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

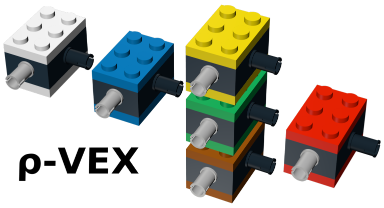
2016

MSc THESIS

A Dynamically Reconfigurable VLIW Processor and Cache Design with Precise Trap and Debug Support

J. van Straten

Abstract



CE-MS-2016-06

This thesis describes the design and implementation of a VLIW processor and associated caches based on the ρ-VEX concept. An ρ-VEX processor must be dynamically (runtime) reconfigurable to behave as a single large processor, two medium-sized processors, or four small processors. This allows a scheduler to optimize for energy and/or performance based on runtime information. The key challenge lies in translating this concept into actual working hardware. Note that reconfiguration must happen quickly for the increase in performance to outweigh the reconfiguration overhead. To accomplish this goal, a new processor and corresponding cache organization had to be developed, verified, and debugged. The dynamic reconfiguration concept used in the ρ-VEX processor is unique and, therefore, the following key components had to be designed: (1) a dynamic instruction cache that can service a single processor or multiple processors depending on the core configuration, (2) a dynamic data cache with coherency since we are dealing with multiple cores, (3) a reconfiguration control unit that synchronizes running threads before reallocating the computational and cache resources, and (4) a mechanism that allows state restoration after handling a trap. State restoration must be possible even if the configuration changed while the trap was being handled. This is an issue, because it is possible for a trap to interrupt a thread in intermediate states that would not normally occur in wider configurations. Reconfiguration takes only six clock cycles if there are no stalls from the memory subsystem, so overhead should be negligible.

On top of the base design of the dynamic ρ-VEX processor, the following features were implemented: (1) variable-length instruction support to decrease instruction cache pressure, (2) a debugging peripheral and accompanying tools, and (3) a trace unit for offline debugging and cache performance logging. Furthermore, many parameters of the processor can be selected at design-time using generics, such as the issue width, the degree of reconfigurability, and the layout and availability of the computational resources. Additionally, the pipeline configuration, instruction set encoding, and control register functionality can be configured using a VHDL code generator.

This work is intended to enable future research and development in dynamic processor design. It has already proven its value, as three MSc projects used the current design as a starting point, and four conference papers were published with results generated based on the current design. Finally, the processor will also serve as the basis for an ASIC design that is intended to be used in robotics and space applications.

Faculty of Electrical Engineering, Mathematics and Computer Science

A Dynamically Reconfigurable VLIW Processor and Cache Design with Precise Trap and Debug Support

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

J. van Straten
born in Zoetermeer, Netherlands

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

A Dynamically Reconfigurable VLIW Processor and Cache Design with Precise Trap and Debug Support

by J. van Straten

Abstract

This thesis describes the design and implementation of a VLIW processor and associated caches based on the ρ-VEX concept. An ρ-VEX processor must be dynamically (runtime) reconfigurable to behave as a single large processor, two medium-sized processors, or four small processors. This allows a scheduler to optimize for energy and/or performance based on runtime information. The key challenge lies in translating this concept into actual working hardware. Note that reconfiguration must happen quickly for the increase in performance to outweigh the reconfiguration overhead.

To accomplish this goal, a new processor and corresponding cache organization had to be developed, verified, and debugged. The dynamic reconfiguration concept used in the ρ-VEX processor is unique and, therefore, the following key components had to be designed: (1) a dynamic instruction cache that can service a single processor or multiple processors depending on the core configuration, (2) a dynamic data cache with coherency since we are dealing with multiple cores, (3) a reconfiguration control unit that synchronizes running threads before reallocating the computational and cache resources, and (4) a mechanism that allows state restoration after handling a trap. State restoration must be possible even if the configuration changed while the trap was being handled. This is an issue, because it is possible for a trap to interrupt a thread in intermediate states that would not normally occur in wider configurations. Reconfiguration takes only six clock cycles if there are no stalls from the memory subsystem, so overhead should be negligible.

On top of the base design of the dynamic ρ-VEX processor, the following features were implemented: (1) variable-length instruction support to decrease instruction cache pressure, (2) a debugging peripheral and accompanying tools, and (3) a trace unit for offline debugging and cache performance logging. Furthermore, many parameters of the processor can be selected at design-time using generics, such as the issue width, the degree of reconfigurability, and the layout and availability of the computational resources. Additionally, the pipeline configuration, instruction set encoding, and control register functionality can be configured using a VHDL code generator.

This work is intended to enable future research and development in dynamic processor design. It has already proven its value, as three MSc projects used the current design as a starting point, and four conference papers were published with results generated based on the current design. Finally, the processor will also serve as the basis for an ASIC design that is intended to be used in robotics and space applications.

Laboratory	:	Computer Engineering
Codenumber	:	CE-MS-2016-06

Committee Members :

Advisor:	Stephan Wong, CE, TU Delft
Chairperson:	Stephan Wong, CE, TU Delft
Member:	Koen Bertels, CE, TU Delft
Member:	Rene van Leuken, CAS, TU Delft
Member:	Said Hamdioui, CE, TU Delft

Dedicated to my family, friends, and colleagues

Contents

List of Figures xi

List of Tables xiii

List of Acronyms xv

Acknowledgements xvii

1 Introduction 1

1.1 Context 1

1.2 Current state of the ρ-VEX project 2

1.3 Problem statement and methodology 3

1.4 How to read this thesis 4

2 Background 5

2.1 FPGAs 5

2.1.1 Theory of operation 5

2.1.2 Softcore processors 6

2.1.3 Hardware description languages 6

2.1.4 ML605 and VC707 development boards 8

2.2 Processor architecture 9

2.2.1 The stack 9

2.2.2 Timing and pipelining 10

2.2.3 Traps 14

2.2.4 Debugging 15

2.2.5 Exploiting parallelism 17

2.2.6 ρ-VEX architecture 20

2.3 Memory architecture 23

2.3.1 Caches 23

2.3.2 Memory management units 26

2.3.3 Busses 27

2.3.4 Peripherals 28

2.4 Conclusion 28

3 Design 29

3.1 Control registers 29

3.2 Reconfiguration 30

3.2.1 Lane configurations 30

3.2.2 Context switching 31

3.2.3 Disabled lane groups 32

3.2.4 Requesting a reconfiguration 32

3.2.5 Configuration word encoding . . . 34
3.2.6 Decoding and committing configurations . . . 34
3.3 Trap control logic . . . 35
3.3.1 Trap identification . . . 35
3.3.2 Returning from a trap handler . . . 36
3.3.3 Nested traps . . . 36
3.3.4 Interrupts . . . 36
3.3.5 Dealing with runtime reconfiguration . . . 38
3.4 Debug interface . . . 39
3.4.1 Self-hosted vs. external debugging . . . 40
3.4.2 Breakpoints and watchpoints . . . 40
3.4.3 Reconfiguration and parallel programs . . . 41
3.4.4 External debug link and software . . . 41
3.4.5 Trace unit . . . 42
3.5 Variable-length bundles . . . 42
3.5.1 Bundle encoding . . . 43
3.5.2 Fetching bundles . . . 43
3.5.3 Syllable distribution . . . 44
3.6 Cache . . . 45
3.6.1 Instruction cache . . . 45
3.6.2 Data cache . . . 47
3.6.3 Coherence and consistency . . . 48
3.6.4 Performance measurement features . . . 50
3.7 Design-time configurability . . . 50
3.7.1 Bundle organization . . . 50
3.7.2 Functional unit resources . . . 51
3.7.3 Pipeline and forwarding . . . 51
3.7.4 Debug resources . . . 51
3.7.5 Opcode decoding . . . 52
3.7.6 Traps . . . 52
3.7.7 Control registers . . . 52
3.8 Conclusion . . . 53

4 Implementation **55**
4.1 Language choice and code style . . . 55
4.2 ρ-VEX processor . . . 56
4.2.1 Reconfiguration . . . 58
4.2.2 Datapath and pipeline . . . 62
4.2.3 Instruction fetch . . . 64
4.2.4 Instruction decode . . . 66
4.2.5 Operand read . . . 68
4.2.6 PC+1 computation . . . 68
4.2.7 PC-relative branch target . . . 69
4.2.8 Branch unit . . . 70
4.2.9 Execute . . . 72

4.2.10 Register files . . . 73
4.2.11 Trap handling . . . 74
4.2.12 Run control interface . . . 76
4.2.13 Tracing . . . 76
4.3 Cache . . . 78
4.3.1 Reconfigurable interconnect . . . 78
4.3.2 Cache blocks . . . 80
4.4 Complete processing systems . . . 81
4.4.1 ρ-VEX bus . . . 82
4.4.2 Standalone processing system . . . 83
4.4.3 GRLIB processing system . . . 84
4.5 Debug UART . . . 85
4.5.1 Protocol . . . 85
4.5.2 Structure . . . 87
4.6 Debug software . . . 89
4.6.1 ρ-VEX server: `rvsrv` . . . 89
4.6.2 ρ-VEX debugger: `rvd` . . . 89
4.6.3 ρ-VEX trace decoder: `rvtrace` . . . 90
4.7 Conclusion . . . 91

5 Verification **93**
5.1 Conformance test suite . . . 93
5.2 Cache simulation . . . 96
5.3 Synthesis . . . 97
5.4 Benchmarks . . . 100
5.5 Runtime reconfiguration and interrupts . . . 102
5.6 Debug link and software . . . 105
5.7 Tracing . . . 105
5.8 Conclusion . . . 106

6 Conclusion **107**
6.1 Summary . . . 107
6.2 Main contributions . . . 109
6.3 Additional contributions . . . 111
6.4 Future work . . . 112
6.4.1 Implementation . . . 112
6.4.2 Research opportunities . . . 114

Bibliography **120**

Glossary **121**

A Paper: context switching **A-1**

B Paper: sparse instruction encoding **B-1**

C ρ-VEX user manual **C-1**

List of Figures

2.1 The FPGA development boards that are used in this project. 8

2.2 Example timing diagram for a processor without a pipeline. 11

2.3 Timing diagram for the same situation as depicted in Figure 2.2, but executed with pipelining. 11

2.4 Example pipeline diagram for a branch instruction in a pipelined processor. The branch target and condition for instruction 2 become known at the end of cycle 3. 12

2.5 Example pipeline diagram for a processor with pipelining and interlocking. The EX0 stage of instruction 2 depends on the result of instruction 1. 13

2.6 Example pipeline diagram for a processor with an exposed pipeline. The depicted program is the same as in Figure 2.5. Instruction 3 is independent of instructions 1 and 2, allowing the compiler to insert it between these two instructions. 13

2.7 Example pipeline diagram for a processor with interlocking and forwarding. The EX0 stage of instruction 2 depends on the result of instruction 1 EX1. 14

2.8 Example pipeline diagram for a processor pipelined processor with a trap occurring in the EX0 stage of instruction 2. Instruction T1 represents the first instruction of the trap handler. 15

2.9 Pipeline diagram for a two-issue processor executing fully independent instructions. 18

2.10 Configurable modes of an eight-way reconfigurable ρ-VEX processor. Each cell represents a lane. A broken line separating two lanes implies that the lanes combine together to form a single core, whereas a solid line separates the cores. There are no modes in which a core only uses a single lane because long immediates require at least two lanes to be present. 22

2.11 Diagram illustrating the read data path of a non-associative cache. . . . 25

3.1 Minimal connectivity needed between contexts and lane groups to make all the configurations depicted in Figure 2.10 possible. The connections on the left show the multiplexing between context registers to the lane groups. The connections on the right show the number of lane groups that a context can be connected to. 31

3.2 Full connectivity between contexts and lane groups, in contrast to the minimal connectivity shown in Figure 3.1. This is the connectivity implemented in this work. 32

3.3 Pipeline diagram of an interrupt that is handled by simply setting the PC, as is done in the old processor, seemingly occurring after interrupts are disabled. The control register that disables the interrupts is updated between EX1 and WB. 37

3.4 Pipeline diagram illustrating unnecessary interrupt delay after re-enabling interrupts when an interrupt is handled by simply setting the PC, as is done in the old processor. The interrupt enable control register is updated between EX1 and WB. 37

3.5 Pipeline diagram illustrating approximately how the new processor handles an interrupt that was pending while interrupts were disabled. The interrupt enable control register is updated between EX1 and WB. In reality, the latency is slightly longer, as we will see in the implementation. 38

3.6 Pipeline diagram illustrating long immediate prefetch in two-way mode after returning from a trap to a mid-generic-bundle address. Note that the trap point points to pair 2. The prefetch of pair 1 is needed because pair 1 may contain a long immediate for pair 2, and must therefore be in the pipeline, even though it must not be executed. Note that the depicted latency from trap to handler is shorter than it is in the implementation. 39

3.7 Two approaches to modifying the instruction fetch system to handle misaligned bundles for a four-way ρ-VEX, or an eight-way ρ-VEX where a bundle must be aligned by syllable pairs. The unit for the program counter is a syllable or a syllable pair respectively. The datapath is highlighted for PC = 0x15. The area within the dashed rectangle is implemented in the memory or cache; the area outside of it is implemented within the processor. 43

3.8 Syllable distribution in the new ρ-VEX processor with variable-length bundle support, shown for two example bundles. 45

3.9 Diagram illustrating the inefficiency of using independent conventional instruction caches for each lane group of the ρ-VEX. The cache lines that are used in 1x8 configuration are highlighted. 46

3.10 Diagram illustrating the organization of the reconfigurable instruction cache. Instead of showing the generalized reconfigurable logic, the effective organization for two example configurations is shown. 47

3.11 Diagram illustrating the organization of the reconfigurable data cache. Instead of showing the generalized reconfigurable logic, the effective organization for two example configurations is shown. 48

3.12 Diagram illustrating which lanes can forward long immediates to which lanes in the two supported modes. Both modes can be supported simultaneously. 51

4.1 Structural overview of the ρ-VEX implemented processor. Each block represents a VHDL entity. Blocks with thick borders are instantiated multiple times, blocks with dotted borders may or may not be instantiated based on the configuration of the parent block. 56

4.2 Configuration word decoding logic for four lane groups. 60

4.3 Context-lane interface structure for an eight-way, four-context, four lane-group ρ-VEX. The lanes are represented on the left, the contexts on the right. 61

4.4 Overview of the ρ-VEX datapath and default pipeline configuration. . . 63

4.5 Instruction buffer logic for a four-way ρ-VEX with two lane groups and no bundle alignment requirements. 65
4.6 Variable-length instruction syllable rerouting logic for a single lane group that has two lanes. 67
4.7 Routing logic needed to support 'limmh-from-previous' forwarding for an eight-way ρ-VEX with four lane groups. The dashed path is only needed for generic bundles larger than eight syllables. 68
4.8 Adder structures for computing PC+1 with variable-length instructions that allow the addition to be done before the bundle length is known. The shaded areas are instantiated for all possible bundle lengths. 69
4.9 Diagram illustrating a memory with two read ports and two write ports constructed from dual port block RAM (BRAM) memories and a live value table (LVT). 74
4.10 Pipeline diagrams showing how traps are handled. The number in italics before the IF stage corresponds to the branch unit condition that selected the PC for the subsequent fetch. 75
4.11 Cache block interconnect structure for both the instruction and data caches. Depicted for an ρ-VEX with four lane groups. 79
4.12 Block diagram of a cache block. The bus snooping and flushing datapath is drawn separately from the rest of the logic for clarity. 80
4.13 Cache block timing behavior, showing when certain signals are valid relative to the cycle in which the request is made. 81
4.14 Diagram indicating how an ρ-VEX bus arbiter and demultiplexer can be used to construct a conventional multi-master bus. 82
4.15 ρ-VEX bus timing example. 83
4.16 Block diagram of the standalone processing. 84
4.17 Block diagram of the GRLIB processing system. 85
4.18 A UART frame. 85
4.19 Example debug link protocol transmission. 86
4.20 Structural overview of the debug UART. 87

5.1 Screenshot of ModelSim after running the conformance test suite in GUI mode. 94
5.2 Register and LUT resource distribution of the ρ-VEX processor. The left bars represent the results for the ML605, the right bars represent the VC707. 99
5.3 FPGA resource of the reconfigurable cache. The left bars represent the results for the ML605, the right bars represent the VC707. 99
5.4 Performance results of the reconfiguration/interrupt test program. 104

List of Tables

2.1 Comparison of the FPGA logic resources on the ML605 and VC707 FPGA development boards. [1] [2] 8

4.1 List of debug link protocol command codes. 87

5.1 List of test cases in the conformance test suite. 96
5.2 List of synthesized designs. 97
5.3 Synthesis results. 98
5.4 Benchmark results for the various configurations. The results of the individual benchmarks are combined. The designs marked with an * use code compiled with the bundle size fixed to eight syllables. 101
5.5 Individual benchmark results for the `gr24` design in 8-way mode. 102
5.6 Configurations with which the reconfiguration/interrupt test program is run. 103
5.7 Performance of the trace unit using the UART debug link. 105

List of Acronyms

ALU Arithmetic Logic Unit. 21, 23, 56, 72, 73, 95

ASIC Application-Specific Integrated Circuit. 2, 3, 5, 32, 115

BRAM Block RAM. 8, 73, 74, 78, 81, 83, 98, 100, *Glossary:* block ram

CLB Configurable Logic Block. 6, 8, *Glossary:* configurable logic block

CRC Cyclic Redundancy Check. 86, 88, 110

DE Decode. 21

DLP Data Level Parallelism. 17

DSP Digital Signal Processor. 1

EX Execute. 10, 21

FIFO First-In-First-Out. 88, 89

FPGA Field-Programmable Gate Array. 2–8, 21, 28, 32, 41, 44, 48, 50, 55, 66, 74, 81, 84, 89, 91, 97, 105, 107, 109, 112, 115, *Glossary:* field-programmable gate array

FSM Finite-State Machine. 59, 88

GPIO General-Purpose Input/Output. 28

GUI Graphical User Interface. 94

HDL Hardware Description Language. 6, 7, 55, 107

HLS High-Level Synthesis. 55

IC Integrated Circuit. 5

`IEN` Interrupt Enable. 37, 38, 40

IF Instruction Fetch. 10, 11, 21

ILP Instruction Level Parallelism. 2, 17–21, 32, 102, 115

ISA Instruction Set Architecture. 19, 52

LRU Least Recently Used. 25, 46, 114

LSB Least Significant Bit. 67, 70, 87

LUT Lookup Table. 6, 8, 66, 74, 98, 100

LVT Live Value Table. 73, 74

MMU Memory Management Unit. 2, 17, 27, 113

NOP No-operation. 3, 9, 12, 13, 19, 37, 38, 70, 77, 101, 102

OS Operating System. 16, 40, *Glossary:* operating system

PAR Place-And-Route. 7

PC Program Counter. 9–11, 14, 15, 17–19, 21, 22, 29, 31, 37–39, 41, 44, 57, 62, 64–66, 68–72, 74, 76, 77, 110, *Glossary:* program counter

RFT Ready For Trap. 36, 37, 40

RISC Reduced Instruction Set Computer. 9

RLP Request Level Parallelism. 17

TLB Translation Lookaside Buffer. 27

TLP Thread Level Parallelism. 2, 17, 18, 21, 115

UART Universal Asynchronous Receiver/Transmitter. 28, 41, 42, 53, 85, 86, 88, 91, 108–110, *Glossary:* universal asynchronous receiver/transmitter

USB Universal Serial Bus. 41, 89

UUT Unit Under Test. 7

VHDL Very High Speed Integrated Circuit Hardware Description Language. 6, 21, 52, 55, 91, 93, 96, 107, 108, *Glossary:* VHDL

VLIW Very Large Instruction Word. 2, 19, 20, 24, 114

WB Writeback. 10, 13, 21

Acknowledgements

I would like to thank...

Stephan,

for being a kind and flexible advisor and, if everything works out, boss

Stephan, Joost, Anthony, Wilco, Maurits, and my dad,

for proofreading various versions of this thesis

Joost, Anthony, Jens, Hugo, and Klaas,

for being great colleagues to work alongside

My parents,

for their support (and coercion)

My parents and grandparents,

for having saved up to fund my BSc. and MSc., so I never had to worry about a student loan

J. van Straten
Delft, The Netherlands
May 11, 2016

Introduction 1

This thesis documents a complete redesign of the ρ-VEX (reconfigurable VEX) implementation created by [3] to support dynamic (runtime) reconfiguration, precise traps, debugging, and variable-length bundles. In addition, a compatible reconfigurable cache is designed and implemented. This thesis is intended for the scientific reader, who is expected to be interested primarily in the new components introduced in this version of the ρ-VEX processor, which design choices were made in their development, and how they affect the performance and usability of the system.

If you came to this thesis to learn how to use the created ρ-VEX design, refer to the ρ-VEX user manual instead. The manual is intended to be kept in sync with the ρ-VEX as it is developed further, something that is impossible with a thesis. The appropriate version of the ρ-VEX user manual should come with its source distribution. In addition, the manual as it was at the time of writing this thesis is included as Appendix C.

The remainder of this chapter serves to further introduce the ρ-VEX project and this thesis. The first section intends to give context to the ρ-VEX project as a whole. The second section describes the current status of the ρ-VEX project itself, and the shortcomings of the current processor implementation. The third section specifies the problem statement and boundary conditions, as well as the approach taken to solve the stated problem. Finally, the last section provides an overview of structure of this thesis.

1.1 Context

In this age of battery-powered devices and sustainability, the market demands ever smarter and more energy-efficient systems. One way to achieve that is to use smaller transistors, but as CMOS gates approach thicknesses of just one or two atoms, this approach is becoming ever more difficult.

Parallelization is a different approach. Put simply, a processor capable of doing more than one addition in one instruction can be more energy efficient than a processor that can only do one addition at a time, simply because instruction decoding and other control logic do not require duplication. Thus, per useful addition, the parallel processor uses less energy.

Unfortunately, the previous example only holds if the application running on the processor can make use of the multiple additions. If it cannot, the idle power consumption of the additional, unused functional units is wasted, and detracts from the efficiency. Therefore, these types of processors are typically engineered for a specific purpose, such as signal processing. In that specific case, the processor is referred to as a digital signal processor (DSP). They are usually part of a larger system, where a conventional processor performs the tasks that do not map well to the otherwise faster and more energy efficient DSP.

The ρ-VEX reconfigurable very large instruction word (VLIW) architecture, developed by the Computer Engineering group of Delft University of Technology, aims to perform well for both highly parallel algorithms and general purpose computing. This is accomplished by its ability of functioning as a single, highly parallel processor, or distributing its many functional units among several threads, similar to a multi-core conventional processor. In other words, the ρ-VEX can adapt to instruction level parallelism (ILP) or thread level parallelism (TLP).

Note that by 'reconfiguration' we do not mean loading a different bitstream into a field-programmable gate array (FPGA). All reconfigurable features are described as such in the hardware description. Plans exist to construct an application-specific integrated circuit (ASIC) design as part of future work.

1.2 Current state of the ρ-VEX project

Two softcore processors that partially implement the ρ-VEX concept already exist. [4] is only design-time configurable, whereas [3] only allows reconfiguration between programs, as the processor state is corrupted by the reconfiguration process. A processor that fully implements the ρ-VEX concept is desired.

Furthermore, in order to advance the research into the ρ-VEX architecture efficiently, a flexible hardware design is needed to facilitate design space exploration for a specific application. Simultaneously, ever more complex features are needed to provide a platform capable of running industry-standard benchmark suites, in order to allow a fair comparison to be made with current commercial processing solutions.

In particular, one of the long-term goals of the ρ-VEX project is to port a recent version of the Linux kernel to the ρ-VEX architecture. An effort to this end has already been made in 2014 [5]. However, as the processor lacked a memory management unit (MMU), this Linux port also lacked many features. Furthermore, the trap controller designed for the ρ-VEX in that work is not 'precise', meaning that a trap does not necessarily interrupt the instruction stream exactly when the trap occurs. Specifically, a trap does not cause a pipeline flush, causing instructions after the trapped instruction that have already been issued to still complete. This makes an MMU an impossibility, as a page fault trap has to be precise. Furthermore, the debug system (also implemented in [5]) can only interrupt the processor by merely disabling the clock signal, as opposed to doing a pipeline flush first. This means that several subsequent instructions are interrupted at various pipeline stages, making it difficult to determine the state of the program.

Another problem with the current ρ-VEX is the lack of a cache tailored specifically to its reconfigurable nature. Before this work, the level one cache from the CARPE project [6] had been ported to interface with the ρ-VEX processor. This cache has two major limitations. First and most obviously, it is not designed for a runtime-reconfigurable processor. This requires the data cache and parts of the instruction cache to be duplicated for each thread that the ρ-VEX can run at once, with the duplicated resources going to waste when fewer threads are running with more computational resources. There is also no way to flush the cache, and the cache is not coherent. This makes it impossible to have communicating threads.

Finally, there is the current instruction fetch unit, which only supports fixed-length instructions. ρ-VEX instructions can be up to 32 bytes in size, encoding up to eight parallel operations (syllables), depending on the design-time configuration. However, programs usually do not have enough parallelism to be able to use all these operations all the time, resulting in no-operation (NOP) syllables having to be inserted to get this uniform instruction width. This waste of cache, memory, and bus resources leads to significant performance degradation, making it difficult to compare the ρ-VEX with commercial processors.

1.3 Problem statement and methodology

In this work, we will address these shortcomings by doing a full redesign of the ρ-VEX processor. This gives rise to the following research question.

> *How to design and implement a dynamically reconfigurable and parameterizable VLIW processor?*

In order to address the problems with the current ρ-VEX processor, the following requirements were identified for the new implementation.

1. The design must be compatible with the current ρ-VEX compiler toolchain.
2. The design must be dynamically (runtime) reconfigurable.
3. The design must support precise traps.
4. The design must support debugging.
5. The design must support variable-length instructions.
6. The design must include a coherent, dynamically reconfigurable cache.

It should be noted that optimization for area, speed or energy are beyond the scope of this project. Instead, the focus is on design flexibility, so the ρ-VEX can be efficiently used in future research projects.

The following tasks will be performed to meet these requirements and answer the research question.

1. Investigate the current ρ-VEX design to determine which parts can be reused for the new design.
2. Identify which parts of the processor are affected by the identified requirements, in particular dynamic reconfiguration, and design them accordingly.
3. Implement the processor on an FPGA, without using FPGA-specific reconfiguration techniques, to maintain ASIC toolchain compatibility.
4. Verify the functionality of the processor using conformance tests in simulation and benchmarks in hardware.

1.4 How to read this thesis

Let us provide an overview of the contents of the remaining parts of this thesis and their intended audiences.

