensibility options for its code generation, we already possess a good compiler for our processor. The processor is accompanied by an application development framework, to optimally exploit the various degrees of freedom for development. Our processor and framework are targeted at VLIW prototyping research and embedded processor design in a stand-alone environment. After some further work regarding MOLEN integration, ρ-VEX will be a scalable co-processor for the the utilization within a MOLEN machine.

In Chapter 2, a background was presented on the technologies used in our research. Different softcore approaches by others resulted in designs and implementations of several processors. These approaches vary between fixed and very extensible designs. Instruction Level Parallelism (ILP) is exposed within an application when a computer system is able to execute multiple different operations, when a single stream of instructions is presented. Architectures that are able to exploit ILP are superscalar (by means of hardware) and VLIW (by means of a compiler). The VEX architecture is designed at Hewlett-Packard, based on the Lx VLIW processor architecture. The VEX ISA supports multi-cluster computing machines with a variable issue-width. A compiler and simulator toolchain are made freely available by Hewlett-Packard. The MOLEN paradigm presents a reconfigurable processor architecture, consisting of a General-Purpose Processor with a fixed instruction set and a reconfigurable co-processor. Custom operations are placed in the co-processor as Custom Computing Units.

In Chapter 3, we presented performance benchmarks performed for the Lx processor configurations (by others), and for different VEX configurations (by ourselves). When looking at the differences per Lx cluster-configuration, we see that increasing the cluster-width from from 1 to 2 results in performance gains of about 28%. Increasing the cluster-width to 4 has a negligible effect in most benchmarks. Our own benchmarks showed that VEX machines scored in-between the PowerPC 405 and a MOLEN CCU performance-wise, as we expected. Furthermore, the performance gain from a 1-cluster

to a 2-cluster configuration was again negligible in many cases. These benchmarks result in the decision to first implement a 1-issue RISC ρ-VEX configuration, followed by a 1-cluster 4-issue standard configuration ρ-VEX.

In Chapter 4, the ρ-VEX design was discussed. Our design is based on a Harvard architecture with physically separated instruction- and data-memories. Four main stages can be identified in the processor architecture: a *fetch*, *decode*, *execute*, and *writeback* stage. A syllable layout template was designed for all syllable configurations. For some operations, address packing was applied within the opcode space to be able to stay compliant to the defined templates. Extensibility of ρ-VEX is provided by two mechanisms: ρ-OPS and VEX machine models. ρ-OPS use the free opcode space to provide opcodes that can be used freely to implement extra functionality. Both sequential and combinatorial operations are supported, as long as the design get properly synthesized. ρ-OPS can be easily added to the existing VHDL code base by adding a few lines. Most of the parametric options within VEX machine models are supported by ρ-VEX. Parameters like issue-width, number of FUs, and the number of registers can be altered easily.

In Chapter 5, we discussed different aspects that had to do with the implementation of ρ-VEX. We used a bottom-up approach for the implementation of ρ-VEX. Starting by implementing small Functional Units, we worked towards an implementation of a 1-issue RISC ρ-VEX. When this model was verified, the design was extended to a 4-issue processor. To be able to map the design to an implementation, we identified three signal flows within the processor. An instruction flow, a data flow, and an inter-stage control signal flow helped the implementation process. For the implementation trajectory, the Unified Verification Methodology (UVM) was used as a guideline. For each identified sub-system, behavioural simulations as well as post-place and route simulations were performed. After simulations presented correct results, verifications were done on real hardware.

In Chapter 6, we presented an application development flow consisting of two phases. The first phase consists of compiling an arbitrary application kernel described in C code. The second phase converts the generated assembly code into an executable program for ρ-VEX. Both phases support extensibility through ρ-OPS and machine model configurations. An assembler/instruction ROM generator ρ-ASM was created, to translate VEX assembly applications into an executable application within the ρ-VEX instruction memory. ρ-ASM also supports ρ-OPS custom operations and is parametric in order to support multiple ρ-VEX configurations. A UART interface was designed and implemented to present the contents of the data memory in a human-readable (ASCII) way. This allows easy debugging and prototyping. It can be easily disabled, as the interface resides within the system wrapper (not in the processor itself). In Chapter 7, we presented the results of the benchmark we performed on different ρ-VEX configurations. We hand coded an assembly application to calculate the 45th number from Fibonacci's Sequence. Different versions were created, to run on 1-, 2- and 4-issue ρ-VEX processors. A version using ρ-OPS custom operations was also created for a 4-issue ρ-VEX processor. The results showed speedups of factor 2 by doubling the issue-width and adding ρ-OPS. We synthesized the different ρ-VEX processor configurations without memories and system wrapper to obtain the area utilization on the Xilinx XC2VP30 FPGA. A 1-issue ρ-VEX configuration consumes 13% of the available logic slices on a XC2VP30

device, a 2-issue configuration consumes 37%, and a 4-issue 76%. The largest impact on these differences is caused by the GR register files. This is caused by the fact that an n-issue ρ-VEX processor should have n-port register file memories, which can not be instantiated as Xilinx primitive elements. All ρ-VEX configurations were synthesized to work correctly with a maximum clock frequency of 89.44 MHz.

8.2 Main Contributions

The following contributions can be assigned to our project:

- **ρ-VEX** – We delivered an open source, reconfigurable and extensible and VLIW processor. The processor architecture is extensible in many ways, by means of an extensible issue-width, register file sizes and available FUs. Reconfigurable operations are supported by the means of ρ-OPS.
- **ρ-ASM** – A custom assembler/instruction ROM generator was created, to translate VEX assembly code to an executable program for ρ-VEX.
- **Application Development Framework** – A development framework is presented to easily write applications to be executed on ρ-VEX, or make adaptations to ρ-VEX' hardware design. A UART interface was created to present human-readable user feedback about the memory contents and executed cycles.
- **Performance/pay-off analysis for MOLEN** – To justify the existence of a VEX VLIW co-processor within a MOLEN machine, we performed a performance analysis. The relative performance of different VEX machine configurations were compared to a MOLEN CCU and to the PowerPC 405 GPP within the current MOLEN prototype.

8.3 Future Work

Recommended future work for the project can be divided in several sections, organized by the four main contributions. The author of this thesis is planning to continue working on several parts within this list as a hobby project.

ρ-VEX

- Investigate whether pipelining and variations of pipeline and memory latencies would help the overall performance. Because the VEX machine model is already parametric in these areas, we already possess a very flexible framework to exploit such architectural trade-offs.
- Add General-Purpose Input/Output (GPIO) ports to support easy communication to peripherals.
- Wishbone [13] bus connectivity could be added to support communication with many Wishbone-compliant (open source) cores from OpenCores.

- Because in our current implementation the GR register file consumes a significant amount of area, it might be an interesting trade-off to have a multi-cluster machine configuration consisting of single-issue ρ-VEX cores instead of one single-cluster, multi-issue ρ-VEX machine. As the VEX compiler has the ability to schedule data moves across VEX clusters, we already have architectural support for this.
- Support for changing ρ-OPS definitions at runtime, by means of partial reconfigurability could be investigated. This would make ρ-VEX a MOLEN replacement instead of an addition.