Chapter 2 provides background information. Section 2.1 discusses FPGA theory and the features of the ML605 and VC707 FPGA development boards. Section 2.2 details architecture-agnostic processor design theory, with the exception of Section 2.2.6, which describes the ρ-VEX architecture specifically, and lists the components of the current design that will be reused in the processor described in this work. Finally, Section 2.3 describes memory hierarchy, focussing on caches in particular. If the reader is already familiar with these generic topics, everything but Section 2.2.6 of Chapter 2 can be skipped. Note that an extensive glossary is provided starting on page 121, which may be referenced instead of Chapter 2 if a word is not clear. If you are reading this thesis digitally, words used in the text that occur in the glossary are typically hyperlinked to the glossary entry.

Chapter 3 lists and defends the major design choices made during the design of the processor. It can be regarded as the core of this thesis.

Chapter 4 details the implementation of the various components of the implemented processor (Section 4.2), cache (Section 4.3), and supporting components (Sections 4.4 through 4.6). While an effort was made to make it possible to read the chapter linearly, it is intended more so as reference material for researchers who wish to understand or make modifications to the source code, or reimplement parts of it. Of particular importance in these cases are Figures 4.1 and Figure 4.4, which present overviews of the structure and datapath of the processor respectively, as well as Figure 4.11, which presents the structure of the cache.

Chapter 5 describes the method taken to verify the functionality and evaluate the performance of the implemented system. The results of these experiments are also listed and discussed.

Chapter 6 summarizes the work and lists recommendations for future work.

Appendices A and B contain copies of two papers that were co-authored by the author of this thesis. The first paper, *'Multiple Contexts in a Multi-ported VLIW Register File Implementation'* [7], discusses the implementation of the general-purpose register file in detail. The second paper, *'A Sparse VLIW Instruction Encoding Scheme Compatible with Generic Binaries'* [8], discusses the approach taken to implementing support for variable-length instructions.

Appendix C contains a copy of the ρ-VEX user manual at the time of writing. It is intended primarily for researchers who will be using the implemented processor for their work. The manual is included as an appendix as it is referenced on various occasions, primarily in Chapter 4, while being unpublished at the time of writing. In consideration of the environment, it may not be included if you have received this thesis in print. Note that the page numbers of the user manual are prefixed with the appendix letter to avoid confusion. In particular, note that the bibliography at the end of this document belongs to the user manual; the bibliography of this thesis starts on page 117.

Background 2

In this chapter, we present the theory required to design and implement a processing system. Examples of contemporary existing processor architectures are given to complement the theory. We start with background information about FPGAs, an essential tool in the verification process of any hardware design. The second section discusses processor design. The third section deals with the supporting components of a processing system, in particular the memory subsystem and caches.

2.1 FPGAs

Field-programmable gate arrays (FPGAs) are integrated circuits (ICs) that allow their functionality to be programmed into them after production, i.e., 'in the field'. This is similar to processors, but instead of loading software, a hardware specification of a digital circuit is loaded. These hardware specifications are similar to those used to specify application-specific integrated circuits (ASICs), allowing ASIC designers to test their design before taking it into production.

However, field-programmable gate arrays (FPGAs) are more than just an ASIC design tool. As FPGAs can be mass produced, they can be cheaper than ASICs for low volume products. In addition, an FPGA can be updated after a problem is found with the design, similar to a firmware update; something that is not possible with an ASIC. This also makes FPGAs very suitable for hardware development and research in general, regardless of whether an ASIC is the goal. The main drawback of an FPGA compared to an ASIC is that they are relatively slow and power hungry; after all, they need to include all the logic needed for any hardware design. However, as speed and power consumptions are not relevant in a prototype design, an FPGA shall be used in this project to develop the new components for the ρ-VEX processor.

2.1.1 Theory of operation

In order to understand how an IC can be designed such that it can support any digital circuit, consider that any digital circuit can be constructed from just NAND or NOR gates [9, pp. 47-49]. In other words, a sufficiently large array of either of these gates with configurable connections between them can form any logic circuit. To visualize how such a configurable interconnect is possible, consider the most extreme case, where any gate input can be connected to any internal gate output and any external input pin. Notice that this requirement simply specifies a multiplexer, of which the select input is programmable. The same thing can be achieved for the outputs of the circuit.

While the preceding satisfies the requirements for an FPGA, it is not practical to implement them in this way. In practice, the interconnect does not allow every input to be connected to any output directly, as this would require excessive routing resources. Also, the basic logic gates are replaced with more complex configurable logic blocks (CLBs).

A CLB typically contains a number of lookup tables (LUTs), registers and carry logic. A LUT is simply a one-bit wide asynchronous memory that can be loaded when the design is programmed into the FPGA, with its read address inputs exposed to the programmable interconnect. This allows any n-input logic gate to be implemented using an n-input LUT, i.e., a LUT with 2^n bits of memory. The registers in a CLB allow sequential logic to be implemented much more efficiently than if LUTs were to be used. Similarly, the carry logic allows adders to be implemented more efficiently.

In addition to CLBs, FPGAs typically have blocks that are hardwired to fulfill specific purposes as well, such as memories or multipliers. Using these blocks when possible is significantly more efficient than using CLBs.

2.1.2 Softcore processors

Sometimes it makes sense to instantiate an entire processor on an FPGA. For example, high level control functions that do not need hardware acceleration but still need to be tightly coupled with the hardware may need to be performed. Two examples of softcore processors are the LEON3 and the MicroBlaze.

The LEON3 is an open source softcore by Aeroflex Gaisler implementing the SPARC V8 instruction set architecture. It is part of the GRLIB IP library [10], a collection of hardware description language (HDL) components. This library is used within this project to provide the ρ-VEX processor with a set of peripherals and a memory controller. In contrast to the LEON3, the MicroBlaze is a closed source processor. It is available as part of the synthesis toolchain for Xilinx FPGAs. Both processors have a 32-bit RISC architecture.

2.1.3 Hardware description languages

A hardware design is normally specified using an HDL. Examples of HDLs are VHDL, Verilog and SystemC. Each of these, in their own way, allow hardware to be described using processes, signals, and entities[1].

A process is a piece of sequential code that operates on a set of input signals in order to drive a set of output signals. When the hardware is generated from a process, loops are unrolled and conditional assignments are transformed into multiplexers, such that the generated hardware generates the same values for a given set of input values as the sequential code would.

A signal represents a physical wire. Unlike a variable, a signal cannot be assigned a value to immediately, similar to how the voltage on a wire cannot change instantaneously. Instead, when a value is assigned to a signal, it will only assume the new value some time in the future.

[1] Wherever there are differences in terminology between these languages, the VHDL terms shall be used.

An entity fulfills a similar role in hardware as a class or function does in software, in that it allows a system to be specified hierarchically. Each entity represents a block of logic that consists of a number of processes and/or other entities. When an entity is used, it is said to be instantiated. Entities have an interface consisting of zero or more input, output or tri-state signals and zero or more generics. A generic is a constant value specified when the entity is instantiated, serving as a configuration parameter.

In an HDL design, one entity must be specified to represent the system as a whole. This is called the toplevel entity. The inputs and outputs of this entity represent the physical inputs and outputs of the FPGA design.

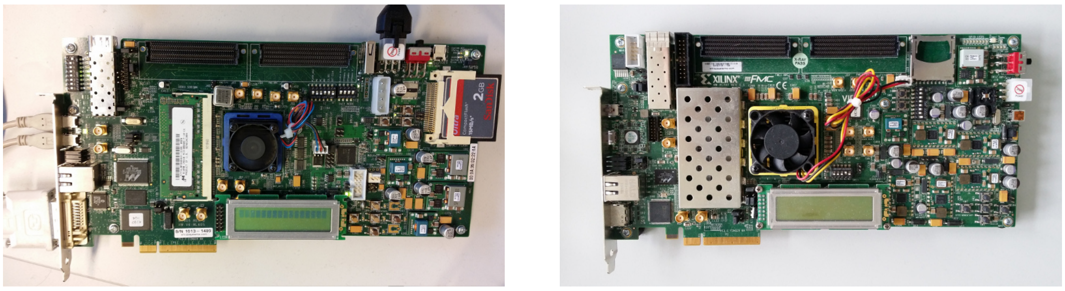
The programming file for an FPGA that determines the functionality is called a bitstream. The process of generating a bitstream from HDL code consists of several steps. The first step is called static elaboration. During this step, constant propagation is performed. The second step is synthesis. Here, each process is converted into a logic block. These logic blocks are then interconnected using signals in order to form a circuit description. The next step is mapping, during which the elements in the circuit description are mapped to the resources available within the target FPGA. Finally, there is placement and routing (PAR). During this step, the mapped components and signals are laid out onto the FPGA fabric. Aside from the HDL code, this step also takes a set of constraints as an input. Most important of these are timing constraints, which specify, among other things, the maximum delay for combinatorial signals from register to register, thereby specifying the minimum clock frequency at which the design must operate. The synthesis tools will attempt to meet these requirements, and iteratively change the placement and/or routing until the constraints are met or they are found to be impossible.

Debugging a hardware description is done by means of simulation, as one cannot for example simply pause execution of a piece of hardware to see what line of code is being executed when something goes wrong. An HDL simulation consists of the design files for the unit under test (UUT) and a testbench. The testbench generates the UUT input signals that cause the UUT to perform some operation that is to be tested. ModelSim by Mentor Graphics is an example of a software package that allows HDL designs to be simulated.

There are different levels of simulation accuracy. Simply simulating the HDL code directly is called behavioral simulation and is the least accurate, but simulates the fastest. The other levels are post-synthesis and post-PAR simulation. For post-synthesis simulation, the synthesized circuit is converted back to HDL and simulated. This allows the designer to ensure that the synthesizer is synthesizing the sequential process descriptions as expected, at the cost of some simulation performance. Post-PAR simulation is similar, but here, an HDL model of the actual FPGA fabric is simulated, including its timing behavior. This is the most accurate simulation model, but it is also the slowest. Furthermore, in post-synthesis as well as post-PAR simulation, it is often difficult to determine what is actually going on, due to the hardware optimizations performed by the synthesis tools.

2.1.4 ML605 and VC707 development boards

Two FPGA development boards are used in this project to verify the hardware. These are the ML605 [11] and VC707 [12] development boards, produced by Xilinx, depicted in Figure 2.1. These boards have a Virtex-6 XC6VLX240T-1FFG1156 and a Virtex-7 XC7VX485T-2FFG1761C FPGA respectively. These boards have various peripherals outside of the FPGA that may be used in the design. In this project, only the DDR3 memory (512 MiB on the ML605, 1 GiB on the VC707) and the USB to serial convertor (available on both boards) are used.



(a) ML605

(b) VC707

Figure 2.1: The FPGA development boards that are used in this project.

Both FPGAs have similar CLBs [13] [14]. Each CLB contains two so-called slices, each containing four 6-input LUTs, eight registers and a 4-bit long carry chain, that can be cascaded with neighboring slices. Some of the slices also allow the LUT configuration memory to be used as a variable depth shift register of up to 64 bits deep, or as a 64-bit RAM with independent read and write ports. The latter is called distributed RAM.

In addition to CLBs, both FPGAs contain block rams (BRAMs). Each BRAM is 36 kib in size, and has two fully independent synchronous read/write ports. These blocks are used for bulk on-chip memory storage. Furthermore, they contain DSP slices, which, among other things, contain a 25x18 bit multiplier each.

The main difference between the Virtex-6 and Virtex-7 series is the speed and density. More specifically, the Virtex-7 series is marketed to have a 2x improvement in system performance compared to the the Virtex-6 series [2]. Table 2.1 summarizes the logic resources of the FPGAs on the ML605 and VC707 boards.

Table 2.1: Comparison of the FPGA logic resources on the ML605 and VC707 FPGA development boards. [1] [2]

Board	Family	Slices	BRAMs	DSP slices
ML605	Virtex-6	37,680	416	768
VC707	Virtex-7	75,900	1,030	2,800

Xilinx maintains its own synthesis toolchains for its products. The toolchain that will be used in this project is Xilinx ISE 14.7. It has recently been superseded by Vivado, but Vivado does not support sub-7-series FPGAs [15].

2.2 Processor architecture

To review the processor architecture concepts needed to understand this thesis, let us define a basic processor. A processor is a device that takes instructions from some instruction memory and executes them one by one. The address that the current instruction is loaded from is called the program counter (PC). The PC is stored in a register that is automatically incremented after the current instruction is completed, in order to load the subsequent instruction.

Most instructions perform operations on data. The ways in which data can be stored vary from processor to processor, but typically, it is stored in the data memory or in a register file. The data memory is large and slow, while the register file can always be accessed directly but is small. The register file is used for values that are currently being operated on, while the data memory stores all other data.

In a reduced instruction set computer (RISC), instructions that perform some kind of operation on data, such as arithmetic instructions, can usually only operate on the register file. In order to access the data memory, load and store instructions need to be used, which can only copy data from the data memory to the register file or vice versa. This is called a load-store architecture.

Most processors also include an instruction that does nothing, called a no-operation (NOP) instruction. NOP instructions can be used for padding or delaying execution for a short amount of time.

Another type of instruction is a branch instruction. A branch instruction does not modify any data, but instead modifies the next PC. This allows constructs such as loops, conditional statements and function calls to be encoded in the program.

Branch instructions can be conditional or unconditional. A conditional branch only affects the next PC if some data-dependent condition is true, whereas an unconditional branch, also called a jump instruction, always sets the next PC. If the condition of a branch instruction is true, the branch is said to be taken. If the condition is false, the branch is not taken. The instruction that is to be executed after a conditional branch is taken, or after an unconditional branch is executed, is called the branch target.

A special kind of jump instruction called a call instruction also stores what the next PC would have been in what is known as the link register. This is used for function calls; the link register contains the return address of the function call. A return instruction is placed at the end of every function, which is a jump instruction that sets the next PC to the contents of the link register.

So far, we have referred to the data and instruction memories as being separate. While this is possible and may be preferable depending on the application, there is no fundamental requirement for this distinction. In particular, when a large amount of memory is needed, it makes sense to use a single large external memory for instructions as well as data.

2.2.1 The stack

If a function needs to call another function, the contents of the link register must first be saved somewhere. Otherwise, the return address is lost, as any call will modify the contents of the link register. Subsequently, before the return instruction, the saved value must be restored. In fact, any data local to a function may need to be saved and restored, in particular if a function can call itself. The data structure used to accomplish this is called the stack [16, pp. 114-116].

A stack is a last-in first-out buffer. That is, when the values A and B are saved to ('pushed onto') the stack, the first value that is read ('popped') from the stack is B, and the next value is A. This is exactly what we want: it does not matter how many values the called functions push and pop to and from the stack, as long as the functions push and pop an equal amount of values. If a function pushes a different number of values as it pops, or otherwise affects the stack space of other functions, stack corruption occurs.

A special register called the stack pointer points to the top of the stack. Typically, the stack pointer is initialized to the end of the data memory allocated for an application and is decremented as the stack grows in size. That is, the stack pointer is decremented before a value is pushed to allocate space, and incremented after a value is popped to free space [16, pp. 116].

In addition to the stack pointer register, a frame pointer register is sometimes also needed. The frame pointer is set to the stack pointer at the start of a function, and when the function returns, the stack pointer is set to the frame pointer. This allows the function to allocate an arbitrary amount of data for local variables without needing to remember how much it actually allocated for when the stack pointer needs to be restored.

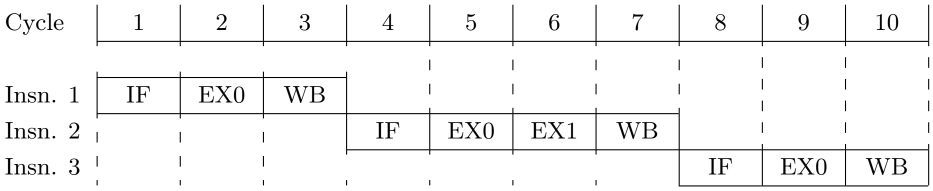
2.2.2 Timing and pipelining

So far, the timing of how instructions are processed has not been discussed. Starting at the basics again, the naive solution is to specify that each instruction executes completely in exactly one clock cycle. The next PC can then be fed directly to the input of the PC register.

While this solution is valid, it is not very efficient, as the clock period has to be long enough to allow the most complex instruction to execute. This problem can be solved by making the number of clock cycles per instruction dependent on the instruction. That way, the clock period is no longer bound by the slowest instruction, but can be chosen based on the delay of the most common instructions only, while the more complex instructions are given multiple cycles to complete.

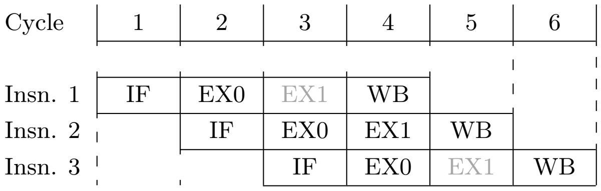
There is more performance to be gained. Let us divide the processing of an instruction into three stages, instruction fetch (IF), execute (EX) and writeback (WB). In IF, the instruction memory at the current PC is read, in EX, the actual processing of the instruction is performed, and in WB, the results of the instruction are written back to the register file. Let us now specify that the IF and WB stages are executed in their own cycle, and that EX may take one or two cycles, depending on the complexity of the instruction. Figure 2.2 depicts what the timing will look like for three instructions, of which the second instruction spends two cycles in the EX stage.

Let us now adjust the timing such that a new instruction is started ('issued') every cycle, instead of waiting for the previous instruction to complete. In addition, let instructions that only need one EX cycle use the worst case amount of cycles. The processor is now considered to be pipelined. The new timing is depicted in Figure 2.3.



Cycle	1	2	3	4	5	6	7	8	9	10
Insn. 1	IF	EX0	WB							
Insn. 2				IF	EX0	EX1	WB			
Insn. 3								IF	EX0	WB

Figure 2.2: Example timing diagram for a processor without a pipeline.



Cycle	1	2	3	4	5	6
Insn. 1	IF	EX0	EX1	WB		
Insn. 2		IF	EX0	EX1	WB	
Insn. 3			IF	EX0	EX1	WB

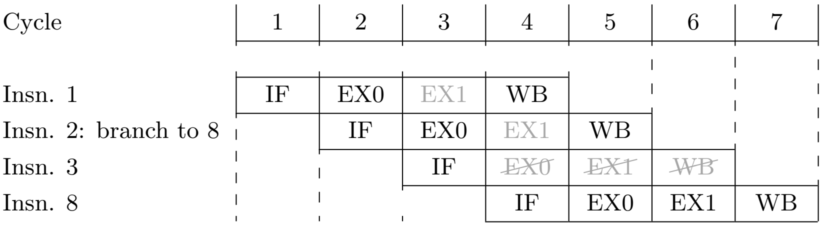
Figure 2.3: Timing diagram for the same situation as depicted in Figure 2.2, but executed with pipelining.

A different way to obtain this result is to start with a processor that executes every instruction in exactly one cycle, and then simply place registers between every execution stage. This method illustrates that pipelining a processor does not in principle require additional datapath hardware aside from registers. There are, however, problems that come with pipelining that need to be dealt with in some way. This may require additional control logic.

Computing the next PC One of these problems is determining the next PC. So far, we have only considered normal instruction flow without branches. That means that the computation of the next PC simply consists of adding the size of the current instruction to the current PC. We will refer to this value as PC+1. Notice that if the length of instructions is not fixed and depends on the instruction itself, this seemingly simple computation already depends on the fetched and decoded instruction. It gets even worse for conditional branch instructions and jump instructions with a non-constant branch target, as the computation now also depends on data and possibly the execution result of an instruction.

For the remainder of this thesis, the assumption shall be made that fetching an instruction and determining its length can be done in a single cycle. If this assumption is violated, the instruction fetch logic and instruction encoding becomes vastly more complicated, as the next PC must then be predicted somehow, and mispredictions need to be handled in some way. Without such logic, the processor would only be able to fetch and issue a new instruction every n^{th} cycle, in which case simply decreasing the clock frequency is a more energy efficient solution.

Thus, only the problem of dealing with branches remains. For our example processor, consider the case where the branch target and condition are determined at the end of the EX0 stage. Now, let the next PC be the branch target from EX0 if the branch instruction in EX0 is taken, and the PC+1 value computed in the IF stage otherwise. Figure 2.4 shows the timing of an example program with a branch instruction that follows these rules. In the example, the desired program execution is instruction 1, followed by instruction 2, which then branches to instruction 8. Observe that instruction 3 is issued regardless of the branch, as the branch target and condition are not yet known when it is issued. In order to still execute the code in the intended order, instruction 3 needs to be disabled. The additional logic required is called pipeline flushing logic. In this particular example, the pipeline is said to be flushed until EX0 in stage three, meaning that the instructions in stages before EX0 are disabled.



Cycle	1	2	3	4	5	6	7
Insn. 1	IF	EX0	EX1	WB			
Insn. 2: branch to 8		IF	EX0	EX1	WB		
Insn. 3			IF	~~EX0~~	~~EX1~~	~~WB~~	
Insn. 8				IF	EX0	EX1	WB

Figure 2.4: Example pipeline diagram for a branch instruction in a pipelined processor. The branch target and condition for instruction 2 become known at the end of cycle 3.

Alternatively, the instruction after a branch instruction can be defined to always be processed, regardless of whether or not the branch is taken. Such an instruction is called a branch delay slot. MIPS [17] is an example of an architecture that does this. Doing this saves some logic and ensures that the energy spent on fetching the instruction after the branch does not necessarily go to waste. However, the success of implementing a branch delay slot depends on whether the compiler is able to find a useful instruction to place in the branch delay slot. If it cannot and needs to place a NOP instruction in the majority of the cases, the increase in program size may not be worth it.

It should be noted that there are also architectures in which the branch target is known in an earlier pipeline stage than the branch condition. In this case, branch prediction may be employed [16, pp. 341]. With branch prediction, the processor jumps to the branch target or continues issuing instructions normally based on some prediction algorithm that can be computed faster than the actual branch condition. It is also possible to construct such prediction logic for the branch target if this requires significant computation. Such prediction schemes are beyond the scope of this work.

Data dependencies Another difficulty with pipelining is handling data dependencies accordingly. Going back to our example instruction sequence and timing in Figure 2.3, let us assume that instruction two needs the result that instruction one writes to the register file for its computation. Instruction two attempts to read this value from the register file in stage EX0. Unfortunately this leads to an incorrect value being read, as instruction one has not yet written its result in cycle 3, even though it has already finished executing. This is called a hazard.

One way to solve this problem is to stall the pipeline until the data is available. This is called interlocking. The updated timing is shown in Figure 2.5.

A different way of accomplishing the same result as interlocking is to 'expose' the pipeline to the compiler. The compiler can then insert independent instructions between two dependent instructions in order to prevent hazards from ever occurring. If useful

Cycle	1	2	3	4	5	6	7	8
Insn. 1	IF	EX0	EX1	WB				
Insn. 2		IF	→	→	EX0	EX1	WB	
Insn. 3					IF	EX0	EX1	WB

Figure 2.5: Example pipeline diagram for a processor with pipelining and interlocking. The EX0 stage of instruction 2 depends on the result of instruction 1.

instructions are not available, the compiler must insert NOP instructions instead. The updated timing is shown in Figure 2.6.

Cycle	1	2	3	4	5	6	7
Insn. 1	IF	EX0	EX1	WB			
Insn. 3		IF	EX0	EX1	WB		
NOP			IF	EX0	EX1	WB	
Insn. 2				IF	EX0	EX1	WB

Figure 2.6: Example pipeline diagram for a processor with an exposed pipeline. The depicted program is the same as in Figure 2.5. Instruction 3 is independent of instructions 1 and 2, allowing the compiler to insert it between these two instructions.

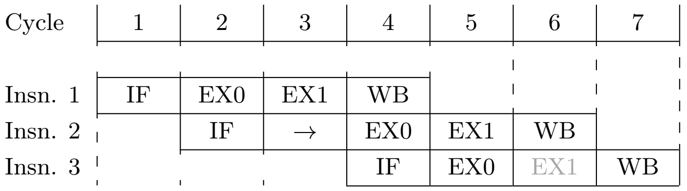
This technique has the advantage of not requiring any interlocking hardware to solve the problem. It also potentially increases performance, as the compiler may be able to move instructions in order to optimize the schedule, as can be seen in the example. On the other hand, the NOP instructions that may be added will incur a penalty in code size and instruction cache performance. Additionally, as it requires the compiler to know what the exact latencies are, programs cannot be compiled generically for processors with different latencies that are otherwise identical, without knowing the exact latencies for each processor and assuming the worst case scenario for the latencies.

Consider again the timing depicted in Figure 2.5. Here, the EX0 stage of instruction 2 is dependent on the value computed in EX0 of instruction 1. Notice that this means that the data is actually already available in cycle 3; the only reason instruction 2 must wait until cycle 5 is because the only way it can access the value is through the register file.

This delay can be avoided by making a direct path from the result of EX0 to the input of EX0 in the next cycle. Such a path is called a forwarding path. A similar path is necessary from the end of the EX1 stage to the start of EX0 in the next cycle. It is no longer needed from WB to EX0 however, as from that point onwards, the value can be read from the register file.

Now, when EX0 reads its operands, it must not only read from the register file, but also check whether the instructions that were in EX0 or EX1 in the previous cycle intend to write to the to-be-read register. If this is the case, the value from the register file is outdated, and the forwarded value is to be used instead. If both the instructions previously in EX0 and EX1 write to the same register, the value from EX0 takes precedence over the value from EX1, as the instruction in EX0 is executed after the instruction in EX1.

Note that forwarding alone is not a replacement for interlocking or having an exposed pipeline if there are multiple execution stages. To understand this, consider the same case as we have used so far, but instead of instruction 2 EX0 being dependent on instruction 1 EX0, it is dependent on instruction 1 EX1. The timing for this case is presented with forwarding and interlocking in Figure 2.7.



Cycle	1	2	3	4	5	6	7
Insn. 1	IF	EX0	EX1	WB			
Insn. 2		IF	→	EX0	EX1	WB	
Insn. 3				IF	EX0	EX1	WB

Figure 2.7: Example pipeline diagram for a processor with interlocking and forwarding. The EX0 stage of instruction 2 depends on the result of instruction 1 EX1.

2.2.3 Traps

While executing a program, unexpected situations can arise when it is either not possible or not desirable to proceed with normal execution. Such a situation can for example be a read from a memory address that does not exist, attempted execution of an undefined instruction, division by zero, or a peripheral requiring immediate attention. Such situations are called traps[2].

A distinction is made between traps that are caused by the processor itself and traps that are caused by external sources, such as peripherals. The former is referred to as a fault, whereas the latter is referred to as an interrupt. Interrupts can usually be ignored for a certain amount of time, depending on the interrupt and the application, and processors may allow the program to temporarily disable interrupts. In contrast, faults can usually not be ignored.

Traps are handled by ignoring the trapped instruction and instead branching to a trap handler. The PC of the interrupted instruction is known as the trap point. In what way the address of the trap handler is determined varies from processor to processor. In some processors the address of the trap handler depends on the kind of trap that occurred, in others the address is always the same and the trap is identified in some other way. In some processors the trap handler addresses are fixed, and in some they are configurable.

For instance, in Atmel AVR microcontrollers the interrupt handler address is the index of the interrupt multiplied by the size of an instruction (ATmega8 as an example: [18, pp. 46]). A jump instruction is expected at each of these addresses that branches to the appropriate handler. The ARM Cortex-M0 series uses a more sophisticated approach

[2] There does not appear to be an agreement in literature to the exact definitions of traps, interrupts, exceptions and faults. In this thesis, the word ‘trap’ shall be used to refer to any unexpected condition that results in a branch to a service routine. An ‘interrupt’ is a trap caused by a source external to the processor, whereas a ‘fault’ is caused inside the processor. The word ‘exception’ shall not be used.

[19, page 2-23]. It uses a data structure in memory called a vector table to store pointers for each trap (exception) handler. This table can itself be moved using an offset register. In contrast, in the STMicroelectronics ST200 series processors, there are only two trap handlers, one for normal traps and one for debug traps [20, pp. 21-27]. The address of these handlers is configurable using control registers. Furthermore, traps are identified by means of status registers.

If it is to be possible to resume execution of a program interrupted by a trap, the trap needs to be precise. According to [21], a trap is considered to be precise if and only if the following requirements are satisfied.

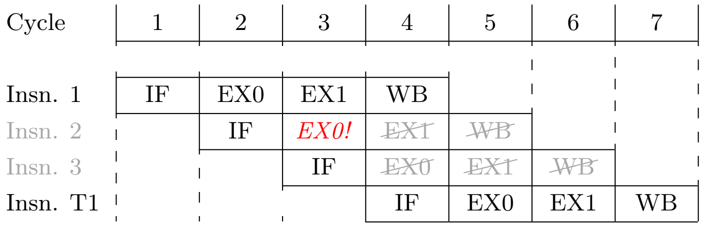
1. All instructions prior to the trap point must have executed and written their results correctly.

2. The instruction at the trap point must either have fully executed and written its results, or it must not have affected the state of the program at all, depending on the architecture and the cause of the trap.

3. All instructions after the trap point must not have affected the state of the program.

4. If the trap is caused by an instruction, the trap point must point to exactly that instruction.

Satisfying the trap preciseness requirements in a pipelined processor requires all instructions following the trap point in the pipeline to be disabled, to prevent them from affecting the state. This process will be referred to as invalidation in this work. At the same time, the next PC is forced to the address of the appropriate trap handler, and the trap point register is set to the current PC. A timing example of this is shown in Figure 2.8.



Cycle	1	2	3	4	5	6	7
Insn. 1	IF	EX0	EX1	WB			
Insn. 2		IF	*EX0!*	~~EX1~~	~~WB~~		
Insn. 3			IF	~~EX0~~	~~EX1~~	~~WB~~	
Insn. T1				IF	EX0	EX1	WB

Figure 2.8: Example pipeline diagram for a processor pipelined processor with a trap occurring in the EX0 stage of instruction 2. Instruction T1 represents the first instruction of the trap handler.

2.2.4 Debugging

Debugging is the process of finding the cause of a problem with a piece of software by inspecting the program while running it on the target processor. In order to give the programmer time to interpret the state of the program, the program is paused. Debuggers allow the program to be paused when some condition is met. Such conditions include the PC matching a certain value (called a breakpoint) or the processor accessing a piece of memory (called a watchpoint). While the program is paused, the debugger allows access to the register file and memories of the program that is being debugged. Finally, when the programmer is done, he or she can instruct the debugger to resume execution.

Debug software is normally aware of the architecture and program that is being debugged. Along with debug information generated by the compiler, the debug software allows the programmer to set breakpoints on lines of high-level code instead of on an instruction address, set watchpoints on variables instead of data addresses, etc. Finally, they can display exactly what the program was doing and why by interpreting the stack to display which functions called which in order to arrive at the current instruction.

When debugging desktop computer applications, the debug software usually runs on the same machine as the program that is being debugged (the 'target'). The debug software then connects with the target through the operating system (OS). This is called self-hosted debugging. With embedded processors, this is typically not feasible. Instead, external debugging is used [22]. In this setup, a desktop computer (the 'host') runs the debugging software, and connects to the target processor with some kind of debug link, such as JTAG or a serial port.

Some hardware support is needed for debugging. [22] specifies the basic commands that need to be supported as reading and writing the register file, reading and writing memory, returning why the program was paused, and continuing execution. In addition, in order to support stepping and breakpoints, an instruction must be available that pauses the program and gives control to the debugger. The debugger can then place breakpoints by replacing the instruction at the breakpoint with the pause instruction, and then restoring the original instruction before continuing execution. Such breakpoints are called a soft breakpoint. Stepping can be accomplished by placing a breakpoint at all possible next instructions or lines of code.

In a self-hosted debugging environment, such a pause instruction is the only hardware support that is needed. Furthermore, all this instruction needs to do is cause a trap, because in an OS environment, all traps are serviced by the OS.

Things become more complicated when external debug support is needed or if there is no OS. In the simplest case, the debug connection is still controlled by software running on the processor. This software resides in the debug trap handler and the startup code. The latter is necessary in order to set breakpoints before the program starts. The debug trap handler is then written such that it does not return control to the interrupted program until the debug software gives the continue command, and until that time, it services memory and register access commands.

A complication arises when the trap handler code is also to be debugged. This is because a trap cannot normally be interrupted immediately by another trap without losing at least the original contents of the trap point register. One way to solve this is to have a second set of trap identification registers used solely for debug traps. The ST200 processor series is an example of an architecture that uses this approach [20].

The link between the host and the target can also be managed by hardware. In that case, the pause instruction does not cause a trap, but instead physically pauses the processor until the debug hardware restarts it. This approach is found often in simple microcontrollers, which often do not support traps. The Atmel AVR `BREAK` instruction is an example of this [23, pp. 27].

Furthermore, microcontrollers can often execute code directly from non-volatile memories such as flash. In this case, it may be undesirable or even impossible to set breakpoints by temporarily replacing instructions. Instead of this, processors may also have breakpoints implemented in hardware by comparing the PC against a number of breakpoint registers, and halting the processor or causing a trap if there is a match. This is called a hardware breakpoint. As there may be many possible next instructions in a program and the number of breakpoint registers is limited, a special step breakpoint may also be implemented. Instead of matching the PC against a value, a step breakpoint always matches when it is enabled, except for the first instruction executed after the continue command. An example of a processor that uses this approach to debugging is the ARM Cortex-M0 [24].

Thus far we have not discussed watchpoints. In complex processors such as those used in a computer, this may be done using the memory management unit (MMU) by temporarily revoking read and/or write permissions to a page of memory. When the program then attempts to access a memory location within that page, a page fault trap occurs, allowing the operating system to respond accordingly. If no MMU is available, watchpoint registers can be used. These are implemented similar to breakpoint registers.

Some processors also support what is known as tracing. Unlike debugging, tracing does not require a program to be stopped in order to get information about it. Instead, a hardware component monitors the state while the program is running and streams information to memory or to an external debug interface at some level of detail. As an example, the ARM CoreSight MTB-M0+ block allows all branches of a Cortex-M0+ processor to be traced to a memory [25]. More sophisticated implementations may also trace data. However, as more information is traced, bandwidth of the trace data stream increasingly becomes a problem, and the processor may be slowed down.

2.2.5 Exploiting parallelism

Exploiting parallelism in programs is key to improving performance. [26] distinguishes four forms of parallelism, namely instruction level parallelism (ILP), data level parallelism (DLP), thread level parallelism (TLP) and request level parallelism (RLP).

- ILP deals with shuffling instructions and their execution stages around to allow them to be executed in parallel.

- DLP deals with algorithms that perform exactly the same operation on multiple datasets at a time. A graphics processing unit (GPU) is a prime example of a processor that exploits DLP.

- TLP deals with algorithms that can be separated into multiple tasks called threads. Threads can run simultaneously. They can communicate with and wait for each other, but otherwise they operate independently. This allows them to be spread over multiple processors in a multiprocessing system.

- RLP handles applications wherein many completely independent algorithms need to be run, such that they can be distributed over a large number of servers dynamically.

Examples of such applications are found throughout the Internet, where many users make requests to webservers simultaneously.

This thesis focuses on ILP and TLP exclusively.

TLP Let us discuss TLP first, as it is simple from a hardware and compiler point of view compared to ILP. As already implied above, TLP allows programs to make use of multiple processors within the same system. It is sufficient and common for such processors to only be connected to each other through a memory and perhaps interrupts. It is up to the programmer to extract TLP and design the program such that it uses multiple threads when it can.

Thus, to a hardware designer, exploiting TLP can be as simple as duplicating a processor a number of times. The only problem that arises, is efficiently handling the multiple accesses to the shared memory. This is an issue in particular when each processor has its own data cache, and may thus have a local copy of the shared data. This local copy needs to either be updated or invalidated in some way after another processor writes to the shared data. This problem is called cache coherence and will be discussed further in Section 2.3.1.

ILP To exploit ILP, instructions need to be executed in parallel in some way. We have already discussed pipelining in Section 2.2.2, which accomplishes this by splitting instruction execution up into stages, each stage processed by a separate functional unit. This allows a stage to start working on the next instruction as soon as the stage finishes instead of needing to wait for the entire instruction to be completed, thereby executing instructions in parallel.

This can be taken a step further by instantiating multiple compute units per stage. For instance, if two compute units are instantiated for each stage, the pipeline diagram for four independent instructions would look as shown in Figure 2.9. Such processors are called multiple-issue processors, as they issue multiple instructions per cycle.

Cycle	1	2	3	4	5
Insn. 1	IF	EX0	EX1	WB	
Insn. 2	IF	EX0	EX1	WB	
Insn. 3		IF	EX0	EX1	WB
Insn. 4		IF	EX0	EX1	WB

Figure 2.9: Pipeline diagram for a two-issue processor executing fully independent instructions.

The proposed ways for handling dependencies in single-issue pipelined processors can be extended for multiple-issue processors. Let us focus specifically on handling the data dependence problem, as next PC computation and forwarding extend naturally.

Recall that two solutions were proposed: one where the data dependencies are detected by hardware through interlocking, and one where the compiler does this by taking the exact architecture and pipeline of the processor into account. The situation is similar for multiple-issue processors. Processors that take the dynamic, hardware based approach are called superscalar processors, the alternative is called a very large instruction word (VLIW) processor.

The hardware that superscalar processors need to effectively extract ILP dynamically (dynamic dispatch) is very complex compared to the techniques discussed so far, and is beyond the scope of this thesis. Regardless of this complexity, superscalar processors dominate the general-purpose computing scene, because a program compiled for a certain sequential instruction set architecture (ISA) can run on any processor with any degree of parallelism out of the box [27, pp. 62].

In contrast, VLIW processors require no extra hardware compared to a single-issue processor at all, aside from obviously requiring the instruction execution units to be duplicated. This makes VLIW processors potentially smaller and more energy efficient than superscalar processors. The major downside aside from processor-specific compilation is that superscalar processors are able to take some things into consideration that the compiler typically does not or cannot know about, such as the chance that a branch is taken or not taken based on branch prediction logic.

The high instruction throughput and energy efficiency make VLIW attractive for embedded digital signal processing applications. Two examples of such processor series are the TriMedia by Philips/NXP [28] and the ST200 by STMicroelectronics [20].

Before we continue, let us define some VLIW vernacular, as it will be used extensively throughout this thesis. A bundle is a set of instructions that are to be executed in parallel. It can be referred to as a 'very large instruction word', hence the name VLIW. The individual instructions in a bundle are called syllables. The part of a VLIW that executes a syllable is called a lane. The number of lanes is referred to as the issue width of the processor.

A problem specific to VLIW processors is how to encode the boundaries of a bundle in the instructions. The trivial way to do this is to specify a fixed bundle size, equating bundle boundaries to alignment boundaries. However, it is unavoidable in that case that the compiler will need to insert NOP syllables just to encode the bundle boundaries, as the compiler cannot be reasonably expected to find enough parallelism to fill a bundle all the time, even for a two-issue VLIW. Increased code size brings with it a higher instruction cache miss rate, and thus decreased performance and energy efficiency.

A more intelligent way to encode the boundaries is a stop bit. A stop bit is a bit reserved in every syllable that indicates whether it is the last syllable in the bundle. The hardware cost of stop bits is the need for the instruction fetch unit and next PC logic to handle what is essentially a variable length instruction encoding.

Going back to ILP in general, there is a limit to the amount of ILP that can be extracted from a program. This depends greatly on the compiler and the processor architecture. The causes for this are beyond the scope of this thesis, but the implications are relevant. Specifically, the implication that there is an asymptotic limit to the speedup that can be achieved in a program by adding more functional units to a superscalar or VLIW processor. Thus, as more and more functional units are added, the increase in performance will no longer outweigh the increased hardware and power consumption of the processor. The tricky thing here, is that because the ILP depends on the program and even the part of the program that is being executed, hardware designers can only estimate how many functional units would lead to good average-case performance.

Idle functional units are an obvious waste of resources. One way in which superscalar processors can mitigate the problem is hyper-threading. A hyper-threaded processor actually runs two or more programs at the same time, sharing the functional units between them, based on which has the highest ILP at a certain instant. This is much harder to do in a VLIW processor however, as VLIW processors are statically scheduled. This is where the ρ-VEX architecture comes into play.

2.2.6 ρ-VEX architecture

The ρ-VEX is a VLIW processor architecture based upon the HP VEX (VLIW-example, [27]), which in turn is based upon the STMicroelectronics ST200 processor family [20]. It is a 32-bit big endian architecture. What sets it apart is that the architecture is designed such that its key metrics are configurable, such as the issue width and the number of available multiplication units. This allows the processor parameters to be tailored to suit the ILP and arithmetic instruction mix of a certain application once the software is available.

As the processor designed in this work is based on the current version of the ρ-VEX, we summarize the architecture in the following paragraphs. A more detailed description is available in Chapter 3 of the ρ-VEX user manual (Appendix C).

The register file of the ρ-VEX is subdivided into three parts:

- The general-purpose registers. These 32-bit general-purpose integer registers are referenced as `$r0.`n. The first of these registers, `$r0.0`, is defined to always read as zero, regardless of what is written to it. The current ρ-VEX implementation has 64 of these.

- The branch registers. These single-bit registers can be used as branch conditions or as carry registers. They are referenced as `$b0.`n. The current ρ-VEX implementation has 8 of these.

- The link register. This register is a single 32-bit register used by hardware to store the return address for a call instruction. In addition, it can be used as a target address for jump instructions.

Notice that the first zero in the register names is constant. This value is used for cluster selection. Clusters are not supported by the ρ-VEX implemented in this work and are therefore beyond the scope of this thesis.

ρ-VEX syllables are encoded as 32-bit words. The operand and destination registers supported differ per syllable, but in general, syllables operate on either two general-purpose source registers or one register and an immediate, and output to an independently selectable general-purpose or branch destination register.

Immediates between -256 and 255 inclusive can be encoded within a single syllable. This is called a short immediate. In order to allow the full 32-bit range to be used, a second syllable is needed. Such a syllable is known as a long immediate syllable. There may be hardware constraints on the relative locations of long immediates and their target syllable. It is the job of the assembler to infer long immediates where necessary.

ρ-VEX syllables can be divided into the following five classes:

- ALU syllables. Syllables in this class can be executed by all lanes, as all lane have an arithmetic logic unit (ALU).
- MUL syllables. Syllables in this class can only be executed on lanes that include a 16x32-bit multiplication unit.
- MEM syllables. Syllables in this class can only be executed on lanes that support memory operations. In addition, the number of MEM syllables that can be executed in a single bundle is currently limited to one by the memory system.
- BR syllables. This class contains all branch instructions. Syllables in this class can only be executed by the lane designated to compute the next PC. Which lane this is depends on the implementation.
- LIMM syllables. This class only contains the long immediate instruction discussed before.

The ρ-VEX does not currently have a hardware division unit, though there are syllables that accelerate software division. There is no hardware floating point support.

Implementations The first softcore implementation of the ρ-VEX architecture was developed in [4]. This implementation is very basic. Firstly, while there is a basic four-stage pipeline, instructions spend multiple cycles in the same stage, so the processor cannot issue a new bundle every cycle. Secondly, changing the configuration of the processor requires manually editing the VHDL sources and resynthesizing. Finally, more complex features such as forwarding, traps and debug support do not exist.

The processor is redesigned in [3]. In particular, the pipeline is completely reworked such that the processor can issue a new instruction every cycle. The pipeline was implemented as five stages: instruction fetch (IF), instruction decode (DE), two execute (EX) stages labeled EX0 and EX1, and writeback (WB). The branch target and condition is computed in DE, the general-purpose register file is read in EX0 and written in WB, and memory operations are performed in EX1. Forwarding logic was also added to further improve performance. Additionally, basic support for interrupts was added by simply forcing a branch. An interrupt controller that makes use of this is designed in [29].

In the following years, among other things, an effort was made to make the processor reconfigurable without reloading the FPGA bitstream. An eight-way ρ-VEX processor was used as the base design. By means of ‘reconfiguration’, it could be configured to work as a single eight-way (1x8), two four-way (2x4), two two-way and a four-way (2x2+1x4) and four two-way processors (4x2). These configurations are shown in Figure 2.10. This reconfigurability allows the processor to dynamically adjust to the available ILP and TLP of the programs to be run. However, the implementation was never refined to allow reconfiguration to be performed *while* a program is running. In particular, if this would be done, the contents of the register file would become undefined.

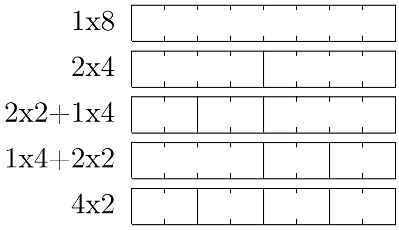


Figure 2.10: Configurable modes of an eight-way reconfigurable ρ-VEX processor. Each cell represents a lane. A broken line separating two lanes implies that the lanes combine together to form a single core, whereas a solid line separates the cores. There are no modes in which a core only uses a single lane because long immediates require at least two lanes to be present.

In order to not have to maintain separate binaries for eight, four and two-way operating modes, generic binaries were developed [30]. An ρ-VEX generic binary is essentially a normal eight-way binary, with the exception that it is permissible for bundles to be executed in two or four cycles as well, corresponding to four-way and two-way mode respectively. In theory, generic binaries also allow the configuration to be changed while the program is running.

The final modifications made to the ρ-VEX prior to this work are the addition of a trap controller, a rework of the external interrupt controller, and the addition of rudimentary hardware-based external debugging support [5]. These modifications were made to make the ρ-VEX suitable to run a port of ucLinux, a Linux kernel port for the ST200 series [31], the processor family that the ρ-VEX is ultimately based upon.

A significant problem with this trap controller is that it is not completely precise. Recall the definition of preciseness in Section 2.2.3. The trap controller complies with rules 1, 2 and 3, but fails rule 4. The reason for this is that a trap is implemented by simply overriding the next PC, without invalidating subsequent instructions that have already been issued. This makes the controller precise enough to handle interrupts accurately, as for interrupts, it does not matter exactly where they interrupt the program. However, it is not good enough for instructions causing a trap, such as a syscall or a page fault.

Another issue is that the debug controller is incomplete. The problems with it are listed in detail in [5, pp. 46-47]. The most important hardware issues listed are that it is not possible for the debugger to write to the register file or the PC and software breakpoints are not supported due to the lack of instruction cache coherence. An additional problem is that the core is halted by simply disabling the clock. As the processor is pipelined, this means that multiple bundles are interrupted in different stages of execution. In particular, as memory operations and register write-back are done in different stages, the state of the program as seen by the debugger is not consistent. Also, the program counter appears as though it is several instructions ahead of the program state.

Regardless of the current issues, let us look at which parts of the current ρ-VEX design can be reused in this work, to fulfill step one of the method specified in Section 1.3.

- There is no need to change the way in which instructions are encoded, as a stop bit is already incorporated into it, it was just unused thus far. This means that no significant changes will need to be made to the assembler.

- Generic binaries are fully compatible with both runtime reconfiguration and stop bits.
- The functional units (ALU and multiplier) do not necessarily need to be changed, nor does the pipeline.
- The list of configurations supported by the current processor (Figure 2.10) can be used as a reference design, even though the way in which reconfiguration is performed needs to be fully redesigned to support reconfiguration during execution.

2.3 Memory architecture

A processor can only execute at its intended speed if the memory system can keep up with it. After all, regardless of how much parallelism is available in a processor, a program cannot continue if the next instruction has not been received from the memory yet. Thus, the efficiency of the memory of a processing system is a very determining factor of the overall system performance.

As memories get larger, access times get longer, and the energy needed to do a memory access increases. This is because the request and data need to travel a greater distance. A common method for hiding this latency is to simply request more data at a time. If more data is requested than what can be transferred in a single cycle due to the physical limitations of the memory interface, such a request is called a burst access. The requested data is then transferred spread out over multiple cycles, but as there is no need to go back and forth between the memory and the processor, there does not need to be any time between the cycles.

Unfortunately, programs do not always need to access consecutive memory locations. In particular, when dereferencing a pointer residing in memory, the value read from the pointer directly determines the address of the next memory access, preventing parallelization. Thus, a more general solution that does not depend on parallelizable memory accesses is desired.

One solution is to have multiple memories of multiple sizes in the system, and have the programmer specify which pieces of code and data are likely to be used often, so they can be placed in the smaller memories. However, this may be difficult to determine at software design time. A more common solution is to use a cache.

2.3.1 Caches

A cache is a small memory local to a processor that keeps a record of recent memory accesses and their results. That is, whenever a piece of data or an instruction is read from the main memory, a copy of the read data is stored in the cache. As the cache fills up, older records are replaced.

Now, whenever the processor needs a piece of data or an instruction, the cache is first checked to see if it contains a copy. If it does, the long access to main memory is not needed. This is called a cache hit. If the cache is found to not contain the the requested data, the main memory needs to be accessed. This is called a cache miss.

This algorithm works because of a phenomenon exhibited by programs called temporal locality. That is, if a program recently accessed a piece of data or executed an instruction, it is likely to access that data or execute that instruction again in the near future. In fact, it is observed that an average program spends 90% of its execution time in only 10% of its code [26, pp. 45].

In addition to temporal locality, programs also exhibit spatial locality. This states that not only the same piece of data that has been accessed is likely to be accessed again, but data at nearby addresses are as well. Therefore, it may make sense to do a burst access when data not in the cache is requested, even if the processor does not request all of it at first. It is particularly easy to see why this makes sense for instructions, as most of the time, instructions are executed sequentially.

Some systems implement multiple levels of caches. The cache closest to the processor is the smallest and fastest. It is referred to as the level-one cache. Every subsequent level cache is larger and slower, but still faster than the main memory. Such systems are prevalent in high performance general purpose computers. One example is the Intel Nehalem microarchitecture, which has three levels [32].

A cache that services both instruction and data accesses is called a unified cache. Using Nehalem as an example again, its level two and level three caches are unified. However, it has separate instruction and data caches at the first level. Regardless of the processor[3], this usually makes sense for at least the first level, as the instruction path and datapath have different requirements. Most obviously, the instruction path is read-only. A less obvious reason is that spatial locality is more prevalent for instructions than for data. Furthermore, many processors can do an instruction access and a data access in parallel, so the two should be parallelized in some way anyway. In addition, in VLIW processors specifically, instructions are much wider than single data words.

Data organization The memory of a cache is organized in cache lines. A cache line contains one or more words of memory and a tag specifying validity information and the main memory address that the line data is a copy of. The simplest type of cache to conceptualize is one where the full memory address is encoded in the tag, allowing any memory address to be saved in any cache line. Such a cache is called a fully associative cache. In practice, such caches are only feasible for a small amount of lines, because every time an access is made, the address of the access must be compared against the tag of every valid cache line.

On the other end of the spectrum, we can define that every main memory address maps to a single cache line. That way, only a single tag needs to be compared when an access is made. Such a cache is called non-associative or direct mapped. Midway solutions are also possible. For instance, a cache where a main memory address can be placed in one of two lines is called two-way set associative. Such a cache can be considered to actually be two parallel caches with shared control logic, each having one line mapping for each address. These 'sub-caches' are called sets.

[3] Except for processors based on the von Neumann architecture, which only have a single interface used for both data and instruction accesses.

In order to find a piece of data in a cache, the request address is split up into three parts: the tag, the index and the offset. The offset selects the word from a given line, the index selects the line within each set, and the tag is compared with the cache tag of the indexed lines of each set. This is illustrated for a non-associative cache in Figure 2.11. For a set associative cache the illustration would need to be duplicated for each set, with some additional logic at the bottom to select the data based on which set hit.

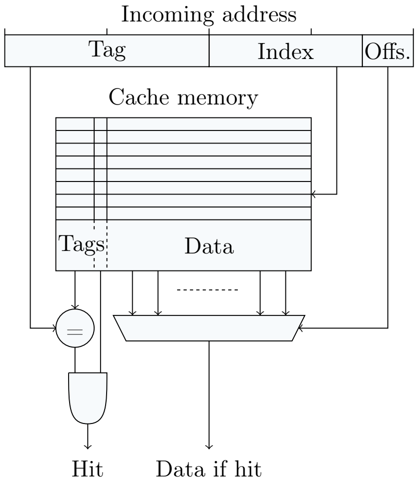


Figure 2.11: Diagram illustrating the read data path of a non-associative cache.

When there is a miss in a set associative cache, it needs to be decided somehow which set will be used to save the data. This is called the replacement policy. There are three ways in which this can be done: randomly, by replacing the least recently used (LRU) line or by replacing the oldest line [26, pp. B-9]. LRU is the best option, but it is also the most complicated to keep track of, as it requires every access to be recorded in some way, independently for each line.

Write accesses Thus far, we have only considered read accesses. There are two main strategies to deal with writes: write-through and write-back [26, pp. B-11]. In a write-through cache, data is written to both the cache and the memory or next level cache at the same time. In contrast, in a write-back cache, writes are written only to the cache. The data will only be written back to the memory or next level cache when the line needs to be removed, usually to make way for another piece of data. This requires the cache tag to include what is known as a 'dirty bit' to keep track of whether the line was modified and needs to be written back or not.

The latency of the memory access can be hidden by buffering the write using a write buffer. This allows the processor to continue executing before the write to memory completes, as long as no read miss occurs.

Furthermore, there are two ways to deal with a write to memory that is not already stored in the cache, i.e., a write miss. In a write allocate cache, the written data is stored in the cache so it can be read later without penalty. In contrast, a no-write allocate cache only forwards the write to the memory or next level cache. Usually, write-back caches use write allocate and write-through caches use no-write allocate [26, pp. B-12].

Consistency and coherence Regardless of how a cache handles write accesses, cached memory accesses made by the processor must appear to be handled in sequential order. This is called cache consistency. An example of a cache that may violate cache consistency without logic to deal with it appropriately is a write-through no-write allocate cache with a write buffer. In such a cache, if the processor writes to an address and misses, then subsequently reads from that address again and thus misses again, the read miss may theoretically be serviced before the write buffer is drained, causing the read to return the previous value.