ρ-ASM

- Support the output of a `.coe` file to initialize the BRAMs. This would imply that re-synthesization of ρ-VEX is not needed after a change of instruction memory contents.
- Support for ρ-OPS and `.fmm` configuration files, instead of changing these parameters directly in the -ASM source files.
- Support for data memory initialization. Currently, only the instruction memory is initialized.
- Auto extract indicated application-kernel functions from VEX compiler assembly output, and generate a stand-alone wrapper around them. Currently, this has to be done manually.

Development Framework

- 'Boot loader' functionality – Support ρ-VEX executables to be uploaded to the instruction memory via the UART interface. This is another method to prevent completely re-synthesizing ρ-VEX when another application is loaded.

MOLEN Integration

- At the time of this project, the available MOLEN prototype exposed unmet timing constraints. These timing constraints did not influence the example CCUs as they came bundled with the prototype. However, as soon as the examples were modified to work on a larger dataset, unpredicted behaviour was exposed. This was also the case when we integrated an early RISC version of ρ-VEX in a custom CCU. To be able to successfully integrate ρ-VEX inside a CCU, these timing constraints should be solved. This could probably be done by adding some extra registers in-between the CCU and ρ-VEX interfaces.

Bibliography

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

[2] G. Kuzmanov, G. Gaydadjiev, and S. Vassiliadis, "The Molen Media Processor: Design and Evaluation," in *Proceedings of the International Workshop on Application Specific Processors, WASP 2005*, September 2005, pp. 26 – 33.

[3] P. Faraboschi, G. Brown, J.A.Fisher, G. Desoli, and F. Homewood, "Lx: A Technology Platform for Customizable VLIW Embedded Processing," in *Proceedings of the 27th annual International Symposium of Computer Architecture*, June 2000, pp. 203 – 213.

[4] K. Compton and S. Hauck, "Reconfigurable Computing: A Survey of Systems and Software," in *ACM Computing Surveys*, vol. 34, no. 2, May 2002, pp. 171 – 210.

[5] S. Vassiliadis, S. Wong, G. N. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. M. Panainte, "The Molen Polymorphic Processor," *IEEE Transactions on Computers*, pp. 1363 – 1375, November 2004.

[6] S. Vassiliadis, S. Wong, and S. D. Cotofana, "The MOLEN $\rho\mu$-Coded Processor," in *in 11th International Conference on Field-Programmable Logic and Applications (FPL), Springer-Verlag Lecture Notes in Computer Science (LNCS) Vol. 2147*, August 2001, pp. 275 – 285.

[7] G. Kuzmanov, G. Gaydadjiev, and S. Vassiliadis, "The Virtex II Pro MOLEN Processor," in *Proceedings of the 4th International Workshop on Computer Systems: Architectures, Modelling, and Simulation (SAMOS 2004)*, July 2004, pp. 192 – 202.

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

[9] GNU. GNU General Public License v3. [Online]. Available: http://www.gnu.org/licenses/gpl.html

[10] Xilinx, Inc. MicroBlaze Processor. [Online]. Available: http://www.xilinx.com/microblaze

[11] Altera. Nios II. [Online]. Available: http://www.altera.com/nios2

[12] OpenCores. OpenRISC 1200. [Online]. Available: http://www.opencores.org/projects.cgi/web/or1k/openrisc_1200

[13] OpenCores. Wishbone System-on-Chip (SoC) Interconnect Architecture. [Online]. Available: http://www.opencores.org/projects.cgi/web/wishbone/wishbone

[14] Lattice Semiconductor. LatticeMico32. [Online]. Available: http://www.latticesemi.com/products/intellectualproperty/ipcores/mico32/index.cfm

[15] K. Schleisiek. (2004) MicroCore. [Online]. Available: http://www.microcore.org/

[16] C. Iseli and E. Sanchez, "Spyder: a Reconfigurable VLIW Processor using FPGAs," in *FPGAs for Custom Computing Machines*, January 1993, pp. 17 – 24.

[17] C. Iseli and E. Sanchez, "Spyder: A SURE (SUperscalar and REconfigurable) processor," *The Journal of Supercomputing*, vol. 9, no. 3, pp. 231 – 252, September 1995.

[18] V. Brost, F. Yang, and M. Paindavoine, "A modular VLIW Processor," in *IEEE International Symposium on Circuits and Systems, ISCAS 2007.*, April 2007, pp. 3968 – 3971.

[19] A. Lodi, M. Toma, F. Campi, A. Cappelli, and R. Canegallo, "A VLIW Processor with Reconfigurable Instruction Set for Embedded Applications," in *IEEE Journal on Solid-State Circuits*, vol. 38, no. 11, January 2003, pp. 1876 – 1886.

[20] A. K. Jones, R. Hoare, D. Kusic, J. Fazekas, and J. Foster, "An FPGA-based VLIW Processor with Custom Hardware Execution," in *FPGA '05: Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field Programmable Gate Arrays*. New York, NY, USA: ACM, 2005, pp. 107 – 117.

[21] W. Chu, R. Dimond, S. Perrott, S. Seng, and W. Luk, "Customisable EPIC Processor: Architecture and Tools," in *DATE '04: Proceedings of the conference on Design, automation and test in Europe*, vol. 3, February 2004.

[22] M. Schlansker and B. Rau, "EPIC: Explicitly Parallel Instruction Computing," in *Computer*, vol. 33, no. 2, February 2000, pp. 37 – 45.

[23] M. Koester, W. Luk, and G. Brown, "A Hardware Compilation Flow For Instance-Specific VLIW Cores," in *Proceedings of the 18th International Conference on Field Programmable Logic and Applications (FPL08)*, September 2008.

[24] S. Hauck, T. Fry, M. Hosler, and J. Kao, "The Chimaera Reconfigurable Functional Unit," in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 12, no. 2, February 2004.

[25] T. Callahan, J. Hauser, and J. Wawrzynek, "The Garp Architecture and C Compiler," *Computer*, vol. 33, no. 4, pp. 62 – 69, Apr 2000.

[26] A. Peleg, U. Weiser, I. Center, and I. Haifa, "MMX Technology Extension to the Intel Architecture," *Micro, IEEE*, vol. 16, no. 4, pp. 42–50, August 1996.

[27] H. Corporaal, *Microprocessor Architectures: from VLIW to TTA*. John Wiley & Sons Ltd, 1998.

[28] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, 3rd ed. Morgan Kaufmann, 2003.

[29] Delft University of Technology Computer Engineering Laboratory. Delft Workbench. [Online]. Available: http://ce.et.tudelft.nl/DWB/

[30] Delft University of Technology Computer Engineering Laboratory. Molen Prototype Examples. [Online]. Available: http://ce.et.tudelft.nl/MOLEN/Prototype/

[31] International Telecommunication Union. G.723 Audio Codec. [Online]. Available: http://www.itu.int/rec/T-REC-G.723/e

[32] Sun Microsystems. G.723 Software Implementation. [Online]. Available: http://www.sun.com/

[33] E. Moore, "Gedanken-experiments on Sequential Machines," in *Automata Studies,Annals of Mathematical Studies*, vol. 34. Princeton University Press, 1956, pp. 129 – 153.

[34] G. Mealy, "A Method for Synthesizing Sequential Circuits," *Bell System Technology Journal*, vol. 34, pp. 1045 – 1079, September 1955.