A similar problem occurs in multiprocessor systems where each processor has its own cache. Because data shared between two processors may end up in the caches of both processors, one of the processors performing a write to shared data does not cause the cache of the other one to be updated automatically.If the caches are write-back, the situation is even worse, as the main memory may not even be updated.

This issue is solved by implementing what is known as cache coherence. There are several ways to do this. In the case of write-through caches, the problem can be solved relatively simply by letting each cache monitor the memory transactions for writes. This is called snooping. When a write transaction of a cached memory location is detected, the local copy of that memory location is either invalidated or updated, depending on the implementation. Dealing with cache coherence is more sophisticated when using write-back caches, as this requires all processor writes to be monitored, which may happen in parallel.

An alternative to snooping is to use a directory based coherence protocol. In such a system, caches keep track of whether their local copies of data exist in other caches as well. Implementations of this approach are beyond the scope of this thesis.

In addition to memory changing because another processor writes to it, some memory locations may also change on their own. This is because not all used addresses actually map to memories in a computing system. For instance, a certain address may be mapped to a counter register that increments every cycle, allowing a program to measure its own performance. Such addresses should never be stored in the cache. An access to such a memory location is called a bypass access, as it bypasses the cache. Whether an access should bypass the cache or not may be determined by one or more bits in the address, or by a non-cacheable flag that is part of the request. In the latter case, the processor may have separate instructions for cached and bypassed accesses.

2.3.2 Memory management units

In an operating system environment, memory needs to be divided among the running processes. In addition, it is desired that processes cannot access the private data of other processes even if they try, in order to prevent bugs in software to be able to affect the entire system. These ‘bugs’ may even be intentionally constructed in order to bypass security features. As it is infeasible for an operating system to check every memory access in software, hardware support is required.

The system that allows this is called a memory management unit (MMU). MMUs translate what are known as virtual addresses coming from an application to physical memory addresses, by looking up the mappings in an operating system memory structure called a page table [26, pp. B-40 - B-41]. Each process has its own page table, to allow different processes to access different pieces of memory, even using the same virtual addresses. In addition to storing the mappings, page tables entries also include flags that independently specify whether the mapped memory may be read, written or executed. Page table entries are cached by a special kind of cache known as a translation lookaside buffer (TLB).

Whenever an application tries to access virtual memory that has not been mapped to physical memory, or it tries to use a virtual memory address in a way it is not allowed to, the MMU causes a page fault trap, allowing the operating system to handle the situation accordingly. Note that a page fault does not imply that the process should be terminated. For example, a page fault may simply mean that a process needs more memory. Therefore, an MMU requires a precise trap controller.

2.3.3 Busses

A bus is a data channel connecting two or more devices together. The interconnect between the largest cache and the memory is an example. Devices can be a master, a slave, or sometimes both. Masters initiate the data transfers, while slaves respond to them. A processor is a master, requesting a data transfer whenever it needs to fetch an instruction or a memory instruction is executed. An example of a slave device is the memory, or memory controller if off-chip memory is used.

In a memory bus, which slave is communicated with is based upon the address that the master wants to access. For instance, a 512 MiB memory may be mapped to address `0x20000000` through `0x3FFFFFFF`. Such mappings should be aligned to their size, such that, in this case, the memory can determine whether it is being selected using only the upper 3 address bits. In order to decrease the amount of bits relevant to the selection further, the memory could also be mapped to a larger area, for instance `0x00000000` through `0x7FFFFFFF`. That does not mean that the memory needs to be bigger: the memory may just ignore the upper 3 bits of the address. That means that `0x01234567` will actually map to the same bit of memory as `0x21234567`, `0x41234567` and `0x61234567`. Such duplicates are called mirrors.

A bus may also have multiple masters. Assuming that the bus can only handle one access at a time, there must be a mechanism to handle the case where two or more masters make a request at the same time. Such a mechanism is called arbitration.

The amount of physical wires dedicated to data bits in a bus is called the bus width. A word is defined to be a value that has the same size as the bus width. Thus, at most, a bus can transfer one word per clock cycle. This is not always possible however, as it takes time to relay the request from the master to the slave, and the reply back from the slave to the master. This delay is called the bus latency. It depends mostly on the physical distance between the master and the slave.

Like memories, a bus may allow multiple words to be accessed in a single transfer. Such an access is called a burst access. Normally, the requested words need to be at consecutive addresses.

There are many implementations of busses available of varying complexity. Examples of busses are Wishbone [33] and ARM's AHB/APB [34].

2.3.4 Peripherals

There would not be much point to having a processing system with just memories and processors: there needs to be a way to interact with the outside world. This is handled by devices known as input/output peripherals. Examples of such peripherals are general-purpose input/output (GPIO) and universal asynchronous receiver/transmitter (UART) controllers. The latter is the low voltage name for an RS232 serial port.

Not all peripherals interact with the outside world. An example of an important peripheral that does not do this is a timer. Timers are peripherals that generate an interrupt a certain amount of time after they are triggered, or periodically at a configurable rate. Timers are essential for preemptive operating systems. Another example is the interrupt controller, which allows the processor to select which interrupt sources should actually cause an interrupt and which events should be masked out.

Most peripherals are slaves, but there are exceptions. Most notably, a hardware debugging interface peripheral needs access to the same memory that the processor has access to, and as such, it is implemented as a master.

2.4 Conclusion

In this chapter, we presented the theory needed to design an implement a processing system. In Section 2.1, we have seen how FPGAs work and how they may be used to test hardware designs. In addition, we have discussed how hardware designs may be specified. In Section 2.2 a detailed description of processor architecture was given, including a summary of the ρ-VEX architecture in Section 2.2.6. Finally, in Section 2.3, we have reviewed theory about caches and memory subsystems in general.

Design 3

In the previous chapter, we discussed processor design theory in general, and briefly looked at the current ρ-VEX design to see which elements can be reused. In this chapter we put the theory into practice and design the major components of the new ρ-VEX processor and cache. The implementation of these components is left to the next chapter.

We will begin by designing the structure of the control registers of the new ρ-VEX in the first section, as they will be needed for all of the new components. In the following four sections, the reconfiguration system, precise traps, debug support and variable-length instruction support are designed, thereby fully specifying the ρ-VEX processor core. In the subsequent section, the reconfigurable cache is designed. Finally, in the last section, we consider which components can be made design-time configurable, to make the processing system as flexible as possible.

3.1 Control registers

In the old ρ-VEX processing system, all processor control registers except for the program counter (PC) were implemented outside the actual core as a bus peripheral. This allows the processor and the debug interface to access the registers naturally through memory accesses. However, it also requires them to be reimplemented for every processing system. As more and more features were added to the ρ-VEX over time, this became increasingly infeasible. Therefore, the new processor has the control registers built in.

There are two ways to allow a program running on the ρ-VEX access to the control registers. The first is to reserve a part of the address space for the registers. Any access to such an address is then routed to the registers instead of the memory system. The alternative is to implement special instructions that can access the control registers directly.

There are advantages and disadvantages for both methods. A major argument in favor of memory mapping the registers, is that no special toolchain support is needed. Software can just dereference pointers that are hard-coded to the register address in order to use them. An argument against memory mapping is that a portion of the address space will be inaccessible. In addition, writing a constant value to a constant memory location requires two instructions in the ρ-VEX instruction set. However, as the project requirements state that the ρ-VEX should be compatible with the current toolchain, the memory mapped control register approach has to be taken.

A distinction is made between control registers that are local to a thread and control registers that are relevant to the ρ-VEX as a whole. These are called context control registers and global control registers respectively. Examples of context control registers are the PC and trap point registers. Examples of global control registers are the current configuration word and reconfiguration status registers.

When the ρ-VEX processor is runtime configured to run multiple threads in parallel, multiple memory accesses can be done at once. That means that there can also be multiple control register accesses at the same time. In fact, the debug port may be doing an access simultaneously as well. Therefore, the control register file either needs multiple access ports or arbitration logic that can stall part of the processor.

In order to avoid simultaneous accesses as much as possible, ρ-VEX threads are specified to only be able to access their own context control registers. Furthermore, they are mapped to the same addresses for each, preventing the need for storing a control register base pointer for each thread. This also allows a read-only thread identification register to be made. Finally, it reduces the size of the address space reserved for accessing control registers.

Furthermore, while all threads and the debug port can access the global control registers simultaneously, only the debug port is specified to be able to write to it. As we will see in Section 4.2.10, most global control registers are read only and have fixed values, for instance to identify the design-time configuration and core version. Since such registers can be implemented as distributed RAM at little hardware cost, having multiple access ports is expected to not be a significant hardware cost.

Finally, when the debug port accesses context control registers, the entire processor is simply stalled for one cycle. This allows the register access to be performed using the same datapath that a memory instruction would use. Debug port accesses are not expected to be common enough for the delay cycle to be significant.

It should be noted that systems with a central memory/peripheral bus are likely to connect the ρ-VEX debug port as a memory-mapped peripheral, as this is required to allow an external debug interface access to the debug port if it is simply implemented as a bus master. This means that, in such a system, the ρ-VEX would also be able to access its own debug port through the bus and cache, thus allowing full access to any control register. The cost of this is, of course, the delay and energy incurred by needing to do a system-wide bus access.

3.2 Reconfiguration

In this section, we will design the runtime reconfiguration system. We will begin by setting up requirements, and then design the way in which reconfigurations are requested, decoded and committed.

3.2.1 Lane configurations

Recall Figure 2.10, showing the configurations that are supported by the old processor. Observe that the structure of the possible configurations is that of a binary tree. Thus, the available modes can be generalized to any number of lanes, allowing the issue width of the processor to be any power of two in theory. Furthermore, the size of a group of lanes that can not be split further can be set to any size. Such an atomic group of lanes will be referred to as a lane group. Notice that if the size of a lane group is set to the issue width of the processor, only one configuration is left, thus nullifying the reconfiguration system.

In order to make the new processor as flexible as possible, let us define both of these parameters to be configurable to powers of two at design time. This allows the size of the processor and level of reconfigurability to be tailored to the application at hand. In further discussions, we will imply an eight-way ρ-VEX with a lane group size of two unless otherwise specified.

3.2.2 Context switching

When the ρ-VEX runs in 4x2 mode, four threads are run in parallel. Each thread has its own state, called a context, consisting of the register files, PC and other control registers. This implies that the registers that store this context need to be instantiated four times. With this in mind, we can think of reconfiguring as changing the interconnect between the contexts and the lane groups. The degree in which this interconnect can be changed determines the available configurations.

It is sufficient to specify that each context is always connected to the same lane group, and that if lane groups are configured to work together, the context of the lowest indexed lane group was chosen. This connectivity is depicted in Figure 3.1.

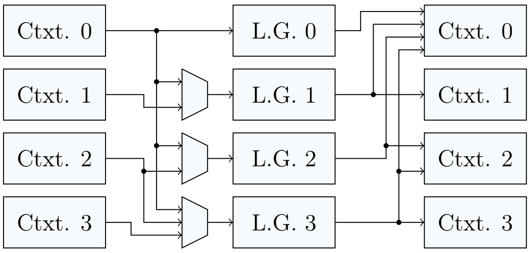


Figure 3.1: Minimal connectivity needed between contexts and lane groups to make all the configurations depicted in Figure 2.10 possible. The connections on the left show the multiplexing between context registers to the lane groups. The connections on the right show the number of lane groups that a context can be connected to.

The advantage of this interconnect configuration is that it is optimal in terms of area. However, there is also a significant disadvantage. With this interconnect, only the program using context zero can be reconfigured to run in two-way, four-way and eight-way mode. Context two can only run in two-way and four-way mode, and contexts one and three can only run in two-way mode.

To illustrate why this is undesirable, consider the following use case. An application has four threads that do independent computations. The computational complexity is dependent on the input, and cannot be easily determined in advance. This program is ported to the ρ-VEX by mapping each thread to its own context, and the ρ-VEX is initially run in 4x2 mode. When a thread completes, it is desirable to make the lane groups that were used for that thread available to the other threads — after all, this is the ideology behind the ρ-VEX architecture. However, if context zero or two are the first to complete, there is no way to remap lane group zero or two to one of the other contexts. At least, not without swapping the contexts through memory in software.

The only way to completely solve this issue is to allow every context to be connected to any lane group. This is depicted in Figure 3.2. As flexibility is considered to be more important than area or cycle time in this work, this is the option that is used. It is left as future work to determine if this full interconnect is worth its additional complexity.

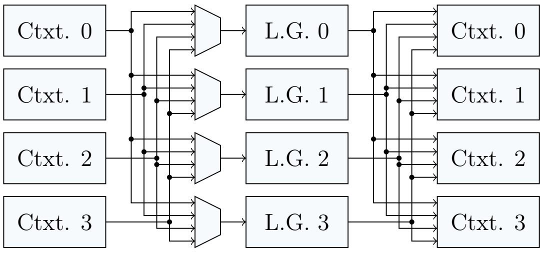


Figure 3.2: Full connectivity between contexts and lane groups, in contrast to the minimal connectivity shown in Figure 3.1. This is the connectivity implemented in this work.

For further flexibility, let us define that the number of contexts implemented should be independently configurable at design time from the number of lane groups. This should be trivial to implement in the hardware design using the full connectivity method. The reasoning for this is, that even in an ρ-VEX processor without multiple lane groups, multiple contexts can be used to accelerate task switching in a multitasking operating system. That is, it allows task switching to be done in hardware through reconfiguration as long as enough ρ-VEX contexts are available.

3.2.3 Disabled lane groups

It is also desirable to be able to not assign a lane group to a context at all, effectively turning it off. This is not that important on a field-programmable gate array (FPGA) development platform where static power consumption dominates, but when implemented on an application-specific integrated circuit (ASIC) combined with clock gating and/or power domains, it could be used to significantly reduce power consumption.

In fact, with this feature in mind, a design-time configuration with less contexts than lane groups can also make sense, as it would allow part of the processor to be powered down while continuing to run the application at decreased speed. This makes sense especially when the application does not always have enough instruction level parallelism (ILP) to use all lanes efficiently.

3.2.4 Requesting a reconfiguration

In the old ρ-VEX processor, reconfiguration could only be done using the debug interface. However, it is desired that the ρ-VEX can do this autonomously as well. The simplest way to allow this is to make a control register that issues a reconfiguration request when written, based on the value written to it.

As there are multiple sources which could theoretically request a new configuration simultaneously, some kind of arbitration needs to be done. What scheme is used to determine which source wins arbitration is not important as simultaneous accesses are unlikely. However, it is important for a program to be able to detect if it lost arbitration, so some kind of status register is also needed.

To illustrate this, consider the following example. Two threads are performing computation jobs in parallel from a shared queue of pending jobs. When they need to enter a critical section to claim the next job from the queue, reconfiguration may be used, by simply claiming all the resources of the processor. This effectively halts the other thread, ensuring private access to the shared resource. At the same time, the critical section will execute as fast as possible, as the thread now has all computational resources available to it. Now consider the scenario where both threads want to enter a critical section at the same time. They both request a reconfiguration, but one thread inevitably loses arbitration and is stopped by the other. When the winning thread leaves the critical section, it allows the losing thread to run again. Now, if there is no way for the losing thread to determine that it has lost, it must assume incorrectly that it has successfully entered the critical section when it really has not.

Another special situation that should be handled in some way is an erroneous reconfiguration request due to a software bug. It is desirable to detect such errors, as they would lead to completely undefined behavior, possibly also compromising the debugging system, especially in self-hosted debug mode. One way to deal with an invalid reconfiguration request is to trap the requesting thread, but this would require the request to be validated in a single cycle. Another way is to simply ignore invalid requests while setting an error flag in a status register. Since a status register is already needed for determining the source that won arbitration, it makes sense to just add the flag to that register.

It is desirable to allow an interrupt to trigger a reconfiguration automatically as well. The only hardware requirement for this is an additional control register and an additional input to the arbitration logic. Two use cases for such a system are presented to illustrate why this is desirable.

The first use case is power saving. With the system as described thus far, it is already possible to essentially power down the processor by simply disconnecting all lane groups through reconfiguration. However, without this system, there would be no way to recover from that state. With it, a timer peripheral could for instance be set to 'wake up' the processor with an interrupt after a set amount of time passes.

The second use case has to do with interrupt latency. Consider a single threaded program that normally runs in context one. Such an automatic reconfiguration system could then be set up to reconfigure the processor to context zero when an interrupt occurs. Context zero is then immediately trapped due to the pending interrupt. It can then handle the trap without needing to first save the state of the interrupted program or needing to restore it later, as no program was running on context zero in the first place. Considering that the ρ-VEX has a large amount of registers, the time and energy saved because of this can be significant if interrupts are common. When the interrupt has been handled, it manually re-enables interrupts, requests a reconfiguration back to context zero and runs a `while(1)` loop, until another interrupt restarts the whole sequence.

3.2.5 Configuration word encoding

The encoding for the value that is written to the reconfiguration register, called the configuration word, is to be specified. The ρ-VEX is a 32-bit architecture, so it is desirable to limit the size of the configuration word to 32 bits. It is also desirable to make the encoding 'human readable' in hexadecimal form to ease manual debugging.

To accomplish this, let us map each lane group to a nibble. The three least significant bits of each nibble specify the context it should be connected to if the most significant bit is zero, or a special mode if it is one. The only special mode defined corresponds to the nibble `8`, which disconnects the lane group. Values `9` through `F` are reserved for future expansion and are considered erroneous by the decoder. If for instance only four contexts are implemented, values `4` through `7` are also considered erroneous. The nibbles for unimplemented lane groups must be set to zero so they do not need to be specified in the hexadecimal value. Finally, any lane group to context mapping that does not satisfy the binary tree structure discussed in Section 3.2.1 and exemplified by Figure 2.10 is also rejected by the decoder.

To give some examples, consider an eight-way ρ-VEX with four lane groups and four contexts. `0x0012` then specifies a 1x4+2x2 lane configuration, with four lanes working on context zero, two lanes working on context one and the remaining two lanes working on context two. `0x8833` would map to 1x4 mode for context three, with the remaining four lanes in power-down mode. `0x0112` is erroneous, because it does not satisfy the binary tree requirement.

The 32-bit size of the register limits the maximum number of lane groups to eight. That would take at least a 16-way ρ-VEX, twice as large as what is currently considered practical, thus still allowing room for expansion. The three bits per lane group used to encode the context limits the maximum number of contexts to eight; again, twice as much as the nominal value.

3.2.6 Decoding and committing configurations

When a reconfiguration is requested, the configuration word must be error-checked, decoded into various signal formats and finally be committed. One of the things that the ρ-VEX project attempts to prove, is that the area overhead incurred by reconfiguration support is worth the cost. It is thus desirable to make the hardware that fulfills these tasks as small as possible. Furthermore, it is not expected that a practical application will ever need to reconfigure often enough that a delay in the order of ten clock cycles will be significant. Therefore, a multi-cycle decoder is preferred over a single-cycle decoder which uses more area.

As reconfiguration requests are thus not instantaneous, the way in which a thread advances after requesting a reconfiguration is to be determined. Perhaps the most obvious way to deal with this would be to halt its execution immediately after it makes the request. However, this would require instructions that have already been fetched to be invalidated, only for them to be fetched again when the context resumes. In contrast, if it is not required that the instructions immediately following a reconfiguration request execute in the new configuration, decoding the new configuration can be done in parallel to further program execution. A blocking reconfiguration request can then still be emulated in software if necessary with the addition of a 'busy' flag in the reconfiguration status register.

A second design choice concerns how a new configuration is committed. In particular, the decoded configuration signals can be made part of the pipeline. Alternatively, the reconfiguration controller could request the affected contexts to stop fetching instructions and then wait for all affected lane groups to drain their pipelines before committing the configuration. While the former is faster, the latter has several advantages and is therefore used. First of all, it requires less area, as the logic needed to correctly halt and resume execution will already be necessary for the debug system. Secondly, this implementation allows external sources to delay the reconfiguration as well: as we shall see in Section 3.6.3, the write buffer in the cache needs to be empty before reconfiguration to guarantee consistency. Thirdly, it ensures that all the register files of the ρ-VEX relating to the reconfigured contexts are in a consistent state.

3.3 Trap control logic

In this section, the major design choices for the precise trap control logic are listed. We will first specify the trap system in general, and then determine how traps are affected by runtime reconfiguration.

3.3.1 Trap identification

We have seen in Section 2.2.3 that there are many ways in which a processor can convey information about a trap to the application. Thus, before anything else, let us decide on the approach that will be taken in the new ρ-VEX processor.

The first choice is whether to make the trap handler address dependent on the trap. The obvious advantage of having a separate trap handler for each trap is that there does not need to be any logic in the handler to determine what the cause of the trap is. However, a disadvantage is code size, as saving the state of the interrupted program requires in the order of a kilobyte worth of instructions in the ρ-VEX due to the large number of registers. This also affects instruction cache performance. Thus, it makes sense for the ρ-VEX to have only a single trap handler for as far as the hardware is concerned, and let the software branch to the appropriate handler after the state is saved.

The address of the trap handler is specified to be configurable using a context control register. This allows each thread to specify its own trap handler, if necessary.

Having a single trap handler for all traps requires that the cause of the trap must be explicitly identifiable in some way. In the ST200, this is done using two control registers: `EXCAUSE` and `EXADDR` [20, pp. 22, 78-79]. `EXCAUSE` is one-hot encoded, allowing up to 32 different traps. `EXADDR` is a 32-bit value that holds additional information for some of the traps. A similar approach will be used in the ρ-VEX, although the trap cause register is 8-bit and is encoded as an integer. `EXADDR` is called the trap argument in the ρ-VEX, because it will be used for holding information other than addresses as well, such as an external interrupt index, or the index of the lane that caused the trap.

3.3.2 Returning from a trap handler

In the ρ-VEX architecture, the regular indirect jump instruction (`IGOTO`) can not be used to return from a trap. This is because it would require the link register to be set to the trap return address, whereas the link register needs to be restored to the value it held at the time of the trap, just like every other register. For this purpose, the VEX architecture has a special instruction, `RFI`, that branches to the interrupt return register instead [27, fig. 9.1]). This instruction is used in the new ρ-VEX processor for returning from a trap, replacing the VEX interrupt return register with the trap point context control register.

A preempting operating system kernel must be able to set the trap return address in order to switch to a different thread. Thus, it is required that the trap point register is writable.

3.3.3 Nested traps

One thing we have not paid attention to thus far is the possibility of a nested trap. Nested traps are unavoidable if a fault occurs within the trap handler. If, at that point, the trap handler has already saved the state of the interrupted program to the stack, nested traps are not a problem unless the stack overflows. However, if a trap occurs while the state is being saved, it may not be possible to restore the state of the originally interrupted program. The same is true near the end of the handler, as the trap point register is used as the trap return address, and thus needs to be restored at some point.

This problem is solved as follows. A context control register flag called ready for trap (`RFT`) is specified. The value of this flag selects between two separate trap handler address registers. The first of these, simply called the (normal) trap handler, is selected when `RFT` is set. The alternative handler, called the panic handler, is selected when `RFT` is cleared. `RFT` is automatically cleared after the processor jumps to the trap handler or panic handler. Its previous state is copied to a different register. When the handler executes the `RFI` instruction to return to the interrupted program, the value of `RFT` is restored from the copy.

As the name suggests, the flag specifies whether the program is ready to receive a trap, or if it is currently saving or restoring registers within the trap handler. It is the responsibility of the software to set the `RFT` flag as soon as possible after saving the state of the interrupted program, and to clear it again as late as possible before restoring the trap point register.

With this mechanism in place, the normal trap handler is only used when the program is in a state that can be restored, while the panic handler is used in cases when this is not possible. The difference is thus that the trap handler can return to the application, while the panic handler cannot. The panic handler can thus only be used to report a serious problem with the program before stopping execution entirely or resetting.

3.3.4 Interrupts

Unlike faults, interrupts can be postponed. The act of disabling interrupts is called interrupt masking. It is important for a processor to be able to mask all interrupts. To understand this, consider the case where two interrupts occur almost at the same time.

It may then happen that the second interrupt occurs while the trap handler is still saving the state of the interrupted program, causing the panic handler to be run if this interrupt cannot be masked.

To accomplish this, we specify a flag very similar to `RFT`: interrupt enable (`IEN`). When this flag is cleared, interrupts are masked, causing only fault traps to be handled. Like `RFT`, `IEN` is automatically saved and cleared when the trap handler is entered, and restored when `RFI` is executed.

The software may wish to mask interrupts temporarily for many reasons, for example when entering a critical section in a sequential program. Obviously, it may do this by writing to `IEN`. However, the ρ-VEX is a pipelined processor. That means that the instructions following the instruction that clears the `IEN` flag have already been fetched by the time that `IEN` is actually modified. If interrupts are handled like they are in the old ρ-VEX processor, i.e., by simply forcing the next PC to the handler, an interrupt could thus appear to occur after an instruction that clears `IEN`. This is illustrated in Figure 3.3. With the old processor, this was handled by inserting a sufficient amount of no-operations (NOPs) after `IEN` is cleared, before actually entering the critical section. Note that a similar situation occurs when re-enabling interrupts, causing the interrupt to be delayed unnecessarily, as shown in Figure 3.4.

				Interrupts disabled		
Cycle	1	2	3	4	5	6
Insn. 1: disable int.	IF	EX0	EX1	WB		
Insn. 2		IF	EX0	EX1	WB	
Trap handler			IF	EX0	EX1	WB

Figure 3.3: Pipeline diagram of an interrupt that is handled by simply setting the PC, as is done in the old processor, seemingly occurring after interrupts are disabled. The control register that disables the interrupts is updated between EX1 and WB.

				Interrupts enabled			
Cycle	1	2	3	4	5	6	7
Insn. 1: enable int.	IF	EX0	EX1	WB			
Insn. 2		IF	EX0	EX1	WB		
Insn. 3			IF	EX0	EX1	WB	
Trap handler				IF	EX0	EX1	WB

Figure 3.4: Pipeline diagram illustrating unnecessary interrupt delay after re-enabling interrupts when an interrupt is handled by simply setting the PC, as is done in the old processor. The interrupt enable control register is updated between EX1 and WB.

Instead, the new processor handles interrupts as if they are traps occurring in the stage immediately before the clock edge that updates `IEN`. This is illustrated for re-enabling interrupts in Figure 3.5. By mentally replacing instruction 2 with one that disables interrupts instead, it can be seen that the last instruction that can then be interrupted is instruction 2.

					Interrupts enabled				
Cycle	1	2	3	4	5	6	7	8	9
Insn. 1	IF	EX0	EX1	WB					
Insn. 2: enable int.		IF	EX0	EX1	WB				
Insn. 3			IF	EX0	*EX1!*	~~WB~~			
Insn. 4				IF	EX0	~~EX1~~	~~WB~~		
Insn. 5					IF	~~EX0~~	~~EX1~~	~~WB~~	
Trap handler						IF	EX0	EX1	WB

Figure 3.5: Pipeline diagram illustrating approximately how the new processor handles an interrupt that was pending while interrupts were disabled. The interrupt enable control register is updated between EX1 and WB. In reality, the latency is slightly longer, as we will see in the implementation.

While this may be somewhat less performant than the old processor in some cases, no NOPs are required. This allows the control register to be written directly from C without needing to somehow force the compiler to insert NOPs after the write. It is also easier to work with in assembly, as one does not have to mentally visualize the pipeline to use `IEN` correctly.

Interrupts need to be identified in some way, just like any other trap. We have already specified that traps in general are identified by an 8-bit trap cause and a 32-bit trap argument. In order to not make the trap cause definitions dependent on the processing system, a single trap cause value is defined for all interrupts, and the trap argument is used to identify the interrupt.

3.3.5 Dealing with runtime reconfiguration

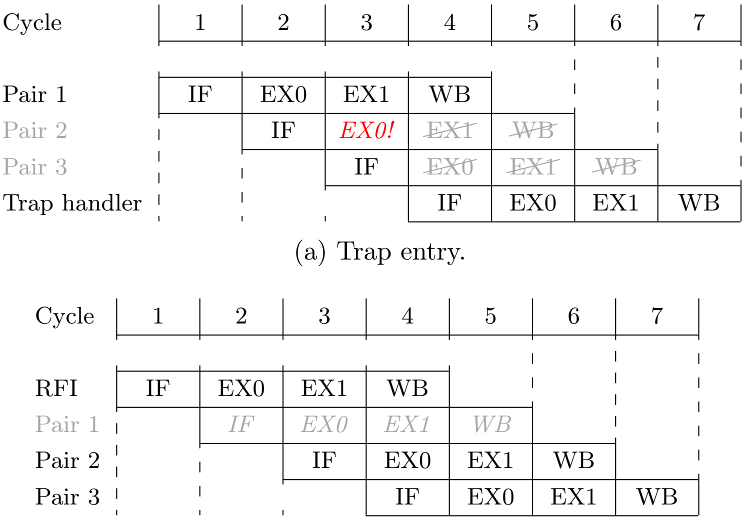
For the most part, runtime reconfigurability does not significantly affect the design of the trap control logic. However, there is one notable exception. Consider the case where the ρ-VEX is running a generic binary thread in two-way mode and a trap occurs. While the trap handler is being executed, the ρ-VEX reconfigures that thread to four-way or eight-way mode. It is then possible that the trap handler needs to return to an address that is not actually the start of a four-way or eight-way bundle.

If variable-length bundles are supported (one of the goals of this project), this is not necessarily a problem. The processor would then simply execute the remaining syllables in the bundle until it encounters a stop bit. However, variable-length instruction support is rather expensive in terms of area and cycle time, and it is thus desirable to make it an optional feature, selected at design-time.

Without variable-length bundles, it is a requirement that the fetched bundle is always aligned to the current bundle size. That is, if the ρ-VEX is running in eight-way mode, the fetched address must be 32-byte aligned. As we have illustrated, the PC may not actually meet this requirement after returning from a trap. The way in which this will be handled is by rounding the PC downward to the previous alignment point before forwarding it to the instruction fetch logic, and disabling syllables that reside at addresses before the real PC.

There is a second problem with returning from a trap that has to do with long immediate syllables. As we will see in Section 3.7.2, long immediate syllables can reside in the other syllable within an aligned syllable pair, or within the preceding syllable pair within a bundle, shown in Figure 3.12. The latter case is handled in two- and four-way mode by checking the subsequent pipeline stage (corresponding to the previous instruction fetch) for long immediates. Without additional logic, these long immediate syllables would be lost when the trap handler returns to a mid-bundle address, as the state of the pipeline stage registers is not restored.

The approach used to solve this problem is to check whether the trap point is aligned to a generic binary bundle when returning from a trap. If it is not, the processor jumps to the trap point minus the current instruction fetch size, and immediately invalidates the syllables fetched in that instruction to prevent them from executing again. This ensures that the long immediates are in the pipeline as they should be, if any exist. This process is called long immediate prefetching, and is illustrated in Figure 3.6.



Cycle	1	2	3	4	5	6	7
Pair 1	IF	EX0	EX1	WB			
Pair 2		IF	*EX0!*	~~EX1~~	~~WB~~		
Pair 3			IF	~~EX0~~	~~EX1~~	~~WB~~	
Trap handler				IF	EX0	EX1	WB

(a) Trap entry.

Cycle	1	2	3	4	5	6	7
RFI	IF	EX0	EX1	WB			
Pair 1		*IF*	*EX0*	*EX1*	*WB*		
Pair 2			IF	EX0	EX1	WB	
Pair 3				IF	EX0	EX1	WB

(b) Trap return.

Figure 3.6: Pipeline diagram illustrating long immediate prefetch in two-way mode after returning from a trap to a mid-generic-bundle address. Note that the trap point points to pair 2. The prefetch of pair 1 is needed because pair 1 may contain a long immediate for pair 2, and must therefore be in the pipeline, even though it must not be executed. Note that the depicted latency from trap to handler is shorter than it is in the implementation.

3.4 Debug interface

In this section, the debugging interface is designed. We first select the overall approach to use, and then define define the which features are to be implemented.

3.4.1 Self-hosted vs. external debugging

In Section 2.2.4 we have seen two distinguishing approaches to debugging: self-hosted and external debugging. We have also seen that, in the latter case, the link between target and host can be managed in software or in hardware.

As the ρ-VEX is closely related to the ST200 series, it is tempting to copy its design to the ρ-VEX. That is, in order to give control over the debugged process to the debugger, a debug trap is generated, handled by an alternate trap handler with alternate state saving registers [20, pp. 106-107]. This method supports self-hosted as well as external debugging, as the debug trap handler may either be handled by the operating system (OS) if there is one or by a piece of code that communicates with an external debugger.

A major disadvantage of this method, however, has to do with the fact that the ρ-VEX is still actively being developed. Therefore, the hardware that the debugging software runs on may contain bugs as well, possibly compromising the debug link. Obviously, in a hardware-based debugging solution, the hardware could also be compromised, but this is expected to be less likely, as it is unlikely to be modified as much as the rest of the processor. It is thus more desirable to use a hardware-based solution. However, keeping self-hosted debug support is also desirable, in order to support OS-based debugging in the future.

In order to satisfy both requirements, we specify that conditions encountered during program execution that require control to be handed to the debugger are handled as a special kind of trap. How these special traps are handled depends on whether an external debugger is connected.

The ρ-VEX boots up in self-hosted debug mode. In this mode, debug traps are handled just like normal traps, going either to the trap handler or panic handler. This is sufficient as long as the trap handler itself never needs to be debugged in this way, which is not necessary when debugging an application running in an OS environment. In order to prevent accidentally setting off a breakpoint while inside the trap handler, debug traps can be masked using a breakpoint enable flag that works in the same way as the `RFT` and `IEN` flags (Sections 3.3.3 and 3.3.4). To make this flag work intuitively, debug traps must be generated in the memory request stage, like interrupts.

When an external debugger connects to the ρ-VEX, it may set a context control register flag to switch to external debug mode. In this mode, all writes to debug-related control registers by the software running on the ρ-VEX are ignored, and debug traps are handled by halting the processor instead of jumping to the trap handler. The trap cause is recorded in a special register as the reason for halting.

3.4.2 Breakpoints and watchpoints

In order to support all the basic features mentioned in Section 2.2.4, we define three kinds of debug traps that can halt program execution.

- The single step trap. When activated by setting a single step mode flag in the control registers, this trap is always generated for the second bundle fetched after resuming execution. Because ρ-VEX traps always invalidate the trapped bundle, only a single bundle is committed.

- The hard breakpoint/watchpoint traps. These are generated by hardware when the PC or the address of a memory operation matches the respective breakpoint/watchpoint address context control registers. These traps can be configured individually to be a breakpoint, a write watchpoint or an access watchpoint, and they can of course be disabled.

- The soft breakpoint trap. These are never generated by hardware, but may be generated from code using the `TRAP` syllable. This allows a debugger to set an unlimited amount of breakpoints by replacing existing syllables.

In addition to these traps, an external debugger can also halt the program manually by setting the 'break' context control register flag. This is not possible in self-hosted debug mode.

3.4.3 Reconfiguration and parallel programs

Reconfiguration, and thus parallel execution, is handled by giving each ρ-VEX context its own debug interface. This is easily done by making all debug-related control registers context-specific. This essentially makes each ρ-VEX context behave like an independent single-core processor. This does mean that there is no way to automatically halt execution of a parallel program for all contexts simultaneously by means of a breakpoint or watchpoint, so it may be difficult to debug, for instance, mutual exclusion algorithms. Developing logic for such things is left as future work for the sake of not overcomplicating the first implementation.

3.4.4 External debug link and software

In the old processor, the JTAG debugging interface supplied by GRLIB (`ahbjtag`, [35, pp. 50-55]) was used for communication between the ρ-VEX and the host. This interface simply allows access to the memory bus, thus also to the ρ-VEX control registers. In addition to the JTAG connection, a universal asynchronous receiver/transmitter (UART) serial port peripheral is used to allow the application running on the ρ-VEX to output debug information or interact with the user. The ML605 and VC707 have onboard converters for JTAG and UART to USB, allowing the FPGA to connect to the host computer using two USB cables.

As mentioned in [5, pp. 45] however, interfacing with the USB-JTAG converter efficiently on the host side is difficult. For this reason, a new peripheral is designed, based completely on the UART. Because there is only one USB-UART interface available on the development boards, both the bus transaction commands and the arbitrary user-application data have to be encoded in a single hardware-managed protocol. Error detection and retransmission is incorporated into the protocol in order to protect bulk memory transfers.

While interfacing with a UART serial port on a computer is simpler, the theoretical bandwidth limit is much lower. Specifically, the onboard USB-UART converter only supports bitrates of up to 115200 bits/second. This limits the transfer speed to 11520 bytes/second[1]. In comparison, the JTAG interface reaches about one MiB per second in practice. However, the UART-based interface is not mutually exclusive with the GRLIB `ahbjtag` component. Thus, the latter may be used for faster bulk transfers if needed, for example to load the approximately 4 MiB ρ-VEX Linux kernel port.

Even if using a serial port is simpler, interfacing software still needs to be written. In particular, it is desired to allow the existing ρ-VEX `gdb` port to connect to the new debug system. The software written to accomplish this will be presented in Section 4.6.

3.4.5 Trace unit

One of the key novel features of the ρ-VEX processing system presented in this work is the reconfigurable cache. In order to evaluate its performance, a simulation system would need to be written, or alternatively, the hardware could be used, if a way is provided to record (trace) the accesses to the cache. As creating a simulator for the ρ-VEX and this cache would cost considerable effort, it is desirable to use the hardware. Thus, a trace unit is specified as part of the debug system.

In order to be usable for evaluating cache performance, the trace unit must at least be capable of tracing program flow (i.e., branches) and the addresses for all memory instructions. In order to make it more interesting for debugging in general, data written to registers and the written or read data for memory instructions can also be traced. Finally, to be able to debug the instruction cache, the fetched instructions can be traced as well, to allow them to be compared against the program binary.

The output of the trace unit is specified to be a stream of bytes, outputted at a maximum rate of one byte per cycle. This means that the processor will run significantly slower when tracing is enabled, as a single byte is nowhere near enough to encode all information for an instruction. However, the debug link is unlikely to achieve even this bandwidth; recall that the UART based debug interface designed in this work can achieve little more than 10 kiB per second, and even the JTAG interface would not be able to keep up. Therefore, it is more important to have a slow interface that attempts to compress the trace data packets as much as possible than to have a fast interface. The details of this are left to Section 4.2.13.

3.5 Variable-length bundles

In this section, the design for the variable-length bundle support logic is presented. The design can be divided into three parts: the way in which the length of a bundle is encoded, the way in which bundles are fetched, and the way in which the syllables in the bundles are distributed over the lanes. These three parts are handled in the following sections. The variable-length bundle support logic is also presented in the paper attached in Appendix B.

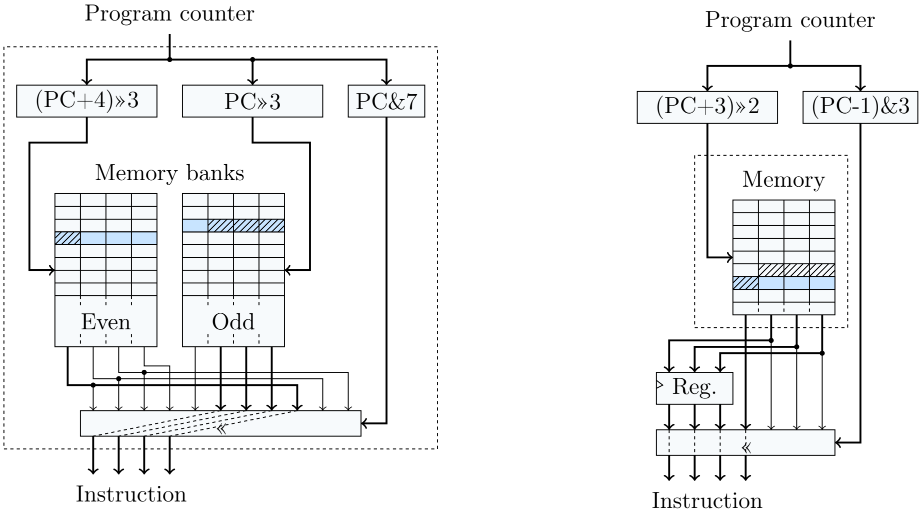
[1] Neglecting byte-level protocol overhead and assuming 8n1 transmission, i.e., one start bit, eight data bits, no parity bit and one stop bit for each transmitted byte.

3.5.1 Bundle encoding

There are many different approaches to encoding the length of a bundle. Without going into detail though, the binaries for the old processor already take this into consideration: each syllable has a bit dedicated to specifying whether it is the last in the bundle. This bit is called the stop bit. In order to retain as much toolchain compatibility as possible, this method is used in the new processor as well.

3.5.2 Fetching bundles

A defining characteristic of any variable-length instruction fetch unit is that it must in some way provide some amount of instruction memory to the processor that is in the worst case aligned to some smaller size than the size of the block being fetched. This is problematic, as a normal memory cannot service such a request in a single cycle. Two possible design approaches are considered: one where the memory and cache are designed to handle misaligned accesses, and one where the ρ-VEX hides the complexity from the memory system. These designs are depicted in Figure 3.7.



(a) Handled in memory.

(b) Handled by the ρ-VEX.

Figure 3.7: Two approaches to modifying the instruction fetch system to handle misaligned bundles for a four-way ρ-VEX, or an eight-way ρ-VEX where a bundle must be aligned by syllable pairs. The unit for the program counter is a syllable or a syllable pair respectively. The datapath is highlighted for PC = 0x15. The area within the dashed rectangle is implemented in the memory or cache; the area outside of it is implemented within the processor.

In the memory-based approach (Figure 3.7a), the memory is divided up into two banks. The access port for each bank is as wide as a maximum size bundle. The left bank contains all even memory locations, whereas the right one contains the odd ones. Both are addressed at the same time with different addresses, decoded from the incoming address in the units at the top. Note that one of these requires a full adder almost the entire width of the memory address. A bitshift is used at the end to select the requested memory region from the fetched data.

In the ρ-VEX-based approach (Figure 3.7b) we essentially need to query the memory twice for a misaligned access, requiring two cycles. A register stores the result of the first fetch. Notice, however, that as long there are no branches, the previously performed fetch will always be one of the required fetches. Thus, we only need a two-cycle fetch when the program branches, and only if it branches to a misaligned PC. Notice that the width of the subtract-one operation is only two bits in this case, allowing it to easily fit within a single level of FPGA logic, but that the adder needs to be almost full width, similar to the adder needed in the memory-based approach. As we will see in Section 4.2.6, however, the adder in the ρ-VEX based solution can be implemented in parallel to the PC+1 adder, and thus does not affect the critical path.

While the ρ-VEX-based approach is slightly less performant when branching, it is chosen instead of the memory-based solution to keep the requirements on the memory system as simple as possible. This should make it easier to design alternative memory solutions for the ρ-VEX in the future.

3.5.3 Syllable distribution

In the old processor (and the new processor if variable-length bundle support is disabled at design time), syllables are mapped to the execution lanes one-to-one based on their location within the bundle. When variable-length bundles are enabled, this is not desirable. The main reason for this is as follows. In generic binaries, any branch syllable must be the part of the last syllable pair. As it is desirable to only support branch syllables in one lane for logic and routing resource reasons, only the last or second to last lane supports them. That means that, if the trivial syllable mapping is used, any bundle that includes a branch must be seven or eight syllables long.

In order to prevent this, we specify that branch syllables must always be the last syllable in a bundle, regardless of the size of the bundle. Initially, the trivial mapping is used. In some pipeline stage before the syllables are executed however, logic will be instantiated to reroute the last syllable within the bundle to the lane that contains the branch unit. Note that, without additional logic, this is only possible for branch syllables, because these are never complemented by a long immediate. The rerouting logic is accompanied by logic that invalidates lanes that are mapped to syllables belonging to the next bundles, i.e., after the syllable with the stop bit set. An example of the instruction routing is shown in Figure 3.8.

It should be noted that it is possible to distribute syllables in such a way that the multiplexers that form the shifter in Figure 3.7b can be simplified. The ST200 uses such an approach, as shown in the paper presented in Appendix B. Such a distribution is more sophisticated to implement in the assembler, however. Such improvements are left for future work.

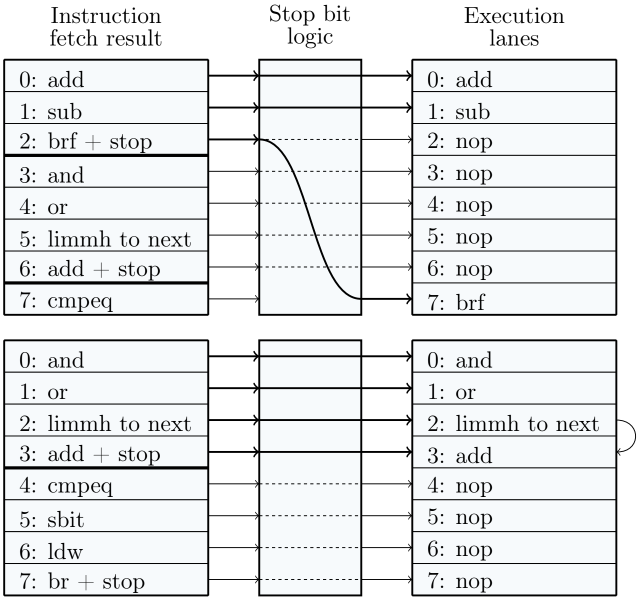


Figure 3.8: Syllable distribution in the new ρ-VEX processor with variable-length bundle support, shown for two example bundles.

3.6 Cache

In order to allow the ρ-VEX to use large off-chip memories efficiently, a suitable cache is needed. As the ρ-VEX does instruction and data accesses in parallel, it needs a separate instruction and data cache. The instruction cache will be treated in the first section, the data cache in the second. The specific project requirement that the cache needs to be coherent and some cache consistency issues relating to reconfiguration are discussed in the third section.

3.6.1 Instruction cache

The ρ-VEX fetches one 32-bit instruction for each of its lanes per cycle. When the reconfiguration features are not present or all lane groups are working together, its instruction fetch ports work together to fetch a single, aligned block of memory. However, when the ρ-VEX is reconfigured to run multiple threads simultaneously, the ports are said to be decoupled, and each can fetch a different block of memory. These blocks are still guaranteed to be aligned on their own, but the size of the block is smaller and thus the alignment constraint is relaxed.

In the most trivial design, the instruction fetch ports of each lane group are simply given their own independent caches. However, such a design suffers from significant inefficiencies when multiple lane groups are coupled. Consider the case where the usual eight-way reconfigurable ρ-VEX is running in 1x8 mode. Also assume, without loss of generality, that we are using caches with a line size of 64 bits, i.e., the size of the access port. Now, lane group zero will now only fetch instructions at line addresses $4i$ for $i \in \mathbb{N}$, lane group one only accesses $4i + 1$, and so on. This implies that only a quarter of the available cache is actually used. This is illustrated in Figure 3.9.

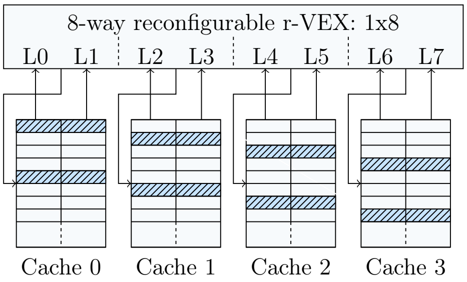


Figure 3.9: Diagram illustrating the inefficiency of using independent conventional instruction caches for each lane group of the ρ-VEX. The cache lines that are used in 1x8 configuration are highlighted.

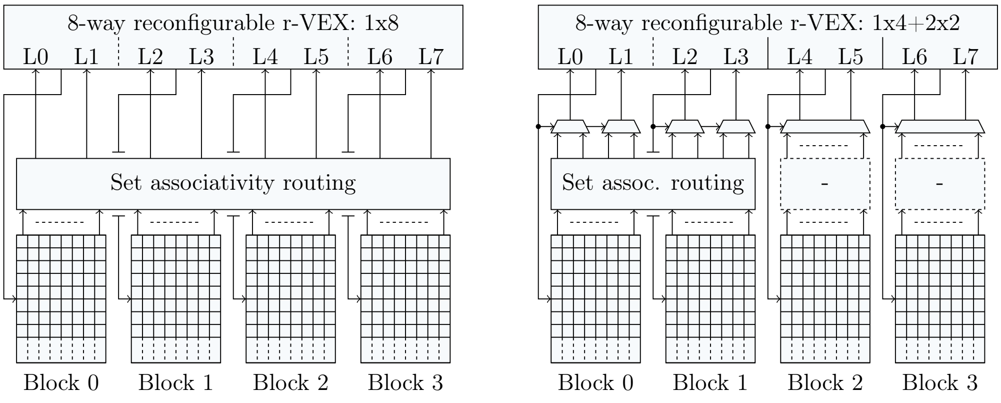
Note that we cannot simply remove the unused locations, as they *are* used in 4x2 mode. One solution could be to ignore two of the least significant bits of the address in eight-way mode, and ignore one in four-way mode. This would however require a cache flush for every reconfiguration, as the cached addresses would not be compatible. Ideally though, the instructions should be stored in the cache in such a way that reconfiguring incurs little to no cache-related performance penalty.

In order to accomplish this, let us start from a four-way set associative cache (or as many sets as the target ρ-VEX has lane groups). Also, let us define the cache lines to be as wide as a full bundle, so 256-bit for an eight-way ρ-VEX. Now, when the ρ-VEX reorganizes its lane groups, we let the cache reorganize its sets along with it. For example, when reconfiguring from 1x8 to 2x4, the four sets are redistributed to form two independent two-way set associative caches. This organization is shown for two configurations in Figure 3.10.

With this cache organization, there is no cache penalty when reconfiguring a thread from a smaller to a wider issue width, as the cache block(s) that it was using in the smaller configuration are still available as sets in the wider configuration. Going in the other direction, half of the cache will no longer be accessible, so there will be some penalty. However, this penalty can be minimized by using an appropriate replacement policy in the set associativity logic.

In the instruction cache implemented in this work, a very simple replacement policy is used: the bits immediately following the cache tag select the set to use. This makes the cache behave like a direct-mapped cache, thereby losing all advantages of a normal set associative cache, but at the same time improving locality within a single set. A status register will be made available to the ρ-VEX indicating which set serviced the most recent instruction fetch, allowing it to shrink a context in such a way that that set is still included in the new configuration.

One way to have both the benefits of a normal set associative cache and the reconfiguration benefits is to use a better replacement policy normally, such as least recently used (LRU), while allowing the ρ-VEX to restrict replacement to a certain subset of


8-way reconfigurable r-VEX: 1x8
L0
L1
L2
L3
L4
L5
L6
L7
Set associativity routing
Block 0
Block 1
Block 2
Block 3
8-way reconfigurable r-VEX: 1x4+2x2
Set assoc. routing
-
-

(a) 1x8 mode.

(b) 1x4 + 2x2 mode.

Figure 3.10: Diagram illustrating the organization of the reconfigurable instruction cache. Instead of showing the generalized reconfigurable logic, the effective organization for two example configurations is shown.

the available sets using a control register some time before it reconfigures. Such more complex replacement policies are left as future work.

3.6.2 Data cache

The ρ-VEX can access one memory location per cycle for each lane group. It can read 32-bit values, or write 8-bit, 16-bit or full 32-bit values. All accesses are aligned. When lane groups are tied together, the lane group used to access the memory depends on the location of the memory syllable within the bundle, while allowing multiple memory operations to be executed simultaneously.

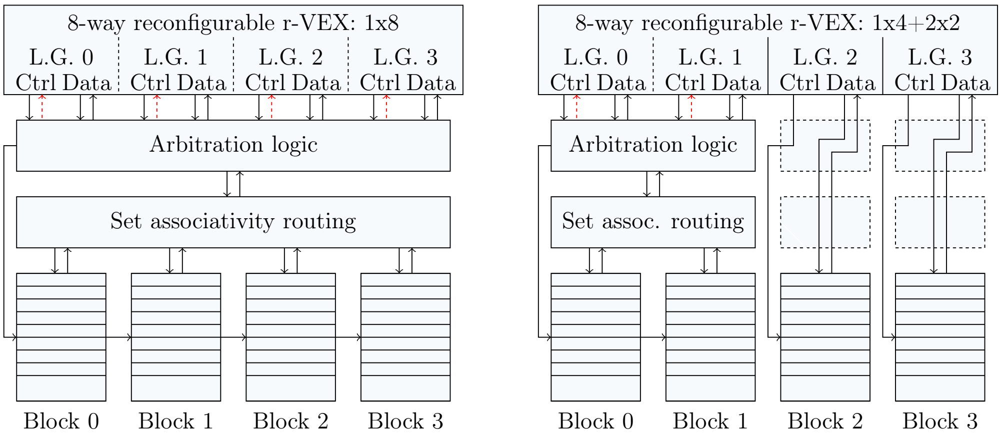
As we did with the instruction cache, we will start with the trivial design, where each lane group is simply connected to its own data cache. This does not pose the same problem as what we have seen with the instruction cache, as the alignment of the accessed addresses is not a function of the configuration. Still, both widening and shrinking a context through reconfiguration is likely to incur a cache penalty, similar to the trivial instruction cache solution. When a context is widened, the chance that a memory location that was in the cache can still be accessed is one half, depending on what the index of the memory syllable within the bundle is, and what the previous configuration was. Similarly, when a context is shrunk, half of the cache will be lost.

More importantly though, when a context runs on two or more lane groups and the cache uses a write buffer, cache consistency problems arise. Consider the scenario where a value is written to some address using cache block zero, and subsequently read using cache block one, because the store and load syllables were put in different places within the bundles. If block one has a copy of this address, the read will not go to to memory. Meanwhile, block zero may still have the write request waiting in the buffer. Even if cache coherence is implemented, the data in block one may not have been invalidated yet, as the write has not actually gone to memory yet.