[35] P. Wilcox, *Professional Verification – A Guide to Advanced Functional Verification.* Kluwer Academic Publishers, 2004.

[36] S. Lee, *Advanced Digital Logic Design – Using VHDL, State Machines, and Synthesis for FPGAs.* Thomson Engineering, 2006.

[37] Xilinx, Inc. XST User Guide 8.1i. [Online]. Available: http://www.xilinx.com/support/documentation/

[38] LEGO. LEGO Digital Designer. [Online]. Available: http://ldd.lego.com/

[39] S. Wong, T. van As, and G. Brown, "ρ-VEX: A Reconfigurable and Extensible Softcore VLIW Processor," *International Conference on Field-Programmable Technology, 2008. ICFPT 2008.*, December 2008.

List of Acronyms

ALU Arithmetic Logic Unit

API Application Programming Interface

ASCII American Standard Code for Information Interchange

ASIC Application-Specific Integrated Circuit

BR Branch Register

CAD Computer-Aided Design

CCU Custom Computing Unit

CPU Central Processing Unit

CTRL Control unit

EDA Electronic Design Automation

EDK Embedded Development Kit

EPIC Explicitly Parallel Instruction Computing

FSM Finite State Machine

FPGA Field-Programmable Gate Array

FU Functional Unit

FVP Functional Virtual Prototype

GPIO General-Purpose Input/Output

GPP General-Purpose Processor

GR General-purpose Register

HDL Hardware Description Language

IDE Integrated Development Environment

ILP Instruction Level Parallelism

ISA Instruction Set Architecture

ISE Integrated Synthesis Environment

MEM Memory unit

MUL Multiplier unit

PC Program Counter

RISC Reduced Instruction Set Computer

SIMD Single Instruction, Multiple Data

SMT Simultaneous Multithreading

UART Universal Asynchronous Receiver/Transmitter

UCF User Constraints File

UVM Unified Verification Methodology

VEX VLIW Example

VHDL VHSIC Hardware Description Language

VHSIC Very High Speed Integrated Circuit

VLIW Very Long Instruction Word

XST Xilinx Synthesis Technology

A Name and Logo

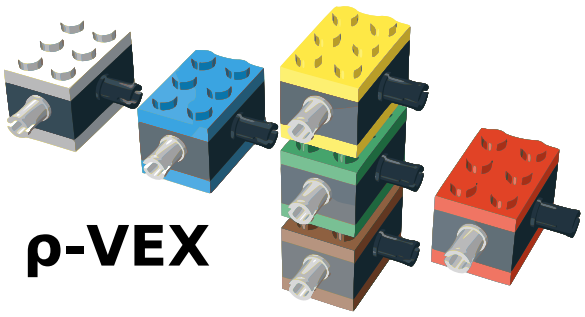


Figure A.1: ρ-VEX logo

The name ρ-VEX stands for 'reconfigurable VEX' processor. Because the letter *Rho* (P or ρ) is the Greek analogous for the Roman R or r, ρ-VEX is pronounced as *r-VEX*. This is also the correct spelling when no Greek letters can be used (for example in ASCII documents).

ρ-VEX' logo as depicted in Figure A.1 is assembled by LEGO bricks. The LEGO bricks resemble the extensibility that ρ-VEX provides. A 1-issue slot with the 4-stage model used by ρ-VEX is depicted. The third stage (the *execute* stage) allows placement of additional Functional Units. Because ρ-VEX can have a variable issue-width, more issue slots can be connected in parallel to the one depicted. Because we present an open source design, the individual functional blocks can be altered as well, as is the case in the LEGO analogy. The logo was created using the LEGO Digital Designer [38] tool.

VEX Operations & Semantics

B

ALU Operations

Operation	Type	Opcode	Mnemonic	Description
ADD	I	1000001	ALU_ADD	Add
AND	I	1000011	ALU_AND	Bitwise AND
ANDC	I	1000100	ALU_ANDC	Bitwise complement and AND
MAX	I	1000101	ALU_MAX	Maximum signed
MAXU	I	1000110	ALU_MAXU	Maximum unsigned
MIN	I	1000111	ALU_MIN	Minimum signed
MINU	I	1001000	ALU_MINU	Minimum unsigned
OR	I	1001001	ALU_OR	Bitwise OR
ORC	I	1001010	ALU_ORC	Bitwise complement and OR
SH1ADD	I	1001011	ALU_SH1ADD	Shift left 1 and add
SH2ADD	I	1001100	ALU_SH2ADD	Shift left 2 and add
SH3ADD	I	1001101	ALU_SH3ADD	Shift left 3 and add
SH4ADD	I	1001110	ALU_SH4ADD	Shift left 4 and add
SHL	I	1001111	ALU_SHL	Shift left
SHR	I	1010000	ALU_SHR	Shift right signed
SHRU	I	1010001	ALU_SHRU	Shift right unsigned
SUB	VI	1010010	ALU_SUB	Subtract
SXTB	VII	1010011	ALU_SXTB	Sign extend byte
SXTH	VII	1010100	ALU_SXTH	Sign extend half word
ZXTB	VII	1010101	ALU_ZXTB	Zero extend byte
ZXTH	VII	1010110	ALU_ZXTH	Zero extend half word
XOR	I	1010111	ALU_XOR	Bitwise XOR
MOV	VII	1011000	ALU_MOV	Copy `s1` to other location
CMPEQ	II	1011001	ALU_CMPEQ	Compare: equal
CMPGE	II	1011010	ALU_CMPGE	Compare: greater equal signed
CMPGEU	II	1011011	ALU_CMPGEU	Compare: greater equal unsigned
CMPGT	II	1011100	ALU_CMPGT	Compare: greater signed
CMPGTU	II	1011101	ALU_CMPGTU	Compare: greater unsigned
CMPLE	II	1011110	ALU_CMPLE	Compare: less than equal signed
CMPLEU	II	1011111	ALU_CMPLEU	Compare: less than equal unsigned
CMPLT	II	1100000	ALU_CMPLT	Compare: less than signed
CMPLTU	II	1100001	ALU_CMPLTU	Compare: less than unsigned
CMPNE	II	1100010	ALU_CMPNE	Compare: not equal
NANDL	II	1100011	ALU_NANDL	Logical NAND
NORL	II	1100100	ALU_NORL	Logical NOR
ORL	II	1100110	ALU_ORL	Logical OR
MTB	V	1100111	ALU_MTB	Move GR to BR
ANDL	II	1101000	ALU_ANDL	Logical AND
ADDCG	IV	1111---	ALU_ADDCG	Add with carry and generate carry.
DIVS	IV	1110---	ALU_DIVS	Division step with carry and generate carry
SLCT	III	0111---	ALU_SLCT	Select `s1` on true condition. (exceptional opcode)
SLCTF	III	0110---	ALU_SLCTF	Select `s1` on false condition. (exceptional opcode)
N/A	N/A	1000000	N/A	ALU ρ-OP

N/A	N/A	1000010	N/A	ALU ρ-OP
N/A	N/A	1100101	N/A	ALU ρ-OP
N/A	N/A	1101001	N/A	ALU ρ-OP
N/A	N/A	1101010	N/A	ALU ρ-OP
N/A	N/A	1101011	N/A	ALU ρ-OP
N/A	N/A	1101100	N/A	ALU ρ-OP
N/A	N/A	1101101	N/A	ALU ρ-OP
N/A	N/A	1101110	N/A	ALU ρ-OP
N/A	N/A	1101111	N/A	ALU ρ-OP

MUL Operations

Operation	Type	Opcode	Mnemonic	Description
MPYLL	I	0000001	MUL_MPYLL	Multiply signed low 16 x low 16 bits
MPYLLU	I	0000010	MUL_MPYLLU	Multiply unsigned low 16 x low 16 bits
MPYLH	I	0000011	MUL_MPYLH	Multiply signed low 16 (s1) x high 16 (s2) bits
MPYLHU	I	0000100	MUL_MPYLHU	Multiply unsigned low 16 (s1) x high 16 (s2) bits
MPYHH	I	0000101	MUL_MPYHH	Multiply signed high 16 x high 16 bits
MPYHHU	I	0000110	MUL_MPYHHU	Multiply unsigned high 16 x high 16 bits
MPYL	I	0000111	MUL_MPYL	Multiply signed low 16 (s2) x 32 (s1) bits
MPYLU	I	0001000	MUL_MPYLU	Multiply unsigned low 16 (s2) x 32 (s1) bits
MPYH	I	0001001	MUL_MPYH	Multiply signed high 16 (s2) x 32 (s1) bits
MPYHU	I	0001010	MUL_MPYHU	Multiply unsigned high 16 (s2) x 32 (s1) bits
MPYHS	I	0001011	MUL_MPYHS	Multiply signed high 16 (s2) x 32 (s1) bits, shift left 16
N/A	N/A	0001100	N/A	MUL ρ-OP
N/A	N/A	0001101	N/A	MUL ρ-OP
N/A	N/A	0001110	N/A	MUL ρ-OP
N/A	N/A	0001111	N/A	MUL ρ-OP

CTRL Operations

Operation	Type	Opcode	Mnemonic	Description
GOTO[1]	XIII	0100001	CTRL_GOTO	Unconditional relative jump
IGOTO[1]	XIX	0100010	CTRL_IGOTO	Unconditional absolute indirect jump to link register
CALL[1]	XVI	0100011	CTRL_CALL	Unconditional relative call
ICALL[1]	XX	0100100	CTRL_ICALL	Unconditional absolute indirect call to link register
BR	VIII	0100101	CTRL_BR	Conditional relative branch on true condition
BRF	VIII	0100110	CTRL_BRF	Conditional relative branch on false condition
RETURN	XVII	0100111	CTRL_RETRN	Pop stack frame and goto link register
RFI	XIV	0101000	CTRL_RFI	Return from interrupt
XNOP	XV	0101001	CTRL_XNOP	Multicycle NOP
N/A	N/A	0100000	N/A	CTRL ρ-OP
N/A	N/A	0101100	N/A	CTRL ρ-OP
N/A	N/A	0101101	N/A	CTRL ρ-OP
N/A	N/A	0101110	N/A	CTRL ρ-OP
N/A	N/A	0101111	N/A	CTRL ρ-OP

MEM Operations

Operation	Type	Opcode	Mnemonic	Description
LDW	X	0010001	MEM_LDW	Load word
LDH	X	0010010	MEM_LDH	Load halfword signed
LDHU	X	0010011	MEM_LDHU	Load halfword unsigned
LDB	X	0010100	MEM_LDB	Load byte signed
LDBU	X	0010101	MEM_LDBU	Load byte unsigned
STW	XI	0010110	MEM_STW	Store word
STH	XI	0010111	MEM_STH	Store halfword
STB	XI	0011000	MEM_STB	Store byte
PFT	XII	0011001	MEM_PFT	Prefetch
N/A	N/A	0011010	N/A	MEM ρ-OP
N/A	N/A	0011011	N/A	MEM ρ-OP
N/A	N/A	0011101	N/A	MEM ρ-OP
N/A	N/A	0011110	N/A	MEM ρ-OP
N/A	N/A	0010000	N/A	MEM ρ-OP

Miscellaneous Operations

Operation	Type	Opcode	Mnemonic	Description
STOP	XIV	0011111	STOP	STOP operation
NOP	XIV	0000000	NOP	No operation
SEND[2]	IX	0101010	INTR_SEND	Send `s1` to the path identified by `path`
RECV[2]	XVIII	0101011	INTR_RECV	Assigns the value from the path identified by `path` to `t`
N/A	N/A	0011100	SYL_FOLLOW	Syllable contains second half of long immediate

VEX Assembly Semantics

Type	Semantics
I	`operation $r0.t = $r0.s1, $r0.s2` `operation $r0.t = $r0.s1, s2`

II	`operation $r0.t = $r0.s1, $r0.s2`
	`operation $r0.t = $r0.s1, s2`
	`operation $b0.b = $r0.s1, $r0.s2`
	`operation $b0.b = $r0.s1, s2`

III	`operation $r0.t = $b0.b1, $r0.s1, $r0.s2` `operation $r0.t = $b0.b1, $r0.s1, s2`
IV	`operation $r0.t, $b0.b = $b0.b1, $r0.s1, $r0.s2`
V	`operation $b0.b = $r0.s1`
VI	`operation $r0.t = $r0.s2, $r0.s1` `operation $r0.t = s2, $r0.s1`
VII	`operation $r0.t = $r0.s1`
VIII	`operation $b0.b, label`
IX	`operation $r0.s1, path`
X	`operation $r0.t = offset[$r0.s1]`
XI	`operation offset[$r0.s1] = $r0.s2`

XII	`operation offset[$r0.s1]`
XIII	`operation label` `operation $r0.lr`
XIV	`operation`
XV	`operation n`
XVI	`operation $l0.t = label` `operation $l0.t = $l0.t`
XVII	`operation $r0.t = $r0.t, label, $r0.lr`
XVIII	`operation $r0.t = path`
XIX	`operation $r0.lr`
XX	`operation $l0.t = $l0.t`

Symbol Declarations

`operation`	The operation as presented in one of the lists above, lowercase
`$r0.*`	GR register where `*` can be `t` (target), `s1` (source 1), `s2` (source 2) or `lr` (link register)
`$b0.*`	BR register where `*` can be `b` (target) or `b1` (source 1)
`s2`	Immediate operand (without preceeding `$r0.`)
`label`	Label to jump to when branching
`offset`	Offset used when loading or storing to data memory
`n`	Number of cycles (in **XNOP**)
`path`	Path for sending/receiving inter-cluster contents (unsupported in ρ-VEX)

[1] According a to a message on the VEX forum, the **CALL** & **ICALL** and **GOTO** & **IGOTO** operations are overloaded in the latest VEX compiler, so **ICALL** and **IGOTO** are obsolete. These are still supported in its original form to stay compliant to the original VEX ISA.

See `http://www.vliw.org/vex/viewtopic.php?t=52` for more information.

[2] The inter-cluster operations **SEND** and **RECV** are not supported by ρ-VEX, but their opcodes are reserved.