Using a similar organization as what was used in the instruction cache solves these problems. However, if this is done, multiple writes per cycle per context cannot be supported. Recall that we did not have this problem with the instruction cache because each context always fetches a single aligned block of instructions. That is, even though the block gets larger as lane groups are combined, it still remains an access to a single cache line.

The problem could be solved by adding access ports to each data cache block. However, memories with more than two ports are expensive on the FPGAs that will be used, and as we will see in the next section, the tag memory already needs a secondary port in order to support cache coherence.

In the end, the choice was made to limit the number of accesses per context to one, resulting in the organization shown in Figure 3.11. The signal shown as a red dotted arrow from the arbitration logic back to the ρ-VEX is a fault signal, activated when simultaneous accesses are attempted.



(a) 1x8 mode.

(b) 1x4 + 2x2 mode.

Figure 3.11: Diagram illustrating the organization of the reconfigurable data cache. Instead of showing the generalized reconfigurable logic, the effective organization for two example configurations is shown.

The same simple replacement policy that was used for the instruction cache is also used for the data cache. This is nowhere near optimal. As with the instruction cache, implementing a more efficient replacement policy is left to future work.

3.6.3 Coherence and consistency

As per the project requirements, the cache must be coherent. This is achieved by means of bus snooping. Whenever a write access is requested on the bus by another master than the snooping cache block, a tag lookup is performed to see if a local copy of the written memory is stored in the block. If it is, that cache line is invalidated. In order to not unnecessarily stall read hits, these snooping tag lookups can be done in parallel to normal tag lookups. Both the instruction and data caches have a bus snooping interface, the former theoretically allowing self-modifying code without needing a full cache flush.

As bus snooping is used, every write must go to the bus. Thus, the cache must be write-through. In order to not incur the full memory access latency for each and every write, a write buffer is used. To keep the control logic simple, only a single write can be buffered at a time in each cache block, and only read hits can be serviced while a write is in progress. That is, a read miss does not attempt to preempt a buffered write to potentially improve performance, but has to wait for the write to be completed first.

Write misses are handled using the write allocate approach. This prevents the need to instantiate logic to check for buffered writes when doing a read, which would otherwise be necessary to guarantee cache consistency. This does pose a problem however. Consider the case where the ρ-VEX writes an 8-bit value to an address that is not currently cached. We need to store this value in the cache somehow, before the processor can be allowed to move on to the next instruction. This either requires a separate validity bit for each byte in the cache, or that the cache line must be fetched from memory before the write is committed to the cache line. The latter option is chosen for this design, because as we will see in Section 4.3.2, the validity bits are relatively expensive in terms of area.

If one of the blocks available to a context hits during a write, that block must be used to service the write. Otherwise, the block that hit would not be updated or invalidated by the write until it goes to memory and the bus snooping system invalidates it. During that time, it is undefined whether a subsequent read would return the old or the newly written value, violating cache consistency.

It is, however, possible for multiple blocks servicing a single context to have copies of a single piece of data. This situation can occur naturally after reconfiguring for instance from 2x2 to 1x4, as both contexts in the 2x2 mode may have been using the same memory location. This poses two potential problems.

The first potential problem is due to the write buffer again. If one of the original two contexts recently wrote to the address in question, its cache block will be updated accordingly immediately, but the other cache block is not invalidated until the buffered write goes to memory. If reconfiguration occurs and the new context reads from the address within this time, we end up with an undefined value as before. This problem is solved by delaying a reconfiguration until the write buffers of all affected lane groups are emptied.

The second potential problem occurs when the new context writes to the address. A write is normally only serviced by one block, thus the other block will, again, not be updated or invalidated until the write goes to memory. This problem is solved by defining that, if multiple blocks hit, the highest indexed block will always be used for both writes and reads. This ensures that, even while conflicting copies are stored in the cache, the correct copy is always used. Note that the actual rule used to select the block is arbitrary, it just needs to be consistent.

To improve the efficiency of the write buffers somewhat when multiple blocks are coupled together, a priority encoder is used to select which block is used to service and thus buffer the write. As we have seen, if one or more blocks hit during a write, the highest indexed block that hit must be used, thus the highest priority is awarded to those blocks. If all blocks missed, then priority is given to blocks that have an empty write buffer. If no block has an empty buffer at the time of the request, the first block to finish its write operation will be used. Due to these last two rules, the system behaves as if a single, deeper write buffer is implemented, as long as all the writes in question miss.

Peripheral access is supported by means of bypass accesses for the data cache only. In addition to bypass accesses not affecting the contents of the cache, the write buffer is bypassed. Thus, even for write operations, the processor is stalled until the bus operation is complete. This ensures that peripheral register accesses are executed in-order. In addition, it allows bus fault conditions to be forwarded to the processor. Bus faults are ignored otherwise.

3.6.4 Performance measurement features

As we have already illustrated in Section 3.4.5, it is desirable to be able to measure the performance of the cache in-system. For this reason, a number of status signals are returned to the processor along with the read data, to be used for tracing and/or to be wired to performance counter registers. These signals provide the following information per cache block/lane group.

- Whether or not an instruction fetch was serviced by this block, and if it hit or missed.
- Whether or not a data access was serviced by this block, and if it hit or missed.
- Whether the serviced data access (if any) was a read, a full line write or a partial line write.
- Whether the serviced data access (if any) bypassed the cache.
- Whether or not a write was buffered in this block at the time of the request.

In addition, each instruction and data block can be independently flushed using a control register. This allows programs to be restarted with an empty cache without needing to reprogram the FPGA bitstream.

3.7 Design-time configurability

In Section 3.2 we have already mentioned some design-time configuration options to make the system as flexible as possible, namely the number of implemented lanes, lane groups and contexts. In this section, we consider other useful configuration options.

3.7.1 Bundle organization

Let us start with the way in which the ρ-VEX expects bundles to be organized in memory. We define a configuration key that specifies the minimum alignment that the ρ-VEX may expect from a bundle. This indirectly specifies two things.

Firstly, if it is less than the issue width of the processor, variable-length instruction support logic must be instantiated to account for the relaxed alignment. If the specified alignment is greater than or equal to the issue width, such logic is not needed. Thus, this key allows the existence and complexity of the variable-length instruction logic to be specified.

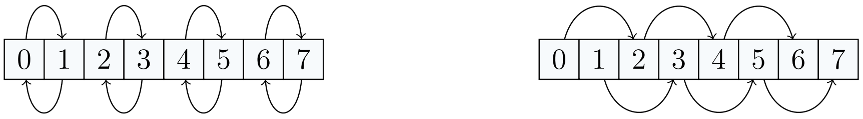
Secondly, we have seen in Section 3.3.5 that long immediate prefetching does not need to be done if the return address is known to be the start of a bundle. When variable-length instructions are not supported, this can be determined based on the alignment of the return address and this configuration parameter.

3.7.2 Functional unit resources

The old ρ-VEX processor allowed the hardware designer to specify which lanes should include a multiplication unit and which should not. We can do the same thing in this design.

Furthermore, the the lanes in which memory and branch operations differ among the various old ρ-VEX versions. To maintain as much compatibility as possible, we can allow the indexes of these lanes within lane groups to be specified.

Finally, we need to specify which lane allow long immediates, and to which lane they can be forwarded. We define that long immediate syllables can reside in the other syllable within an aligned syllable pair, or within the preceding syllable pair within a bundle, as shown in Figure 3.12. The logic for each of these options can be enabled or disabled as part of the configuration. To save branch unit complexity, the latter option is not supported in combination with variable-length instruction support.



(a) From neighbor within an aligned syllable pair.

(b) From previous pair.

Figure 3.12: Diagram illustrating which lanes can forward long immediates to which lanes in the two supported modes. Both modes can be supported simultaneously.

3.7.3 Pipeline and forwarding

A subject that we have avoided thus far is what the pipeline of the new processor looks like, and to what degree forwarding is implemented. This is because there is no single pipeline design. Instead of fixing the pipeline stages in which the various computational blocks perform their computations, we will keep these blocks as small as possible and allow the stages to be specified as configuration parameters (under certain conditions, of course). The way in which this is done is left entirely to Chapter 4.

3.7.4 Debug resources

There may be situations in which a minimal ρ-VEX implementation is desired. Perhaps the least important feature in such a minimal implementation is debug support. In particular, the trace unit requires a lot of routing resources for gathering all the traceable information. Thus, instantiation of the trace unit is controllable using a configuration parameter. To further save hardware, the number of hardware breakpoints is made configurable up to a maximum of four.

3.7.5 Opcode decoding

A key feature of the original VEX instruction set architecture (ISA) and toolchain is that it supports custom instructions. Although it is difficult to specify additional functional unit implementations as part of the configuration, making the opcode decoding configurable is relatively easy. This makes the opcodes for existing instructions configurable, and also allows support for any set of instructions to be disabled. In the latter case, the synthesis tools may then prune unused logic that only deals with the disabled instructions. In addition, new instructions may be specified merely through configuration, as long as these instructions can be handled by the existing datapath.

Opcodes are specified using a configuration file that is used to generate both the processor hardware description and the user manual. Thus, if the instruction set is modified, the documentation will automatically update. It is also used by the simulation model to output which instruction is being executed in human-readable form, and by the conformance test suite to assemble the test cases. It may also be used to generate the instruction set definition for the GNU binutils port, but such a generator is left as future work.

3.7.6 Traps

Similar to the opcode decoding, the trap cause identifiers of existing traps may be changed. It is also possible to select which of the undefined trap causes, which may be generated using the TRAP instruction, are handled as debug traps versus normal traps.

As was the case for the opcode decoding configuration, the trap cause configuration is specified with a configuration file that can generate a multitude of output files. Specifically, these are the hardware description files, the user manual, and C/assembly header files to facilitate software trap decoding, and human-readable trap descriptions in the simulation model.

The precise trap logic cannot currently be disabled or made imprecise to save logic resources in systems that do not need this feature. This would take considerable effort, as the trap logic is tightly woven into the pipeline and branch logic.

3.7.7 Control registers

Finally, the control register logic may also be generated from a configuration file. This is useful as control registers are frequently added and removed to test features inside or outside of the processor. Aside from just specifying the control register offsets, the logic of each control register is also specified, using a custom C-like programming language that can generate both VHDL and C code. The C code output is intended for a simulator, although this has not yet been developed.

The control register configuration file can be used to generate the hardware description, the user manual, C/assembly header files, and the aforementioned simulator sources.

3.8 Conclusion

In this chapter, we have discussed the design of the new ρ-VEX processor.

We first looked at how the control registers for the various features of the ρ-VEX could best be implemented in Section 3.1. A distinction was made between context control registers and global control registers, the former being local to a thread and the latter being shared between all threads. We also discussed how the registers can be accessed by means of the debug port.

Next, in Section 3.2, we designed the runtime reconfiguration system. The description of a configuration was abstracted to a mapping from lane groups (the computational resources) to contexts (the register files and control logic), encoded in a configuration word. We discussed in what ways a reconfiguration can be requested, how the configuration word is subsequently decoded in hardware, and how the hardware synchronizes the hardware resources before committing the new configuration.

The precise trap system is designed in Section 3.3. Here, we decided that all traps are handled by the same set of trap handlers regardless of the cause of the trap, and that traps would instead be identified by means of a trap cause and a trap argument context control register. We distinguished between the regular trap handler and the panic handler to deal with non-maskable nested traps. We specified an interrupt enable flag to prevent interrupts from unnecessarily being handled by the panic handler. Finally, we illustrate potential problems with returning from a trap handler after reconfiguration, and how these problems are solved.

In Section 3.4, the debug system is designed. Here, we decided that both self-hosted and external debugging are to be supported, and determined how this can be implemented. We specified that a UART-based debug interface peripheral is needed, and that software needs to be written to interface with this. Finally, we illustrated the need for a trace unit and specified the required features.

Subsequently, in Section 3.5, we determined how variable-length instruction support is to be implemented. We specified that stop bits can be best used to encode the bundle boundaries, as the toolchain already supports this method. We then argued that it is preferable to hide the complexity of fetching misaligned bundle from the memory system by implementing an instruction buffer in the processor.

We then determined how the reconfigurable cache for the processor is best implemented in Section 3.6. We set the requirement that a thread can only perform one memory operation per cycle in order to be able to merge the caches belonging to separate lane groups, improving performance in configurations where multiple lane groups are coupled. We then discussed how transient cache performance after a reconfiguration can be optimized, although many optimizations are left to future work. Finally, we discussed cache coherence and consistency problems caused by runtime reconfigurations, and how these are be solved.

The above fulfills the project requirements set in Section 1.3. To accelerate future research, we specified additional requirements for design-time configurability in Section 3.7

that are relatively simple to implement during the initial implementation compared to adding them later.

Implementation 4

In the previous chapter, the ρ-VEX processor and cache were designed. In this chapter, we implement these designs.

We begin by selecting a hardware description language (HDL) to use for the implementation in the first section. In the second and third section respectively, the implementations of the processor and cache are discussed. In the fourth section, we discuss the implementation of two processing systems that can be used to connect the processor and cache to various on-chip memories and peripherals using the custom ρ-VEX bus architecture. In the fifth section we implement the debug UART peripheral, that allows these systems to be connected to a computer for debugging and experimentation. Finally, in the last section, we discuss the software that was written to interface with the debug UART.

4.1 Language choice and code style

A major decision to be made in any hardware implementation project is the HDL to use. In Section 2.1.3 we have discussed VHDL, Verilog and SystemC. VHDL and Verilog are very similar languages in nature, so the choice between them is mostly a matter of preference. Since Delft University of Technology teaches VHDL, it is the obvious choice.

SystemC, however, is very different. It is based on high-level synthesis (HLS) technology, intending to shift more of the work from the programmer to the systhesis toolchain. While this may be beneficial, HLS is still relatively new, with limited synthesis toolchain support. In particular, the now legacy Xilinx ISE design suite only supports VHDL and Verilog; support is only added in the new Xilinx Vivado design suite. At the same time, Vivado does not support 6-series field-programmable gate arrays (FPGAs) or older [15]. As the Computer Engineering lab currently only owns one 7-series based development board large enough to fit the full ρ-VEX processor (the VC707) that has to be shared between research projects, using SystemC would significantly hamper development. Therefore, VHDL will be used as the implementation language.

In VHDL, generics may be used to configure entities. Where generics are inadequate due to language constraints, scripts are created to generate the VHDL files. The latter should be avoided where possible, as it precludes the possibility of having differently configured ρ-VEX processors in a single FPGA design.

VHDL entity descriptions can become very large, because traditionally every interface signal is specified individually. One way to solve this is to define record types that contain all the signals that need to go from one entity to another, such that a single signal of that record type is sufficient. Unfortunately, this does not work very well with generics, as the contents of a record are not parameterizable. Consider, for instance, the decoded runtime configuration signals: the size of virtually all these signals depends on the design-time configuration. Thus, this usual approach is not applicable.

In an attempt to nevertheless organize the interface signals, we define a signal naming convention. Each major entity in the design is assigned an abbreviation consisting of lowercase letters. The name of any interface signal is prefixed with the abbreviation of the source entity, followed by a 2 (‘to’), followed by the destination entity. The separator between the prefix and the signal name is an underscore to make it easily visible. An example is `pl2br_opcode`, the branch unit opcode supplied by the lane. This allows a programmer to very quickly identify a signal. Furthermore, interface signals are to be grouped by function where possible.

4.2 ρ-VEX processor

In this section, we will discuss the implementation of the processor. Its structure is shown in Figure 4.1. The abbreviations used for each entity are listed alphabetically below.

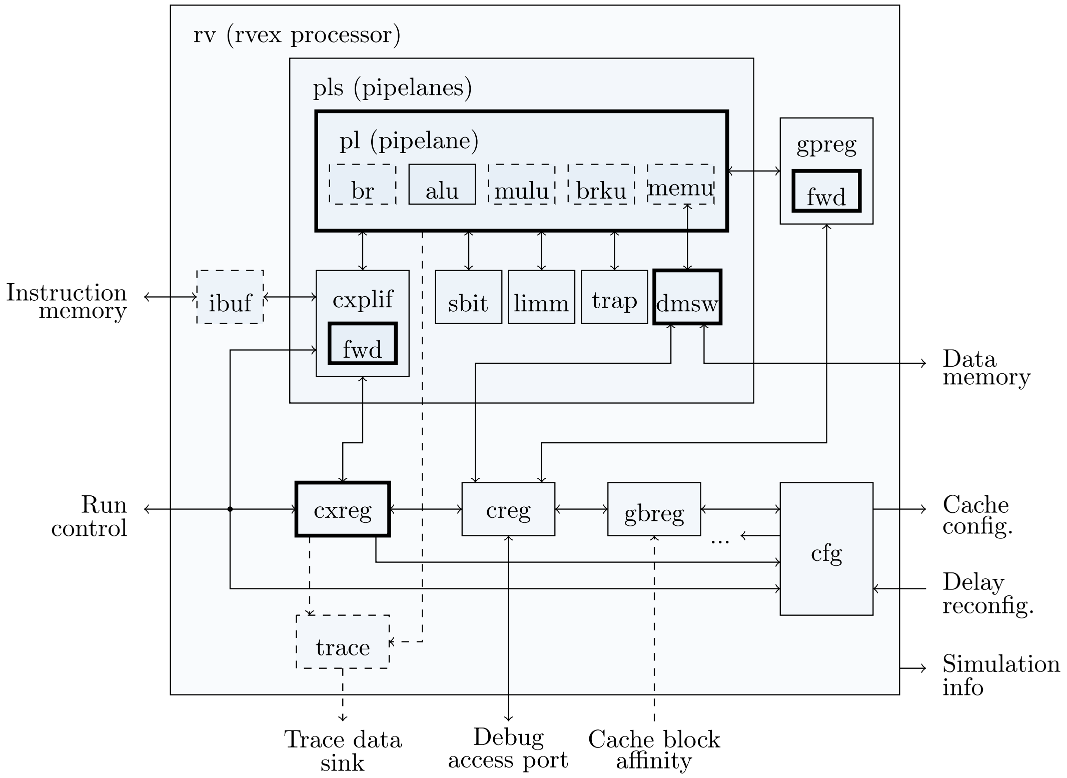


Figure 4.1: Structural overview of the ρ-VEX implemented processor. Each block represents a VHDL entity. Blocks with thick borders are instantiated multiple times, blocks with dotted borders may or may not be instantiated based on the configuration of the parent block.

alu Arithmetic logic unit. Handles ALU syllables.

br Branch unit. Handles branch syllables, selects the next program counter (PC), and controls instruction fetching.

brku Breakpoint unit. Generates a trap if a breakpoint or watchpoint is hit.

cfg Reconfiguration controller.

creg Control register bus routing logic.

cxplif Context-pipelane interface. Connects the lane-based resources with the context-based resources depending on the current configuration.

cxreg Context control register file.

dmsw Data memory switch. Multiplexes between memory and control register accesses, and optionally lengthens the latency of the control register access to match up with the configured memory latency.

fwd Forwarding logic.

gbreg Global control register file.

gpreg General-purpose register file.

ibuf Instruction buffer. Hides misaligned instruction fetches due to variable-length instructions from the instruction memory.

limm Long immediate routing logic.

memu Memory unit. Handles memory syllables.

mulu Multiplication unit. Handles multiply syllables.

pl Pipelane. Contains the datapath and pipeline logic for a single lane.

pls Pipelanes. Contains all the lanes.

rv ρ-VEX toplevel entity.

sbit Stop bit routing. Performs PC+1 selection, reroutes branch syllables to the highest indexed coupled lane, and invalidates lanes that have received syllables beyond the stop bit of the current bundle.

trace Trace unit. Gathers execution information and compresses it into trace packets.

trap Trap routing. Handles lane/pipeline stage invalidation if a trap occurs and ensures that the right trap information is forwarded to the branch unit and control registers in case multiple traps occur at the same time.

The following sections describe the processor in more detail. For the sake of coherence, the sections break down the processor in a functional rather than a structural way. We start with describing the reconfiguration-related logic in the first section. The second section describes the datapath and the way in which the configurable pipeline is realized. The remainder of the sections describe the remaining components of the processor in a roughly chronological order from an instruction execution point of view.

In all the following sections, there are many references to control registers. Fefer to Chapter 4 of the ρ-VEX user manual (Appendix C) for their specifications.

4.2.1 Reconfiguration

The reconfiguration system consists of two major parts: the controller (`cfg`) and the context-lane interface (`cxplif`). The task of the controller is to arbitrate between and decode incoming requests, and synchronize the running contexts that are affected by the reconfiguration before reconfiguring. The context-lane interface connects the lane and context resources together based on the current decoded configuration.

After a hard reset of the processor, the configuration is reset to 0. That is, the ρ-VEX behaves like a single-core processor until a reconfiguration is requested. This configuration does not need to be decoded at runtime; the decoded configuration registers are loaded with precomputed values during reset.

There are several sources that can request a reconfiguration:

- The `CRR` context control register to allow the threads running on the processor to request a reconfiguration.
- The `BCRR` global control register to allow the debug bus to request a reconfiguration.
- The wake-up system.

The wake-up system is part of the reconfiguration controller itself. It automatically requests a reconfiguration to the configuration word in the `WCFG` register if all of the following conditions are true:

- The `S` (sleep) flag in the `SAWC` register is set. This flag enables or disables the system.
- An interrupt is pending on context zero.
- Context zero is not already part of the current configuration.
- There is no reconfiguration in progress.

These conditions ensure that a wake-up is only requested once, and only after the thread that configured the wake-up system has successfully disabled.

In addition to the reconfiguration request, `WCFG` is set to the current configuration and the `S` flag in `SAWC` is cleared. `WCFG` (wake-up configuration) and `SAWC` (sleep-and-wake control) are context control registers that only exist for context zero. Refer to Section 6.3 of the ρ-VEX user manual (Appendix C) for more information.

Arbitration among the requests is performed using a priority encoder. The wake-up request has the highest priority, followed by the `CRR` requests by ascending context index, followed by the `BCRR` debug bus request. The index of the source that wins arbitration is stored in the `RID` (requester ID) field of the `GSR` global control register.

Configuration decoding When a reconfiguration is requested, the new configuration word must be checked for validity and decoded. Before anything else, a simple OR gate is used to check if any reserved bits are set. If so, reconfiguration is canceled and the error flag (`E` in `GSR`) is set until the next request clears it.

Decoding and further validation is done using a finite-state machine (FSM). This state machine must check whether the lane group mappings for each context are contiguous and aligned, as well as decode the following configuration signals:

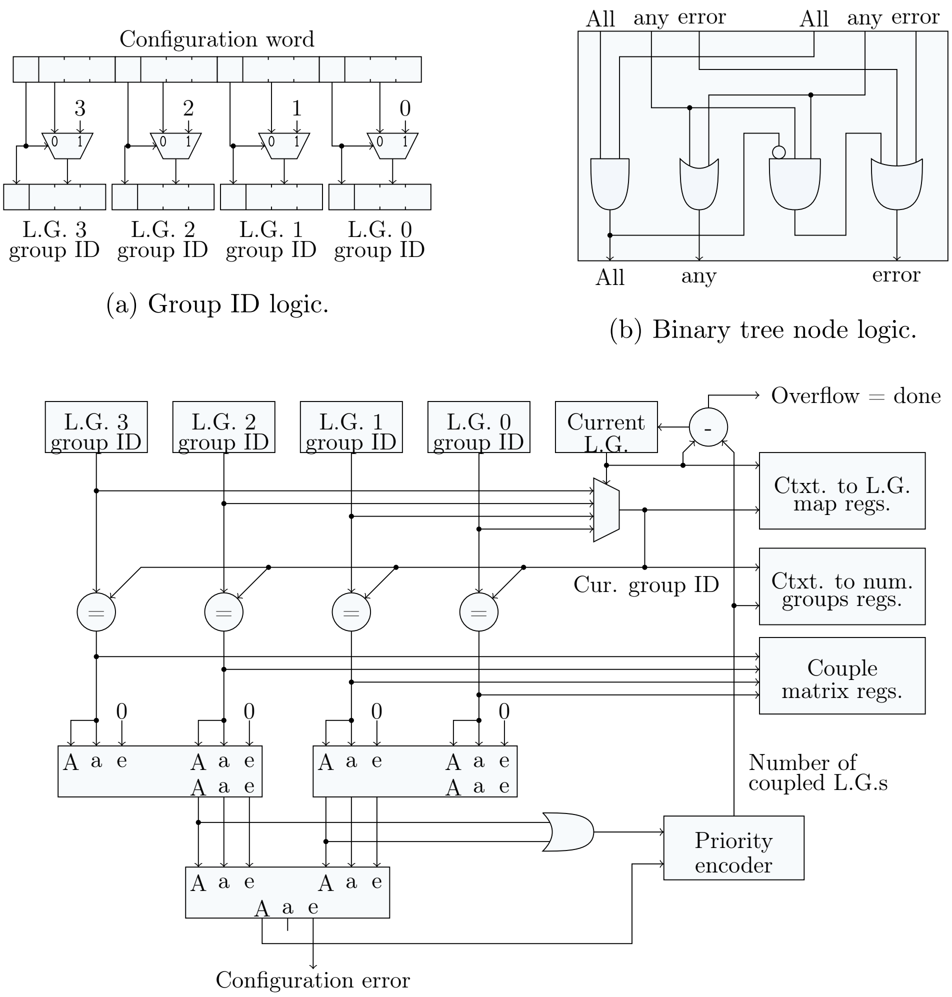
- For each context, whether the context is connected to any lane group and, if so, what the highest index of the connected groups is. The highest index is used because this is the lane group that contains the active branch unit for the context.
- For each context, the logarithm of the number of lane groups connected to it. This is used for generating PC+1 when variable-length instruction support is disabled at design time.
- A signal for each pair of lane groups specifying whether they share the same context or not. This is represented as a diagonal block matrix of size nxn, where n is the number of lane groups in the processor. It is called the couple matrix.

These signals are constructed while iterating over the lane groups in the configuration word in descending order. To that end, the FSM state consists of the index of the lane group that is currently being decoded, in addition to a 'busy' bit. The 'busy' bit is set when a new request is received, and cleared when decoding is complete or the requested configuration is found to be erroneous.

While the 'busy' bit is set, three other things happen. Firstly, the lane group index state is set to the number of lane groups in the system minus one. Secondly, the registers for the decoded configuration are reset to the state where all lane groups and contexts are disabled and decoupled. Thirdly, the configuration word is converted as shown in Figure 4.2a. This logic assigns a number to each lane group that we will refer to as a group identifier (ID). For enabled lane groups the group ID just maps to the context they should be connected to, while disabled lane groups each get a unique ID. This allows all lane groups sharing the same group ID to be coupled together. Observe that we cannot just use the configuration word nibbles for this directly, as configurations like `0x8880` or `0x8128` would then violate the alignment and contiguity requirements.