C

ρ-VEX Machine Model

The VEX machine model (in `.fmm` format) as used in our benchmarks is presented below. It should be noted that this is the default VEX machine configuration, except for the number of General-purpose Register (GR). This number has been scaled down to 32 (instead of the default value 64), because this causes the synthesization time to be halved and the on-die area usage to be lowered significantly (because implementing an 8-port register file is very inefficient in FPGA logic slices).

```
RES: IssueWidth 4
RES: MemLoad 1
RES: MemStore 1
RES: MemPft 1
RES: IssueWidth.0 4
RES: Alu.0 4
RES: Mpy.0 2
RES: CopySrc.0 1
RES: CopyDst.0 1
RES: Memory.0 1
DEL: AluR.0 0
DEL: Alu.0 0
DEL: CmpBr.0 1
DEL: CmpGr.0 0
DEL: Select.0 0
DEL: Multiply.0 1
DEL: Load.0 2
DEL: LoadLr.0 3
DEL: Store.0 0
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 1
DEL: CpBrGr.0 0
DEL: CpGrLr.0 2
```

```
DEL: CpLrGr.0 0
DEL: Spill.0 0
DEL: Restore.0 2
DEL: RestoreLr.0 3
REG: $r0 31
REG: $b0 8
```

Fibonacci Benchmark Assembly Code

D

D.1 1-Issue VEX Assembly Code

```
1  # Fibonacci Sequence demo
2  # -----------------------
3  # Copyright (c) 2008, Thijs van As
4  #
5  # Calculates 45th Fibonacci number, and stores it in data memory
6  # at address 0x00
7  #
8  # 1-issue version
9
10 add $r0.1 = $r0.0, 1
11 ;;
12 mov $r0.2 = $r0.0
13 ;;
14 mov $r0.3 = $r0.0
15 ;;
16 mov $r0.4 = $r0.0
17 ;;
18 add $r0.6 = $r0.0, 45
19 ;;
20 mtb $b0.7 = $r0.0
21 ;;
22 LABEL_BEGIN:
23 br $b0.7, LABEL_END
24 ;;
25 cmpeq $b0.7 = $r0.3, $r0.6
26 ;;
27 add $r0.3 = $r0.3, 1
28 ;;
29 mov $r0.4 = $r0.1
30 ;;
31 add $r0.1 = $r0.1, $r0.2
32 ;;
33 mov $r0.2 = $r0.4
34 ;;
35 goto LABEL_BEGIN
36 ;;
37 LABEL_END:
38 add $r0.15 = $r0.0, 1
39 ;;
40 stw 0x0[$r0.15] = $r0.1
41 ;;
```

D.2 2-Issue VEX Assembly Code

```
1  # Fibonacci Sequence demo
2  # -----------------------
3  # Copyright (c) 2008, Thijs van As
4  #
5  # Calculates 45th Fibonacci number, and stores it in data memory
6  # at address 0x00
7  #
8  # 2-issue version
9
10 add $r0.15 = $r0.0, 1
11 add $r0.10 = $r0.0, 45
12 ;;
13 mov $r0.1 = $r0.0
14 add $r0.2 = $r0.0, 1
15 ;;
16 LABEL_BEGIN:
17 cmpeq $b0.0 = $r0.9, $r0.10
18 br $b0.7, LABEL_END
19 ;;
20 add $r0.2 = $r0.1, $r0.2
21 add $r0.3 = $r0.0, $r0.2
22 ;;
23 add $r0.9 = $r0.9, 1
24 add $r0.1 = $r0.0, $r0.3
25 ;;
26 goto LABEL_BEGIN
27 ;;
28 stw 0x0[$r0.15] = $r0.1
29 ;;
```

D.3 4-Issue VEX Assembly Code

```
1  # Fibonacci Sequence demo
2  # -----------------------
3  # Copyright (c) 2008, Thijs van As
4  #
5  # Calculates 45th Fibonacci number, and stores it in data memory
6  # at address 0x00
7
8  add $r0.15 = $r0.0, 1
9  mov $r0.1 = $r0.0
10 add $r0.10 = $r0.0, 44      # 44 + 1 iterations
11 add $r0.2 = $r0.0, 1
12 ;;
13 LABEL_BEGIN:
14 add $r0.2 = $r0.1, $r0.2
15 add $r0.3 = $r0.0, $r0.2
16 cmpeq $b0.0 = $r0.9, $r0.10 # if ($r0.9 == $r0.10) $b0.0 = 1;
17 br $b0.0, LABEL_END         # if ($b0.0 == 1) goto LABEL_END;
18 ;;
19 add $r0.9 = $r0.9, 1
20 add $r0.1 = $r0.0, $r0.3
21 goto LABEL_BEGIN
```

```
22 ;;
23 LABEL_END:
24 stw 0x0[$r0.15] = $r0.1
25 mov $r0.9 = $r0.0
26 ;;
```

D.4 4-Issue VEX Assembly Code With ρ-OPS

```
1  # Fibonacci Sequence demo
2  # -----------------------
3  # Copyright (c) 2008, Thijs van As
4  #
5  # Calculates 45th Fibonacci number, and stores it in data memory
6  # at address 0x00
7  #
8  # r-OPS FIB3 and FIB4
9
10 add $r0.15 = $r0.0, 1
11 mov $r0.1 = $r0.0
12 add $r0.10 = $r0.0, 9
13 add $r0.2 = $r0.0, 1
14 ;;
15 LABEL_BEGIN:
16 fib4 $r0.2 = $r0.1, $r0.2
17 fib3 $r0.3 = $r0.1, $r0.2
18 cmpeq $b0.0 = $r0.9, $r0.10 # if ($r0.9 == $r0.10) $b0.0 = 1;
19 br $b0.0, LABEL_END         # if ($b0.0 == 1) goto LABEL_END;
20 ;;
21 add $r0.9 = $r0.9, 1
22 add $r0.1 = $r0.0, $r0.3
23 goto LABEL_BEGIN
24 ;;
25 LABEL_END:
26 stw 0x0[$r0.15] = $r0.1
27 mov $r0.9 = $r0.0
28 ;;
```

Fibonacci Benchmark Simulation Waveforms

E

The waveforms presented in this Appendix resulted from the benchmark simulations running the VEX assembly code for calculating the 45th Fibonacci number as presented in Section D.3 of Appendix D. Section E.1 presents the waveforms of the behavioural simulations and Section E.2 presents the waveforms of the post-place and route simulations.

The resulting value in the data memory at address 0 equals 0x43A53F82, which is the hexadecimal representation of the 45th Fibonacci number 1134903170. It should be noted that only selected signals are presented in the waveforms, to avoid losing overview.

E.1 Behavioural Simulation Waveforms

The simulation results presented in Figure E.1 depict the first 1330 nanoseconds of the behavioural simulations of our benchmark with Fibonacci's Sequence. The first two iterations of the second and third instructions in Section D.3 are presented. Figure E.2 depicts the last 1330 nanoseconds of the behavioural simulations. The last iteration of the second and third instructions is presented, as well as the last instruction. It should be noted that ρ-ASM always adds a **STOP** operation together with 3 **NOP** operations in an instruction after the last instruction defined. This to make sure program execution ends.

The `instr` signal represents the 128-bit instruction fetched by ρ-VEX. `run` is the external signal that should be raised to invoke execution. `done` will be raised after execution is finished. `pc` and `pc_goto` represent the current and next value of the PC, respectively. `d_memory` represents the contents of the data memory.

Table E.1 maps the hexadecimal syllable codes from the waveforms to human-readable VEX assembly code. This resembles the assembly code in Section D.3.

E.2 Post-Place and Route Waveforms

Figures E.3 and E.4 depict the same transitions as Figures E.1 and E.2, but after a simulation that also simulated the electrical behaviour of the circuitry. Delays caused by logical path lengths and several heterogeneous components within the chosen XC2VP30 FPGAtechnology were taken into account.

The presented signal names are slightly different from the behavioural simulations. As the `pc_goto` signal is measured at another point in the design, this value differs sometimes from the behavioural simulations.

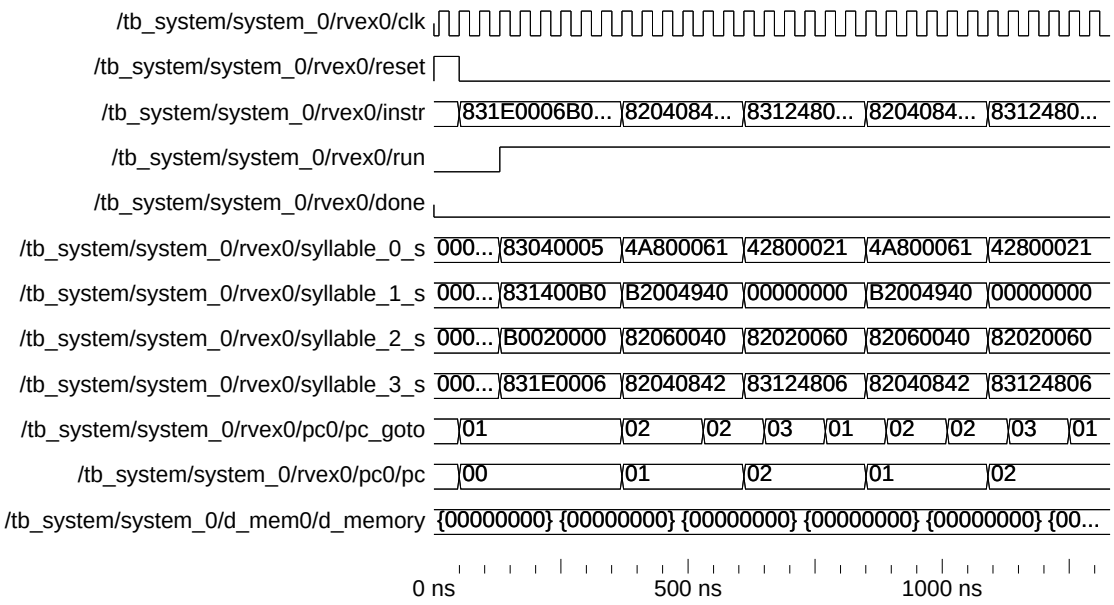


Figure E.1: Behavioural simulation: 0 – 1330 ns

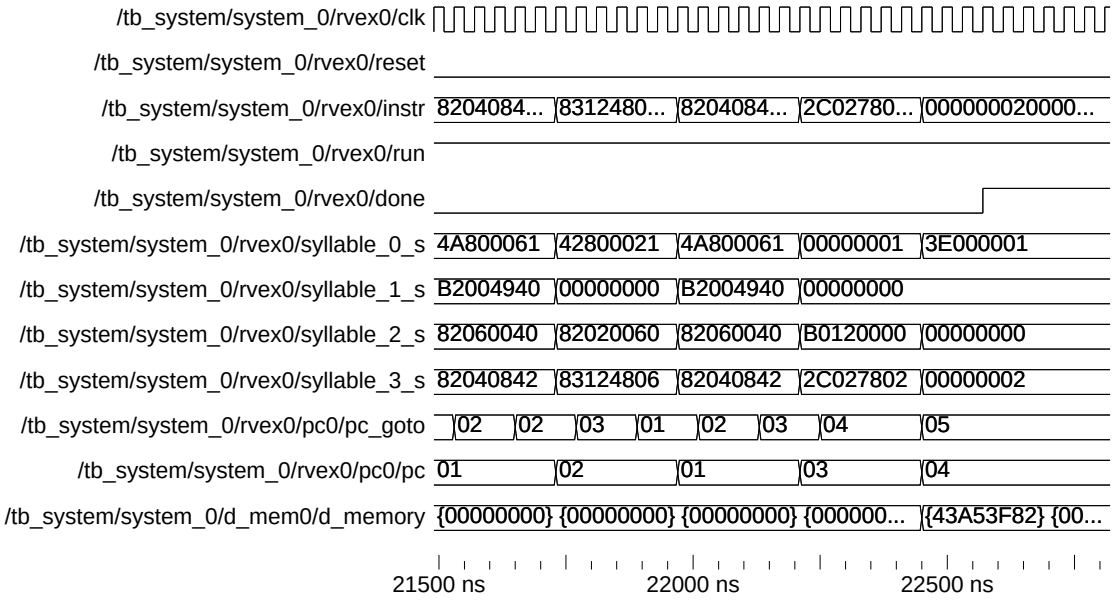


Figure E.2: Behavioural simulation: 21490 – 22820 ns

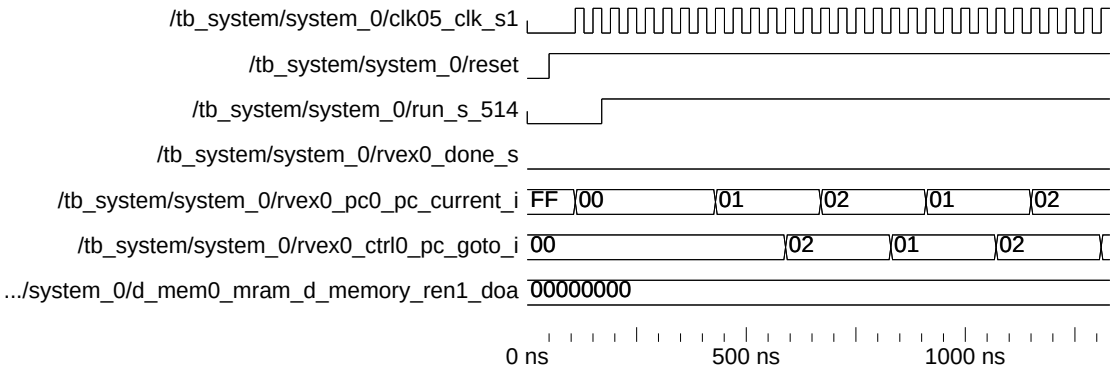


Figure E.3: Post-place and route simulation: 0 – 1330 ns

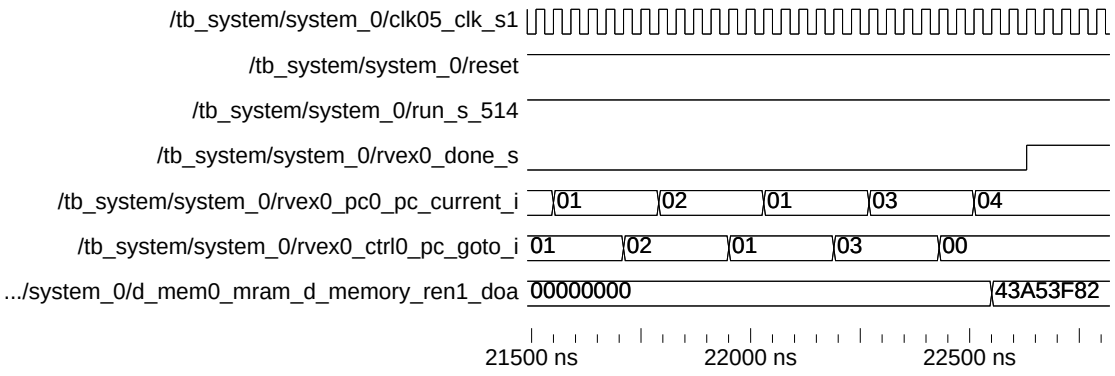


Figure E.4: Post-place and route simulation: 21490 – 22820 ns

Address	Syllable	VEX Semantics
00	0x831E0006	add $r0.15 = $r0.0, 1
	0xB0020000	mov $r0.1 = $r0.0
	0x831400B0	add $r0.10 = $r0.0, 44
	0x83040005	add $r0.2 = $r0.0, 1
01	0x82040842	add $r0.2 = $r0.1, $r0.2
	0x82060040	add $r0.3 = $r0.0, $r0.2
	0xB2004940	cmpeq $b0.0 = $r0.9, $r0.10
	0x4A800061	br $b0.0, 3
02	0x83124806	add $r0.9 = $r0.9, 1
	0x82020060	add $r0.1 = $r0.0, $r0.3
	0x00000000	nop
	0x42800021	goto 1
03	0x2C027802	stw 0x0[$r0.15] = $r0.1
	0xB0120000	mov $r0.9 = $r0.0
	0x00000000	nop
	0x00000001	nop
04	0x00000002	nop
	0x00000000	nop
	0x00000000	nop
	0x3E000001	stop

Table E.1: Syllables explained from the Fibonacci's Sequence benchmark

Quickstart Guide

F

This is a Quickstart Guide to easily deploy ρ-VEX, mainly focused on the utilization with a Xilinx University Program Virtex-II Pro FPGA board by Digilent. If you want to use this guide together with another FPGA platform, you should first add the definitions of your board to the workflow as described in Section F.6.

F.1 Requirements

- A Xilinx University Program Virtex-II Pro board (`http://www.digilentinc.com/Products/Detail.cfm?av1=Products&Nav2=Programmable&Prod=XUPV2P`)
- PC running Linux or Windows[1]
- Xilinx ISE Suite (tested with 8.1.03i, should work with later versions too)

F.2 Deploying ρ-VEX on an FPGA

1. Acquire the latest snapshot from the 'Downloads' section at the ρ-VEX website, or checkout the latest code from the Subversion repository. The project page can be found at `http://r-vex.googlecode.com/`.
2. Inside the `r-VEX/src/` directory, synthesize ρ-VEX by entering the following command:

   ```
   make v2p
   ```

3. After the synthesis process has completed, the generated bit-file can be uploaded to the FPGA board by entering:

   ```
   make fpga
   ```

4. Connect a serial cable to the RS-232 interface on the XUP V2P board. Connect the other side of the cable to a PC and start a terminal application, like *Minicom* (Linux), *Putty* or *Hyperterminal* (Windows). Connect using the following settings: *115200 bps transfer rate, 8 data bits, no parity*

[1] When you want to make use of the Makefile method described below on a Windows machine, a Cygwin installation should be present with GNU *Make*. Xilinx EDK automatically installs a version of Cygwin. However, some GNU tools like *cat* are not included. This results in an error while combining the individual log files after synthesis. This can be safely ignored.

5. Press button SW2 on the XUP V2P board, which acts as the reset button. In the terminal application, you will see the contents of the first 16 data memory addresses, as well as a cycle counter.