In the 'busy' state, the logic shown in Figure 4.2c is active. The first part of it determines the group ID of the current lane group. If it represents a context, the 'enable' configuration register for that context is set, and its 'highest indexed connected lane group' configuration register is set to the current lane group.

Next, the current group ID is compared against all the other group IDs. The resulting vector represents all the lane groups that need to be coupled together. Thus, this vector



(a) Group ID logic.

(b) Binary tree node logic.

(c) Main decoding logic.

Figure 4.2: Configuration word decoding logic for four lane groups.

can be fed to the couple matrix configuration registers, such that every matrix index (i, j) is set if i and j are both set in the vector.

The comparison outputs are also fed to a binary tree network. This network checks whether the current set of coupled lane groups is aligned and contiguous. The logic contained within each tree node is shown in Figure 4.2b. Each node outputs whether all of the child nodes are coupled with the current lane group, whether any of them are, and whether an error has been found yet. The error output is enabled when both immediate child nodes have *some* nodes that are coupled, but not *all* of the child nodes are coupled.

The final output we need is the logarithm of the number of coupled lane groups, as decoded configuration output and for determining by how many lane groups to advance. This is done by means of a priority encoder connected to the 'all' signals within the binary tree: the last level of the tree where one of the 'all' signals is on determines the number of coupled lane groups.

Decoding terminates with an error as soon as the 'configuration error' signal activates. If it does not activate, decoding terminates successfully when the current lane group counter is back at its initial value. In either case, the 'busy' state is cleared.

Synchronization When the decoder successfully finishes processing the requested configuration, the decoded configuration for each context (enabled or not, number of lane groups connected, index of the last lane group) is compared with the current configuration. Any context for which the configurations differ is affected by the reconfiguration, and must be halted before the new configuration can be committed. To do so, the 'run' signal to the branch units of those contexts is forced low, causing the branch units to stop fetching new instructions.

The reconfiguration controller then waits until all the affected pipelines are idle, and until the affected cache blocks report that their write buffers are empty. Only then, finally, is the new configuration committed.

Performance It is difficult to determine the performance of the reconfiguration system exactly, because it depends on stalls from the memory subsystem, the state of the cache write buffers, the requested configuration, its difference between the requested and current configuration, and the pipeline configuration. As an indication though, let us evaluate the timing of reconfiguring from `0x0123` to `0x3210` in the absence of any memory or

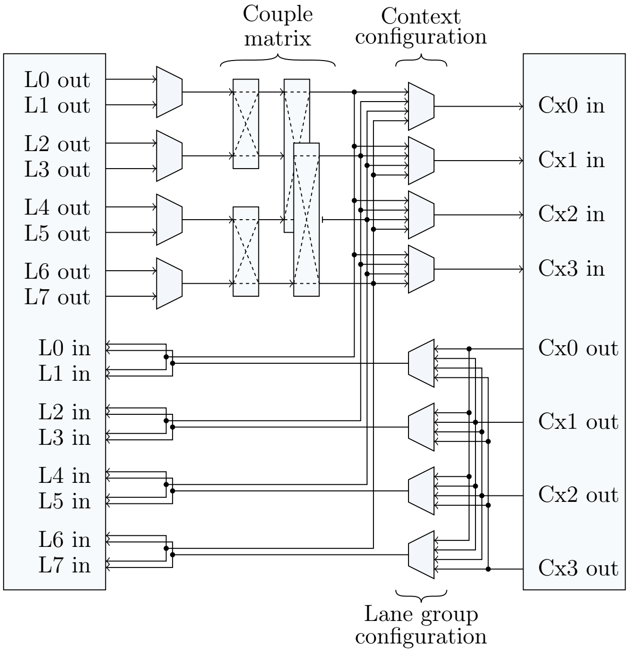


Figure 4.3: Context-lane interface structure for an eight-way, four-context, four lane-group ρ-VEX. The lanes are represented on the left, the contexts on the right.

cache stalls. This is the worst case performance in terms of decoding delay and affected contexts.

Let us define the issue of the instruction that requests the reconfiguration to be $t = 0$, and let t be measured in clock periods. The write to the control register is then made in $t = 2$ due to the pipeline delay from `IF` to `EX1`. There is an additional cycle latency in the control register logic, which means that the reconfiguration controller receives the request at $t = 3$. Thus, the first decoding cycle starts at $t = 4$. `0x3210` contains four group IDs, so decoding is complete at $t = 8$. At this point, the 'run' signal to the contexts is released, causing them to flush their pipelines. This completes at $t = 12$, as the ρ-VEX has four pipeline stages. Thus, the new configuration is activated at $t = 13$, and the first instruction in this configuration is issued at $t = 14$.

This means that the latency from the issue of the instruction that requests a reconfiguration to the issue of the first instruction in the new configuration is 14 clock cycles in this scenario. However, this value is only relevant for the context that requests the reconfiguration. The other contexts only experience a 6-cycle latency from $t = 8$ to $t = 14$, primarily due to the required pipeline flush.

The performance is tested in practice in Section 5.5.

Context-lane interface The context-lane interface handles the vast majority of the reconfigurable interconnect between the lanes and the contexts. Its structure is shown in Figure 4.3. The couple matrix first controls a network that merges the data from coupled lane groups together. Parts of this data, such as the PC, are broadcast back to the lane groups. The data is also forwarded to the context logic, through a set of multiplexers controlled by the last coupled lane group for each context. Finally, data from the contexts can also be forwarded to the lane groups, through a set of multiplexers controlled by the lane group to context mapping.

4.2.2 Datapath and pipeline

As specified in Section 3.7.3, the ρ-VEX has a configurable pipeline. This is accomplished by dividing the datapath up into several blocks, and specifying the pipeline stage in which each of those blocks is placed using constants. These blocks and the default pipeline configuration is shown in Figure 4.4. The pipeline can be regarded as having four or five stages, depending on whether the general-purpose register write delay is considered to be part of it. The cause of this delay is explained in Section 4.2.10.

The configurable pipeline is implemented using two-process methodology; that is, one process describes just the inter-stage registers, and the other describes all the elements of the datapath as shown in the figure. To understand how it is made configurable, first consider a simplified datapath with three combinatorial computational blocks and a (stage) register before the input of the first block and after the output of the last block. This can be written as follows:

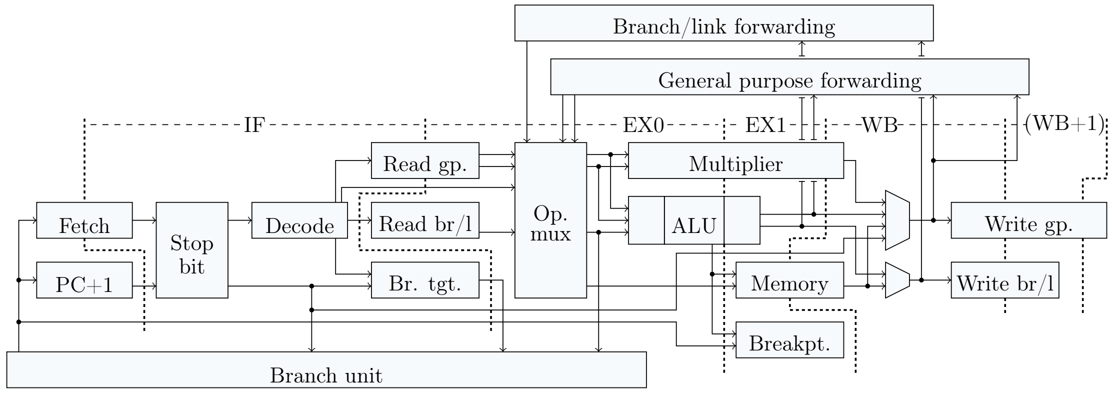


Figure 4.4: Overview of the ρ-VEX datapath and default pipeline configuration.

```
comb: process(r1) is
  variable s: state_type;
begin
  s := r;
  s := block_a(s);
  s := block_b(s);
  s := block_c(s);
  d <= s;
end process;
```

```
reg: process(clk) is
begin
  if rising_edge(clk) then
    r <= input;
    output <= d;
  end if;
end process;
```

Now, to insert an additional pipeline stage between block A and block B, we could rewrite the code as follows:

```
comb: process(r1) is
  variable s1, s2: state_type;
begin
  s1 := r1;
  s1 := block_a(s1);
  d1 <= s1;
  s2 := r2;
  s2 := block_b(s2);
  s2 := block_c(s2);
  d2 <= s2;
end process;
```

```
reg: process(clk) is
begin
  if rising_edge(clk) then
    r1 <= input;
    r2 <= d1;
    output <= d2;
  end if;
end process;
```

Notice that the numbers that we have appended to the variable names can also be array indices. Also notice that, since `s1` and `s2` are independent, the variable initialization can be moved to the start of the process, and the signal assignment to the end. This leads to the following code:

```
comb: process(r) is
  variable s: state_array(1 to NUM_STAGES);
begin
  s := r;
  s(1) := block_a(s(1));
  s(2) := block_b(s(2));
  s(2) := block_c(s(2));
  d <= s;
end process;
```

```
reg: process(clk) is
begin
  if rising_edge(clk) then
    r(1) <= input;
    r(2 to NUM_STAGES) <= d(1 to NUM_STAGES-1);
    output <= d(NUM_STAGES);
  end if;
end process;
```

In this code, the pipeline configuration is fully specified by the array indices used and the constant `NUM_STAGES`. For instance, to go back to the original situation, we simply replace all indices with `1` and set `NUM_STAGES` to `1`, or to also insert a stage between block B and C, we set `NUM_STAGES` to `3` and replace the index for block C to `3`. Thus, we can replace these numbers with configuration parameters. This is exactly what is done in the ρ-VEX pipeline, though on a larger scale.

Of course, it is possible to misconfigure this pipeline description by for example assigning block A to stage 2 and block B to stage 1. In order to detect such misconfigurations at synthesis, assertion statements are used.

Some of the ρ-VEX blocks also have a configurable latency. The memory block is an example. To support this, the latency of the block is simply added to the index for all writes to `s`.

4.2.3 Instruction fetch

The instruction fetch system is in charge of requesting instructions from the instruction memory or cache, and routing the fetched syllables to the appropriate lanes.

This is relatively trivial when variable-length instruction support is disabled. In this case, exactly one instruction fetch is needed per cycle, the fetched data makes up exactly one bundle, and the syllables can be routed to the lanes one on one. This is, in fact, not even dependent on the runtime configuration. The only complication is generating the fetch address for each lane group from the PCs when multiple multiple lane groups are coupled.

First of all, the PCs must then be appropriately aligned to handle the cases listed in Section 3.3.5. Secondly, lane groups that take syllables that come before the misaligned PC must be disabled. Finally, lane group must give the appropriate address. To illustrate the latter, consider two coupled lane groups in an ρ-VEX with two lanes per group, fetching the bundle at `0x00`. The first lane group must then indeed request address `0x00`, but the second lane group needs to request `0x08`, as `0x00..0x0F` needs to be fetched.

All these things can, fortunately, be handled trivially in the context-lane interface. The operations that need to be performed on the PC all come down to overriding some bits in the PC. Lane invalidation is done simply by comparing the actual PC with the fetch addresses. This requires only two-bit comparators when there are four lane groups, as only the bits that are actually modified need to be compared.

Things get significantly more complicated when variable-length instruction support is required. As described in Section 3.5.2, a register and shifter is needed to store the previous instruction fetch result, and as described in Section 3.5.3, syllable distribution is no longer trivial.

Instruction buffer Figure 3.7b in the design chapter shows an overview of the logic needed to handle misaligned instruction fetches. This piece of logic is called the instruction buffer[1]. Figure 4.5 shows how this logic is implemented for an ρ-VEX with four lanes, two lane groups and no bundle alignment requirements.

[1] Not to be confused with the instruction buffer of a superscalar processor. The name is simply chosen because it buffers an instruction fetch.

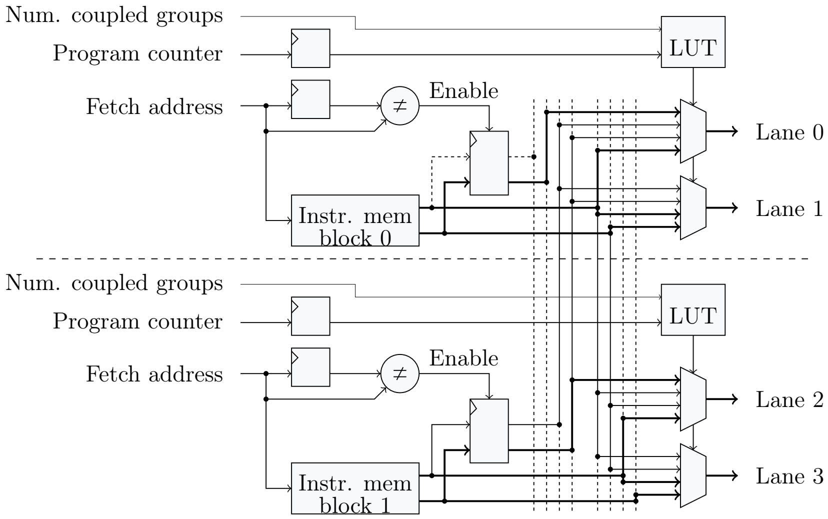


Figure 4.5: Instruction buffer logic for a four-way ρ-VEX with two lane groups and no bundle alignment requirements.

Let us start at the left of the diagram and work our way right. First, notice that we are still using the fetch address aligned by the context-lane interface to handle misaligned PCs after returning from a trap in the fixed-length instruction case. After all, the memory interface still has to comply with those alignment rules, as we have decided in the design that the ρ-VEX should hide all variable-length instruction complexity from the memory system.

However, instead of this address always needing to be aligned downwards, it needs to be aligned upward after the first fetch after a branch, as we normally have to fetch the second part of a misaligned bundle. This upward-aligning requires a full-width adder, as shown in the top-left corner of Figure 3.7b. This addition is done in parallel to the PC+1 computation, as discussed in Section 4.2.6.

To handle the special case where we *do* need to align downwards, i.e., when branching to a misaligned address, the branch unit will feed the normal PC+1 signal as fetch address to the context-lane interface when such a branch occurs, in addition to doing a double fetch. Refer to Section 4.2.6 for more information about this.

In addition to routing the fetch address to the instruction memory system, we also monitor when it changes, because we only want to save the most recently fetched line in the buffer register when a new line is being requested. Otherwise, we would end up overwriting the previous fetch with the result of the same fetch that we are currently doing[2].

Note that the not-equal comparator and register that checks for changes in the fetch

[2] Note that this implies that we could also be using this as a read enable signal for the instruction memory. This is not currently implemented as neither the cache or the on-chip memory system incur a penalty for requesting the same address in subsequent cycles, and it makes the one-bit optimization more complex.

address do not actually need to be and are not the full width of the address. It is in fact sufficient to only check the bit that immediately follows the least significant bits that are constant due to the alignment requirements, as proven below.

- When not branching: regardless of the length of the previous bundle, we will never move forward by more than one line at a time, as bundles can never be larger than a line.
- When branching to an aligned address: the buffer has a fifty percent chance of being loaded, but the loaded value is never used, as the address was aligned and the fetch result can immediately be forwarded to the lanes. The checked bit of the fetch address will necessarily change in the cycle immediately following the branch, overriding the undefined value.
- When branching to a misaligned address: the buffer has a fifty percent chance of being loaded during the first cycle of the double fetch. However, the fetch address is necessarily different in the second cycle, thus overriding the undefined value before it is used.

This leaves only the lookup table (LUT) block and the multiplexers. The thick syllable paths represent those that are needed for 2x2 modes, the rest of the solid paths are needed in addition to those in 1x4 mode. The dotted paths are never used, and are only shown to make the recurring structure more clearly visible. Which paths are chosen depends on the least significant bits of the PC (up to the alignment boundary) and on the runtime configuration, which is uniquely specified for any lane group by the number of lane groups connected to it. As this is only five bits for the standard reconfigurable ρ-VEX configuration, the decoder is simply implemented as an FPGA lookup table (LUT).

Syllable routing By default, syllables are routed to the lanes one to one. With variable-length instruction support enabled, additional logic is instantiated to route the last syllable in the bundle to the last physical lanes and to disable lanes that did not receive a syllable from the current bundle. This logic is marked as 'stop bit' in the datapath (Figure 4.4). The routing logic is shown for a single two-lane lane group in Figure 4.6. It is simply repeated for each lane group.

4.2.4 Instruction decode

The decode block is responsible for decoding syllables into control signals for the functional units and datapath. It also routes long immediate information to the appropriate syllables. If illegal or unknown syllables are encountered, traps are generated.

Syllable format All control signals of an ρ-VEX syllable can be derived from the most significant byte, bit 23 and bit 1. The most significant byte is the opcode field, bit 23 directly switches between an immediate or general-purpose register for the second operand, and bit 1 is the stop bit. Bit 0 is intended to be used to mark cluster boundaries, which are not supported by the current ρ-VEX implementation, thus bit 0 is completely

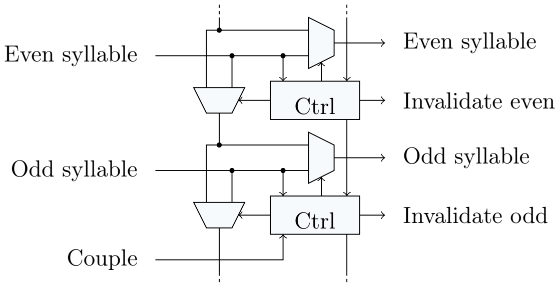


Figure 4.6: Variable-length instruction syllable rerouting logic for a single lane group that has two lanes.

unused. The remainder of the bits is used to encode immediates and register indices in a way that depends on the 8-bit opcode and bit 23. For more information, refer to Section 3.7 of the ρ-VEX user manual (Appendix C).

All control signals are decoded from the opcode by means of a table lookup. This table is generated by the opcode configuration scripts, thereby making the instruction set configurable as specified in Section 3.7.5.

Long immediates Any non-branch ρ-VEX syllable that needs an immediate operand less than -256 or greater than 255 requires a long immediate syllable (`limmh`). A `limmh` syllable consists of an abbreviated opcode, three bits that designate the target syllable index within an eight-way bundle, and 23 immediate bits to extend the 9 bits that are already available in the target syllable to get a full 32-bit word.

As all non-branch syllables are mapped to lanes directly, the target syllable index can also be regarded as a lane index within a set of coupled lane groups. This only holds, however, when a complete bundle is executed at a time, which is not the case when running an eight-way generic binary in two-way or four-way mode. Thus, a naive implementation supporting this would need to keep track of when the previous bundle ended.

The implemented hardware avoids this by only implementing two routes through which a long immediate syllable can forward its immediate extension to another syllable. These routes are shown in Figure 3.12. Observe that the intended route can be determined by only comparing the least significant bit (LSB) of the `limmh` lane index with the LSB of the target syllable index. Because the number of syllables already executed since the end of the previous bundle is always a factor of two, the LSB of a syllable index always equals that of the lane index. Thus, we can just ignore the remaining index bits entirely.

As an ρ-VEX lane group always consists of an even number of lanes, the '`limmh`-from-neighbor' route can be trivially implemented. The '`limmh`-from-previous' is however more complicated. Depending on the runtime configuration, the previous syllable pair may be executed in parallel in the previous lane pair, or it may have been issued in the last lane pair of a set of coupled lane groups in the previous cycle. This requires logic as shown in Figure 4.7. It also requires that the syllable pair immediately before the trap point be fetched before returning from a trap, as explained in Section 3.3.5.

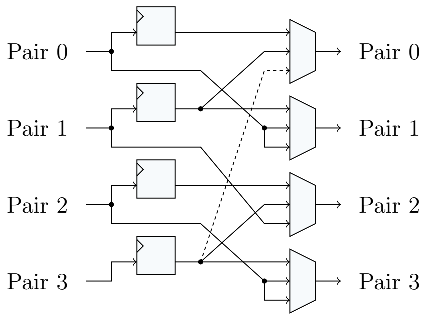


Figure 4.7: Routing logic needed to support '`limmh`-from-previous' forwarding for an eight-way ρ-VEX with four lane groups. The dashed path is only needed for generic bundles larger than eight syllables.

Each of the two forwarding methods can be enabled or disabled at design time to save logic. Note that if both are enabled at the same time, it is possible for a syllable to receive two long immediates. This is an illegal condition that results in a trap. A trap is also generated if a `limmh` syllable attempts to forward to a syllable that does not use an immediate.

4.2.5 Operand read

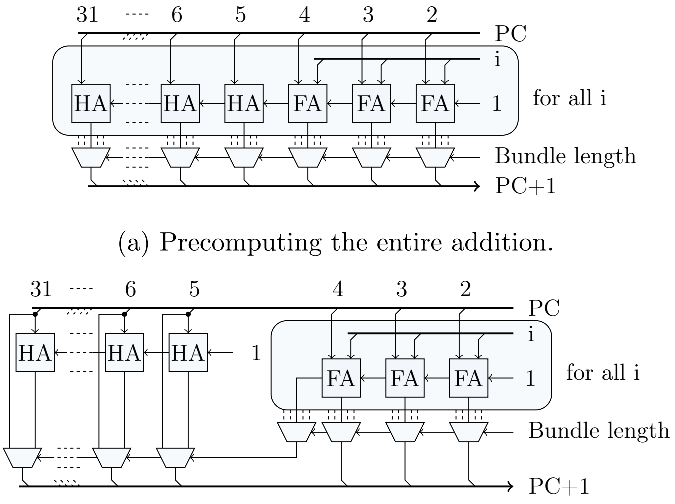
The ρ-VEX allows two general-purpose registers, the link register and all branch registers to be read per lane per cycle. This data, along with the normal or long immediate, is the multiplexed into three integer operands and a branch operand. Three operands are needed in particular for memory store operations, which use two integer operands for determining the address and one for the data.

Forwarding can be enabled or disabled at design time. In the default pipeline configuration, data can be forwarded to the register read stage (EX0) from all subsequent stages. It is also possible to enable forwarding to multiple stages. For example, if forwarding to EX1 is also enabled, the memory write data can be forwarded directly from a preceding multiply or memory load instruction. If this would be done, the operand multiplexing logic would be instantiated in both EX0 and EX1. The EX1 multiplexers would then override the operands determined in EX0, to allow any functional unit that starts in EX1 to use the more recent forwarding data.

4.2.6 PC+1 computation

When variable-length instruction support is disabled, the PC+1 block consists of a single adder for each lane group, that adds the number of coupled lanes to the current PC. Approximately the same thing could be done with variable-length instructions as well, but as the value added to the PC then depends on the result of the instruction fetch, we would have to fit the entire instruction fetch datapath plus a 30-bit adder in a single cycle.

The adder can be taken out of the path by precomputing PC+1 for all possible bundle lengths, and then multiplexing between the results when the bundle length is known, as shown in Figure 4.8a. However, this many parallel adders is quite area intensive. We can optimize this further by using the fact that the number that we add is only a few bits wide. This allows us to modify the circuit to what is shown in Figure 4.8b.



(a) Precomputing the entire addition.

(b) Precomputing only what is needed.

Figure 4.8: Adder structures for computing PC+1 with variable-length instructions that allow the addition to be done before the bundle length is known. The shaded areas are instantiated for all possible bundle lengths.

Recall from Section 4.2.3 that the fetch address differs from the actual PC with variable-length instructions. Specifically, the fetch address is the actual PC either rounded down or rounded up to the next alignment point. Rounding down is handled in the context-lane interface by replacing bits. To round up, we need to add the number of coupled lanes minus one to the PC before replacing the bits. This extra addition can be done in parallel to the PC+1 computation. In fact, the wide adder at the top left of Figure 4.8b can be shared between the normal and fetch PC computation.

A special case arises during the second half of a double fetch. During this cycle, the syllables coming from the instruction buffer are undefined, thus the bundle length is also undefined. To still allow the branch unit to use the computed fetch address during this cycle, the bundle length is forced to zero whenever no valid syllables have been fetched.

4.2.7 PC-relative branch target

All conditional branch instructions use a 19-bit immediate to specify the branch target. This immediate is interpreted as a signed integer and added to the PC+1 value to get the actual target. This adder is marked as 'br. tgt.' in the datapath (Figure 4.4).

It is worth noting that it is no accident that PC+1 is used as the base address instead of PC. This has to do with generic binaries. In a generic binary, branch syllables must always be the last in the bundle. Thus, PC+1 necessarily always points to the next bundle when the branch syllable is executed. The current PC, on the other hand, depends on whether the generic binary is executed using two, four or eight lanes.

The weight of the LSB of the immediate is design-time configurable to either a syllable or a syllable pair. The latter is the default, as this is what the toolchain assumes. In order to support variable-length instructions down to the syllable level though, either the immediate must be syllable-based, or all bundles that are to be branched to must be artificially aligned by appending a no-operation (NOP) syllable to the previous instruction if necessary.

4.2.8 Branch unit

The branch unit is responsible for determining what operation is performed next. To accomplish this, it outputs the next PC, next fetch address (which may differ from the PC, as we have seen in Section 4.2.3), the read enable signal for the instruction memory, and various invalidation signals for the next instruction.

The branch unit determines the next operation by checking for the following conditions. The first one that is true determines the performed operation.

1. *The 'jump' flag in the debug control register is set*

   While a program is running, the current PC is taken from the pipeline stage registers, not from the control register. Thus, if the PC control register is written by the debugger, the written value would be ignored and overwritten in the next cycle. To solve this, the 'jump' flag in the debug control register is set when the debugger writes to the PC, activating this case, wherein the next PC is forced to the newly written PC. The 'jump' flag is automatically cleared whenever an instruction is fetched.

2. *A trap other than `STOP` is to be handled, but condition 8a, 8b or 8c is also true, so the context should halt*

   In this case, the trap is deferred to when the context is restarted by setting the next PC to the instruction that caused and/or was interrupted by the trap. Thus, when the processor is restarted, the trap would most likely be generated again, unless the condition that caused the trap was solved while the context was halted.

3a. *The `STOP` trap is to be handled*

3b. *A debug trap is to be handled, and the context is in external debug mode*

   In this case, the 'break' flag in the debug control register is set, and the next PC is set to the instruction that caused and/or was interrupted by the trap. If the to-be-handled trap is a `STOP` trap, the 'done' flag is also set, and the PC is actually set to the bundle immediately following the bundle with the `STOP` syllable (see also condition 6).

4a. *A debug trap is to be handled, and the context in self-hosted debug mode*

4b. *A fault trap is to be handled*

4c. *An interrupt trap is to be handled*

In this case, the next PC is set to the trap handler or panic handler address, depending on the state of the ready-for-trap flag (Section 3.3.3). The trap information is forwarded to the appropriate context control registers, and the states of the ready-for-trap, interrupt enable and breakpoint enable flags are backed up and cleared. In case 4c only, the trap argument is overridden by the interrupt identification signal from the interrupt controller, and the interrupt is acknowledged.