By default, an application to calculate the 45th Fibonacci number is loaded and synthesized. The VEX assembly source code of this application can be found in Section D.3 of Appendix D, or in the `demos/` directory. The output of ρ-VEX transmitted over the UART will be the following:

```
r-VEX
-----
Cycles: 0x00000231

Data memory dump

addr | contents
-----+-----------
0x00 | 0x43A53F82
0x01 | 0x00000000
0x02 | 0x00000000
0x03 | 0x00000000
0x04 | 0x00000000
0x05 | 0x00000000
0x06 | 0x00000000
0x07 | 0x00000000
0x08 | 0x00000000
0x09 | 0x00000000
0x0A | 0x00000000
0x0B | 0x00000000
0x0C | 0x00000000
0x0D | 0x00000000
0x0E | 0x00000000
0x0F | 0x00000000
```

F.3 Assembling and Running Code

The instruction memory ROM can be found in the file `i_mem.vhd`. A new instruction ROM file can be generated by the ρ-ASM tool. This tool requires a UNIX operating system with the GNU C libraries to be compiled.

1. To compile ρ-ASM, go to the `r-ASM/src/` directory, and enter the command

   ```
   make
   ```

2. To assemble a VEX assembly file, run ρ-ASM by entering the following command:

```
./rasm <source.s>
```

To see more options of ρ-ASM (like enabling debug output) run the application without any arguments. Some demo applications with their corresponding outputs can be found in the `demos/` directory.

3. By default, the resulting instruction ROM is written to `i_mem.vhd` in the current directory. Copy or move this file to the ρ-VEX source directory. When using the standard repository structure, this can be accomplished by entering

```
mv i_mem.vhd ../../r-VEX/src/
```

4. Now, repeat the steps from the Quickstart in Section F.2.

F.4 Using ρ-OPS

This guide on adding ρ-OPS to an implementation of ρ-VEX is based on the example of adding the **FIB3** and **FIB4** operations, as used in our benchmark in Chapter 7.

Adapting ρ-VEX

1. Determine what FU will execute the operation. An unused ρ-OPS opcode can be obtained from Appendix B.

2. In `r-vex_pkg.vhd`, constant definitions should be added for the new operations.

```
1 constant ALU_FIB4    : std_logic_vector(6 downto 0) := "1000010";
2 constant ALU_FIB3    : std_logic_vector(6 downto 0) := "1101100";
```

3. In the determined FU, support for the new operations should be added in the corresponding VHDL file. In our benchmark, lines 9 – 12 are added in `alu.vhd` from the code snippet below.

```
1  alu_control : process(clk, reset)
2  begin
3      if (reset = '1') then
4          out_valid <= '0';
5          result_s <= (others => '0');
6          cout_s <= '0';
7      elsif (clk = '1' and clk'event) then
8          ...
9          elsif std_match(aluop, ALU_FIB4) then
10             result_s <= f_FIB4 (src1, src2);
11         elsif std_match(aluop, ALU_FIB3) then
12             result_s <= f_FIB3 (src1, src2);
13         ...
14     end if;
15 end process alu_control;
```

4. Functions and function prototypes should be added to the corresponding file. In our example, the code below was added to `alu_operations.vhd`.

```
1  function f_FIB4   ( s1, s2 : std_logic_vector(31 downto 0))
2                      return std_logic_vector;
3
4  function f_FIB3   ( s1, s2 : std_logic_vector(31 downto 0))
5                      return std_logic_vector;
6
7  ...
8
9  function f_FIB4   ( s1, s2 : std_logic_vector(31 downto 0))
10                     return std_logic_vector is
11 begin
12     return (s1 + s2 + s2 + s1 + s2 + s2 + s1 + s2);
13 end function f_FIB4;
14
15 function f_FIB3   ( s1, s2 : std_logic_vector(31 downto 0))
16                     return std_logic_vector is
17 begin
18     return (s1 + s2 + s2 + s1 + s2);
19 end function f_FIB3;
```

Adapting ρ-ASM

To adapt ρ-ASM to support the new ρ-OPS, two small additions should be made in `syllable.h`. A new opcode define should be made, as well as an addition of the new opcode's mnemonic to the `operation_table` lookup table (lines 11 – 12).

```
1  #define FIB3   108
2  #define FIB4    66
3
4  ...
5
6  static struct operation_t {
7      const char *operation;
8      int opcode;
9  } operation_table[] = {
10     ...
11     { "fib3",  FIB3   },
12     { "fib4",  FIB4   },
13     ...
14 };
```

F.5 Running ModelSim Simulations

To run ModelSim simulations within the workflow, you should have installed a version of Mentor Graphics ModelSim (we simulated with the SE 6.3d edition). To run the simulations using the system wrapper testbench `r-VEX/testbenches/tb_system.vhd` (which simulates the execution of the current instruction memory in `i_mem.vhd`), enter the following command in `r-VEX/src/`:

```
make modelsim
```

F.6 Adding Support For Other FPGA Boards

To add support for other FPGA boards with Xilinx a FPGA in this workflow, the file `r-VEX/src/Makefile` should be edited. The following changes should be applied:

1. Choose a mnemonic for your new target. For example, we used `v2p` for the XUP V2P board containing a Virtex-II Pro FPGA. This mnemonic will be referred to as `<mnemonic>`. Check the Xilinx identifier for this FPGA. For example, the identifier for the Virtex-II Pro chip we used is *xc2vp30-ff896-7*. This identifier will be referred to as `<xilinx_identifier>`.

2. Add the Xilinx FPGA identifier as a new variable. Use the following template:

   ```
   XIL_PART_r-vex_<mnemonic> = <xilinx_identifier>
   ```

3. Add a 'help' text to the `default` *Make* target.

4. Create a new *Make* target according to this template:

   ```
   <mnemonic>: r-vex_<mnemonic>.deps r-vex_<mnemonic>.bit log_<mnemonic>.txt
   ```

5. Create a new *Make* target `r-vex_<mnemonic>.ut` and let it write your preferred options to the `.ut` file.[2]

6. Create a new *Make* target `r-vex_<mnemonic>.xst` and let it write your preferred options to the XST configuration file.[2]

7. Create a new *Make* target `r-vex_<mnemonic>.ucf` and let it write your UCF user constraints to the `.ucf` file.[2]

8. Create a new *Make* target `r-vex_<mnemonic>.cmd` and let it write the commands for Xilinx IMPACT to program your FPGA.[2]

[2] As a reference, you could use the *Make* target with the mnemonic `v2p`.

Package Contents

A listing of the directories and remarkable files as they can be found in the ρ-VEX package at `http://r-vex.googlecode.com/` is presented below, together with a description.

All ρ-VEX and ρ-ASM source code is released under the *GNU General Public License v3 (GPL)* [9].

General Overview

.	
`|-- demos/`	Demo VEX assembly applications
`|  '-- reference_outputs`	Reference outputs of demos
`|-- doc/`	Documentation files
`|  |-- diagrams/`	Some diagram `.svg` files as used in this thesis
`|  |-- logo/`	ρ-VEX logo files
`|  |-- instruction_layout.txt`	The ρ-VEX instruction layout description as presented in Chapter 4
`|  |-- quickstart_xupv2p.txt`	A Quickstart Guide
`|  '-- syllable_layout.txt`	The ρ-VEX syllable layout description as presented in Chapter 4
`|-- r-ASM/`	Root directory of ρ-ASM assembler
`|  '-- src/`	C source files of ρ-ASM
`'-- r-VEX/`	Root directory of ρ-VEX processor
`    |-- src/`	VHDL source files of ρ-VEX
`    '-- testbenches/`	Testbench for the system wrapper
`        '-- old/`	Old testbenches for individual Functional Unit (FU) simulations

ρ-VEX Source Files

.	
`|-- Makefile`	Automated synthesize, simulate and build instructions for *Make*
`|-- alu.vhd`	Arithmetic Logic Unit
`|-- alu_operations.vhd`	Individual ALU operations as functions
`|-- clk_div.vhd`	Clock divider to use XUP V2P's on-board clock directly
`|-- ctrl.vhd`	Control unit
`|-- ctrl_operations.vhd`	Individual CTRL operations as functions
`|-- d_mem.vhd`	Data memory
`|-- decode.vhd`	*Decode* stage
`|-- execute.vhd`	*Execute* stage
`|-- fetch.vhd`	*Fetch* stage
`|-- i_mem.vhd`	Instruction memory (generated by ρ-ASM)
`|-- mem.vhd`	Memory unit
`|-- mem_operations.vhd`	Individual MEM operations as functions
`|-- mul.vhd`	Multiplier unit
`|-- mul_operations.vhd`	Individual MUL operations as functions
`|-- pc.vhd`	Program Counter
`|-- r-vex.vhd`	ρ-VEX top-level entity
`|-- r-vex_pkg.vhd`	Common package with opcode definitions and parameters

`|-- registers_br.vhd`	General-purpose Register file
`|-- registers_gr.vhd`	Branch Register file
`|-- system.vhd`	Top-level system-wrapper entity including memories and UART
`|-- uart/`	Universal Asynchronous Receiver/Transmitter code
`|   |-- clk_18432.vhd`	Clock generator for 115200 bps data transmission
`|   |-- uart.vhd`	UART top-level entity and data memory interface
`|   |-- uart_pkg.vhd`	Common package with UART functions
`|   '-- uart_tx.vhd`	UART transmitter unit
`'-- writeback.vhd`	Writeback stage

ρ-ASM Source Files

`.`	
`|-- Makefile`	Automated compile instructions for *Make*
`|-- rasm.c`	The biggest part of ρ-ASM code
`|-- rasm.h`	Common parameter definitions
`|-- syllable.c`	Syllable-manipulating helper functions
`|-- syllable.h`	Common opcode definitions
`|-- util.c`	Common (I/O) helper functions
`|-- util.h`	`util.c` function prototypes
`|-- vhdl.c`	VHDL printing functions
`'-- vhdl.h`	`vhdl.c` function prototypes

Curriculum Vitae

Thijs van As was born on November 2, 1983 in Vlaardingen, The Netherlands. From 1996 to 2002 he attended the Stedelijk Gymnasium in Schiedam, where he obtained his VWO diploma. In September 2002 he started his Electrical Engineering studies at Delft University of Technology. After obtaining the Bachelor of Science degree, he continued his studies at the Computer Engineering Laboratory where he is hoping to receive the Master of Science degree.

His research interests include, but are not limited to: computer architecture, reconfigurable computing, microprocessor architectures, and embedded computing. His research on ρ-VEX resulted in a paper to be published on ICFPT 2008 [39].

Next to his studies, he is employed by ASK Community Systems in Rotterdam as a software developer for Internet applications. Apart from professional interest, he is also an enthusiastic hobby designer. In addition to several other hobby projects, he intends to proceed working on ρ-VEX after his graduation.