5. *The `RFI` (return from trap) branch syllable is to be handled*

In this case, the next PC is set to the value in the trap point register. In addition, the pipeline is set up such that the ready-for-trap, interrupt enable and breakpoint enable flags are restored in the stage in which the memory is accessed (EX1 in the default pipeline configuration).

If the ρ-VEX is configured to allow long immediate syllables to forward to the next syllable pair, and the trap point is not aligned to a generic binary bundle boundary, a long immediate prefetch (Section 3.3.5) is performed. That is, the next PC is set to the trap point minus the current instruction fetch size. Notice that the subtractor that does this does not need to be the full width of the PC because it will never cross a bundle alignment boundary.

Notice that the above only makes sense if variable-length instruction support is disabled. Otherwise, it is not clear from just the address if a long immediate prefetch is needed, because any address could be mid-bundle. Thus, a prefetch would always need to be performed, which also means that the subtractor would need to be full-width. Because of this, long immediates that forward to the next syllable pair are not supported when variable-length instruction support is enabled.

If the trap that is being returned from was a debug trap, debug traps are masked for the following bundle. This allows the self-hosted debugger to resume execution after the breakpoint or watchpoint that caused the trap without having to remove it, and also allows single-stepping to be performed.

6. *The `STOP` syllable is to be handled*

In this case, the branch unit will generate a `STOP` trap in the *next* cycle. Thus, the program is interrupted between the bundle that contains the `STOP` syllable and the next.

7a. *An unconditional branch syllable is to be handled*

7b. *A conditional branch syllable is to be handled, and the branch condition is true*

In this case, the next PC is set to the branch target.

8a. *The 'break' flag in the debug control register is set*

8b. *The 'run' input signal for this context is low*

8c. *The reconfiguration controller is requesting that the context is paused to synchronize for reconfiguration*

8d. *A trap has occurred, but has not yet been forwarded to be handled, because earlier instructions may still cause traps*

In this case, the next PC is forced to the current PC to halt execution. The read enable signal to the instruction memory is turned off, and the next instruction is invalidated.

9a. *Condition 8a, 8b or 8c were true in the previous cycle*

9b. *The processor was reset in the previous cycle*

This means that the program is (re)starting execution. The next PC is set to the value stored in the PC control register.

10. *None of the above*

The next PC is set to PC+1 for normal program flow.

Whenever a value other than PC+1 is selected for the next PC, all pipeline stages before the branch stage are invalidated.

As we have seen in Section 4.2.3, the next PC differs from the next instruction fetch address if variable-length instruction support is enabled. In addition, when branching to a misaligned address, two instruction fetches must be performed to fetch the first bundle. Most of the logic needed for this has already been specified in Section 4.2.3 and 4.2.6. The branch unit ties everything together using logic that follows the following rules:

- If the next PC is set to PC+1 for normal program flow, the fetch address sent to the context-lane interface is taken from the special fetch address adder specified in Section 4.2.6.
- If the next PC is taken from a different source and it is aligned, the fetch address is set to the next PC.
- If it is not aligned, two instruction fetches are needed to fetch the bundle at the branch target. The fetch address is still set to the next PC; the context-lane interface will round it down to align it. In order to prevent the partially fetched bundle from being executed, the branch unit invalidates it. This in turn causes the PC+1 logic to treat it as a zero-length bundle. The second part of the double instruction fetch can therefore be handled in the same way in which normal program flow is handled.

4.2.9 Execute

The ρ-VEX has three kinds of functional units: ALUs, memory units and multipliers.

ALUs are instantiated for every lane. An ρ-VEX ALU can perform roughly the following functions: additions, subtractions, bitshifts, bitwise logic, single-bit logic, C-style boolean logic, comparisons, sign/zero extension, count-leading-zeros and some miscellaneous operations. It has two integer operands (op1 and op2) and the branch operand at its disposal, and computes one integer result and/or one branch result, depending on the instruction.

The ALU is divided into three blocks. Stage boundaries may be enabled or disabled between each of these. The first block contains operand preprocessing logic, such as one's complement and preshifts. The second block contains an adder, a bitwise logic unit, a barrel shifter, a count-leading-zeros unit, and two equality checkers, all operating in parallel. The third block contains the output multiplexers.

The memory unit allows read and write operations to be performed on memory and control registers. One memory unit is available for each lane group; the exact lane it is instantiated in is configurable. All ρ-VEX memory operations first compute their address by adding one of the general-purpose register operands with the immediate. The ALU adder is used for this. To decrease latency somewhat, the address is connected directly to the ALU adder instead of also going through the output multiplexers, thereby also bypassing the third ALU block entirely. The latency of the memory unit is configurable to one or more cycles.

Memory instructions exist to load or store general-purpose registers as aligned 32-bit, 16-bit or 8-bit values. When loading 16-bit or 8-bit values, sign or zero extension may be used. In addition, the link register can be loaded or stored directly as an aligned 32-bit value, and the entire branch register file can be loaded or stored as an 8-bit value. All accesses are big endian.

Finally, the multiplier can execute various 16x16 and 16x32 bit multiplication instructions. The latency is fully configurable. The availability of a multiplier is design-time configurable for each lane individually.

As soon as a functional unit completes, its result is made available to the forwarding network. The result is committed to the register files in the write-back stage, if the bundle has not been invalidated by a branch or trap.

4.2.10 Register files

There are three physical register files in the ρ-VEX, namely the general-purpose register file, the context control register file, and the global control register file. The logical branch and link register files are implemented as special context control registers.

The general-purpose register files is the most complex, as it needs two read ports and one write port for each lane. For the full ρ-VEX this means that 16 read ports and 8 write ports are needed. This memory is constructed from two-port block ram (BRAM) primitives by duplicating the memory for each read port/write port pair. Each BRAM contains a copy of the complete memory. However, when the memory is written, only the BRAMs corresponding to that write port will have their contents updated. Thus, logic is needed to keep track of which write port wrote to a register last. Such logic is called a live value table (LVT). This is essentially yet again a memory with the same amount of ports as the complete memory; however, it is only as wide as the logarithm of the number of write ports. Thus, while it has to be implemented in LUTs, it still uses much less resources than implementing the whole memory in LUTs. A diagram of this is shown for two read ports and two write ports in Figure 4.9.

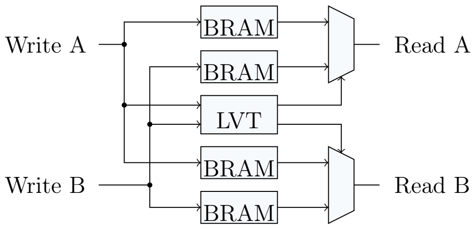


Figure 4.9: Diagram illustrating a memory with two read ports and two write ports constructed from dual port block RAM (BRAM) memories and a live value table (LVT).

A single BRAM is large enough to contain eight ρ-VEX register files. Thus, up to eight contexts can be supported by only increasing the size of the LVT. This is described in more detail in Appendix A.

The timing of the BRAMs is such that, if port A writes a value, the read result of port B for that value is undefined in the next cycle; its value is only guaranteed to be updated the cycle after. Because we always use one port for writing and one for reading, an additional forwarding stage is needed to override the undefined read. This leads to the WB+1 pipeline stage depicted in Figure 4.4.

The context control registers contains all control registers that need to be instantiated for each context. They are implemented in the general-purpose FPGA fabric. Examples are the PC, the trap status and control registers, the debug control registers, the reconfiguration request register, and performance counters. The global control registers are implemented similarly, although they are only instantiated once for the whole processor. These registers include the reconfiguration request register for the debug port, the reconfiguration status register, the cache block affinity status register, a cycle counter for timing, and a large set of hardware version and design-time configuration read-only registers. For details, refer to Chapter 4 of the ρ-VEX user manual (Appendix C).

4.2.11 Trap handling

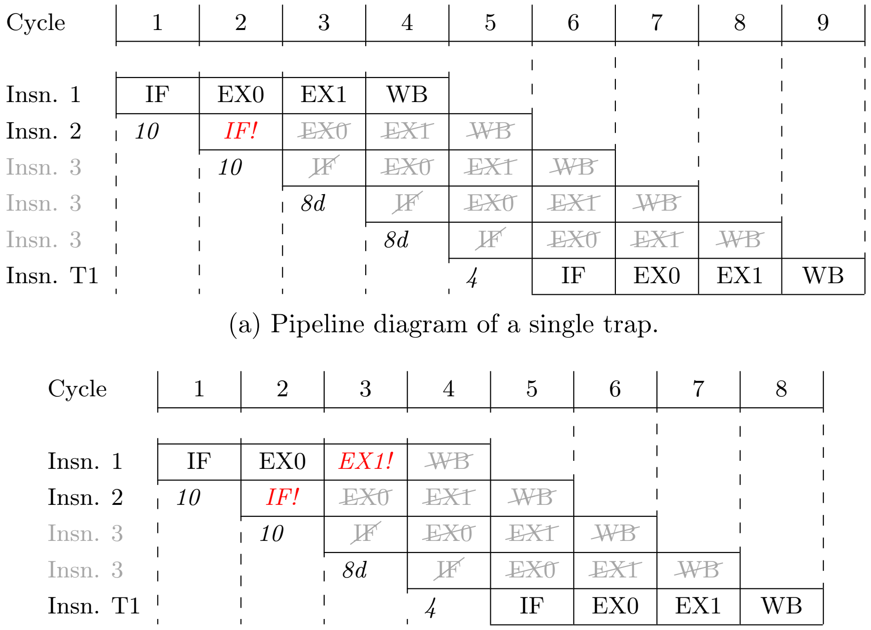
When any kind of trap occurs, all syllables in all pipeline stages up to and including the one in which the trap occurred are invalidated. That is, they will not commit to the register files or memory. This is done for all coupled lanes.

ρ-VEX instructions start to commit in the stage in which the data memory request is given. In the default pipeline, this is EX1. This means that traps occurring after EX1 cannot be precise, as the instruction has already partially committed its results. An exception is made for memory fault traps, which imply that the memory request has failed and thus did not affect the program state. If the data memory latency is configured to be more than one cycle, or if multiple data memory accesses can be handled by the memory system in parallel, it is up to the memory system to ensure that any parallel requests made by the same context are also canceled.

As traps can occur in many pipeline stages, traps may not be generated in program order. To handle this situation appropriately, trap handling is delayed until after the last pipeline stage that may cause a trap (EX1 by default). An additional delay cycle is implemented to break the path from the logic that generates the trap to the instruction fetch command.

Figure 4.10a depicts a pipeline diagram of how a single trap is handled. The trap occurs in cycle 2. From cycle 3 onwards, the branch unit receives a status signal indicating that a trap occurred, but the trap information has not yet been forwarded. This makes condition 8d in the branch priority encoder valid, preventing further instructions from being fetched until trap handler entry. The trap information is forwarded from EX1 in cycle 4. With the aforementioned delay cycle included, this causes the branch unit to respond to the trap with condition 4 in cycle 5.

Figure 4.10b illustrates why trap handling should be delayed. If the delay would not exist, the trap caused by instruction 2 would incorrectly take precedence over the additional trap caused by instruction 1.



Cycle	1	2	3	4	5	6	7	8	9
Insn. 1	IF	EX0	EX1	WB					
Insn. 2	*10*	*IF!*	~~EX0~~	~~EX1~~	~~WB~~				
Insn. 3		*10*	~~IF~~	~~EX0~~	~~EX1~~	~~WB~~			
Insn. 3			*8d*	~~IF~~	~~EX0~~	~~EX1~~	~~WB~~		
Insn. 3				*8d*	~~IF~~	~~EX0~~	~~EX1~~	~~WB~~	
Insn. T1					*4*	IF	EX0	EX1	WB

(a) Pipeline diagram of a single trap.

Cycle	1	2	3	4	5	6	7	8
Insn. 1	IF	EX0	*EX1!*	~~WB~~				
Insn. 2	*10*	*IF!*	~~EX0~~	~~EX1~~	~~WB~~			
Insn. 3		*10*	~~IF~~	~~EX0~~	~~EX1~~	~~WB~~		
Insn. 3			*8d*	~~IF~~	~~EX0~~	~~EX1~~	~~WB~~	
Insn. T1				*4*	IF	EX0	EX1	WB

(b) Pipeline diagram of two out-of-order traps.

Figure 4.10: Pipeline diagrams showing how traps are handled. The number in italics before the IF stage corresponds to the branch unit condition that selected the PC for the subsequent fetch.

When multiple traps occur in a single bundle, traps from earlier pipeline stages and, secondarily, lower indexed lanes take precedence. An exception is made for debug traps, which always take precedence even though they have to be generated in the memory request stage due to the breakpoint enable control register. This allows a user to set a breakpoint on an instruction which he or she expects will cause a trap, without having the trap handler be executed unexpectedly before the breakpoint is handled.

4.2.12 Run control interface

Aside from the memory and debug port interfaces, the ρ-VEX also has a control interface. This interface provides the following signals for each context independently:

- A reset signal, first PC and 'done' flag. The latter is activated when the ρ-VEX executes the `STOP` instruction. This allows the ρ-VEX to be used as a basic coprocessor. In particular, the 'done' signal could be wired such that it allows the ρ-VEX to interrupt the master when it finishes a task.

- A 'run' input and 'idle' output. May be used to pause execution temporarily for whatever reason.

- An interrupt interface. This consists of an interrupt request and a 32-bit identifier input, as well as an 'acknowledge' output signal.

Let us look at the interrupt interface in greater detail. Put simply, when the interrupt request signal is asserted while the interrupt enable flag is set, an interrupt trap is generated, using the 32-bit identifier as the trap argument. The 'acknowledge' signal is then used to signal to the interrupt controller that a trap has been handled. It is important that this is done in such a way that it is impossible for an interrupt to be acknowledged without it being handled, otherwise there is a possibility of missing an interrupt. In particular, since the interrupt trap is not the highest-priority trap, we cannot simply acknowledge the interrupt whenever interrupts are enabled.

This problem is solved by having the branch unit assign the interrupt identifier to the trap argument and acknowledge the interrupt in the same cycle only when it branches to the trap handler to handle the interrupt trap that was generated a few cycles earlier. In addition to solving the aforementioned problem, it also allows a higher priority interrupt to override the original interrupt within those cycles. However, if the interrupt controller allows interrupts to be canceled, there is a possibility that a trap is generated for an interrupt that is no longer active at trap handler entry. It is then up to the interrupt controller to specify a special 'canceled' interrupt identifier, which can then simply be ignored in software.

4.2.13 Tracing

When enabled, the trace unit takes execution information from the last stage of the pipeline of each lane, and serializes that data to a byte stream. That stream may then be logged outside the processor for evaluation. While serialization is performed, the entire processor is stalled. Furthermore, the trace data sink can stall the serialization if it cannot keep up with one byte per clock cycle.

The byte stream is divided up into trace packets. Each packet encodes the information for a single syllable. A packet is at most 37 bytes long, and can encode the following:

- The index of the lane that executed the syllable, and the index of the context that this lane is bound to, to convey the current runtime configuration.

- The PC, and whether it was fetched because of normal execution flow or some kind of branch.
- The memory operation performed (if any), including address, write data, and write bytemask. Read data is not recorded explicitly, but can be deduced from the register commit data.
- Any writes to the general-purpose, branch, or link registers, including which register is written as well as the written data.
- Information about the trap that caused the branch (if any), including the trap point, cause and argument.
- Cache information from the block associated with this lane for the reported bundle, including hit/miss information, whether the data access (if any) bypassed the cache, and whether or not the write buffer was filled when the request was made.
- The executed syllable.

Sending all this information for each executed syllable is not efficient. Assuming that the debug UART is used to transfer the trace data and eight-way execution, processing speed would be limited to merely 33 bundles per second. This performance loss is mitigated in several ways:

- Tracing can be enabled or disabled for each context independently.
- Information about memory accesses, register writes, traps, cache performance, and the current syllable can be disabled independently if they are not needed.
- Information that is common to all lanes (such as the PC) is only given once per cycle.
- If all information other than the PC is disabled, or for some other reason a trace packet would be empty save for the PC, the packet is omitted entirely, unless the instruction causes a branch or is the first instruction after a branch.
- If only the lower 8 bits or 16 bits of the PC have changed since the previous packet, the rest of the bits are omitted.

Which fields of a packet are omitted is encoded in two header bytes. Furthermore, the existence of the second header byte is encoded by a bit in the first. In the most extreme case, all fields are disabled, resulting in a NOP packet. These are used to encode clock cycle boundaries. The smallest useful packet is two bytes long, encoding a change in the lower 8 bits of the PC.

Trace buffer While not strictly a part of the processor (it is instantiated separately), the trace buffer can be used to make the trace data stream available to debug interfaces that can only do bus transactions, such as the one implemented in this work. The trace buffer is a bus slave consisting of two memory-mapped buffers. While the debug interface accesses one of these buffers, the other is used to record the trace data stream.

The buffers are implemented using BRAMs. The size of the buffers is configurable. The two buffers can share a single BRAM if the size is small enough. One of the two access ports of each BRAM is used to write the incoming trace data, the other port is connected to the bus slave interface.

Each buffer has an accompanying increment/reset counter. It is incremented for every byte written to the buffer. Its value is used for three purposes: stalling the processor when it reaches its maximum value (i.e., when the buffer is full), the write address for the port that records the trace data, and a memory-mapped status register. This status register is laid over the first word of the buffer, making the first 32-bit word inaccessible from the bus. For this reason, the counter register resets to four, so the first four bytes are never used.

Reading the trace buffer with the debug interface is done by alternating between reading the two buffers using bulk read accesses. As stated, buffer swaps occur automatically every time the interface starts reading the other buffer. For every bulk read, the first word indicates how many of the read bytes are valid. The invalid bytes and the counter can then simply be trimmed from the read data to extract the trace data.

4.3 Cache

In this section, we describe the implementation of the reconfigurable instruction and data caches. We first look at the interconnect between the ρ-VEX lane groups and the cache blocks. This is what gives the cache its reconfigurable nature. Subsequently, we look at how the cache blocks are implemented.

4.3.1 Reconfigurable interconnect

Figure 4.11 depicts the interconnect between the ρ-VEX lane groups and the cache blocks for both caches. Based on the runtime configuration, each routing logic block can either pass its inputs through without modification to decouple lane groups/cache blocks, or it can apply a logic function to couple them. This function depends on the signal. Let us list the signals that pass through the routing and the applied logic functions by tracing a request and its response through the network.

The data cache only allows one of the coupled lane groups to give a request at a time. This request is broadcast to all coupled blocks. If any routing block receives two requests in the same cycle, it asserts a fault signal, canceling the requests and trapping the offending ρ-VEX context. The instruction cache does not need any special routing logic for the request, as coupled ρ-VEX lane groups always fetch the same line. The current implementation, however, broadcasts the request of the highest-indexed coupled lane group to all blocks, and ignores the requests from the other coupled lane groups. This was done to make the cache compatible with the previous ρ-VEX as well.

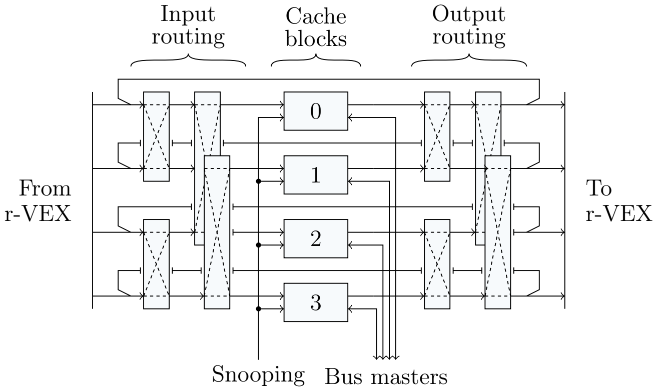


Figure 4.11: Cache block interconnect structure for both the instruction and data caches. Depicted for an ρ-VEX with four lane groups.

The cache blocks then return the hit/miss status and the cache line corresponding to the requested address after one cycle. The output network then routes these signals such that the signals from the highest-indexed coupled cache block that returned a hit are broadcast to all coupled lane groups. This is sufficient to handle data cache read hits. For the instruction cache, multiplexers are needed after the output network to select the appropriate syllables from the cache line. This could be done using the offset portion of the incoming addresses as generated by the context-lane interface of the new ρ-VEX, but again, for compatibility reasons, these offsets are generated within the routing network based on the runtime configuration.

If a read miss occurs, the affected ρ-VEX context is stalled. In addition, the signal is forwarded back to the input routing, which is then to determine which block should handle the miss, according to the selected replacement policy. As decided during the design of the cache, this is done simply using the address bits immediately following the cache tag. After the selected block finishes updating the tag and data memories from the bus, it will report a hit, thereby releasing the stall signal and supplying the newly read data.

The stall signals from the cache blocks are broadcasted only to coupled ρ-VEX lane groups in the output network. In other words, decoupled lane groups can be stalled independently.

Data cache writes are handled as follows. First, each block checks if the written address is currently cached. When that information is known, the routing networks determine which block is to service the write by assigning a 2-bit priority level to each block. The following levels are defined:

1. Highest priority, assigned when a block is already servicing the write to prevent a second block from 'taking over' the request.

2. Second highest priority, assigned when the written address is cached. This is necessary for cache consistency. Refer to Section 3.6.3 for more information.

3. Third highest priority, assigned when the write buffer is empty. This causes successive writes to be load-balanced between the available write buffers.

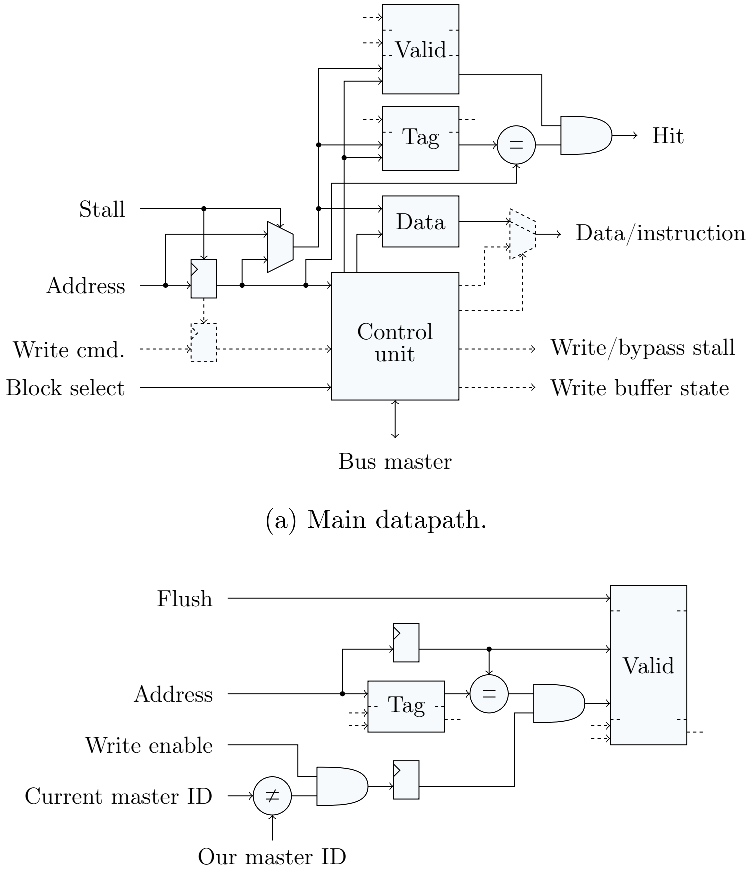
4. Lowest priority, assigned when none of the above apply.

If there is a tie, the highest-indexed block is chosen.

Finally, data accesses which are to bypass the cache are always handled by the highest-indexed block.

4.3.2 Cache blocks

Like the interconnect, the instruction and data cache blocks have a similar structure. It is shown in Figure 4.12. The primary difference between the two is that an instruction cache block does not support write or bypass operations, because the dashed paths do not exist.



(a) Main datapath.

(b) Invalidation and flushing datapath.

Figure 4.12: Block diagram of a cache block. The bus snooping and flushing datapath is drawn separately from the rest of the logic for clarity.

The data and tag memories are implemented as BRAMs, and thus have a one-cycle latency. The 'valid' memory is implemented in the configurable FPGA fabric however, because neither BRAM or distributed RAM can be reset in a single cycle, needed to support cache flushing. This is much more expensive in terms of area per bit, but this is still feasible as the 'valid' memory consists of only one bit per cache line.

Notice that the direct path from the control unit to the data/instruction output is dashed: it only exists for data cache blocks, and is only used for requests that are to bypass the cache. The data from a normal read miss is always passed through the data BRAMs. Essentially, while the control unit is executing the bus transactions to service the miss, the data, tag and 'valid' memories are continually being read. Thus, as soon as the control unit updates them, the remainder of the request is serviced as a read hit.

To understand the registers and the multiplexer on the left of Figure 4.12a, observe the timing diagram depicted in Figure 4.13. The request from the processor can only be considered valid when it is not stalled, as the request may depend combinatorially on the resource that is currently stalling the processor because it is not available yet. Thus, the request needs to be saved while it is being serviced. At the same time, it is desired to pass the request on to the memory blocks as soon as possible. This is the function of the multiplexer. While the processor is stalled, the registered request is passed on, but in the first cycle of a request, it is passed through combinatorially.

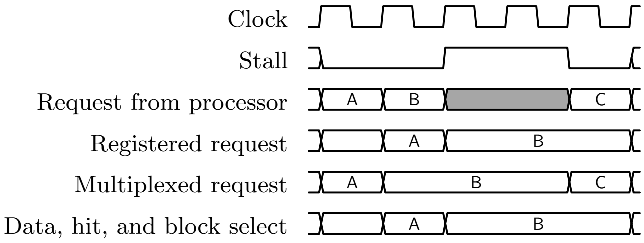


Figure 4.13: Cache block timing behavior, showing when certain signals are valid relative to the cycle in which the request is made.

Notice that the write command receives no such treatment. This is because the tag and 'valid' memory blocks need to be checked before the interconnect can select the cache block which is to service the write anyway. Thus, all write accesses take at least two cycles to complete, and thus incur at least one stall cycle.

4.4 Complete processing systems

In this section, the ρ-VEX processor, trace buffer, and cache are combined to form complete processing systems. To connect these devices together, a basic bus protocol is first designed and implemented in the first section. Two basic processing systems are implemented, one that uses BRAM-based instruction and data memories, and one that connects to GRLIB peripherals and DDR memory through the cache and an AHB bus bridge.

4.4.1 ρ-VEX bus

In order to be able to connect ρ-VEX project components together without having to rely on external IP or having to implement complex features that are not necessary or not supported by the ρ-VEX, an easy-to-use bus protocol is specified. The protocol is point-to-point; that is, it allows a single master device to connect to a single slave device. In order to connect more than two devices together, routing blocks must be instantiated. Two kinds of routing blocks are implemented; arbiters and demultiplexers.

An arbiter has multiple slave interfaces and a single master interface, thus connecting multiple master devices to a single slave device. When multiple masters make simultaneous requests, they are forwarded to the slave device one by one using round-robin scheduling. If necessary, a master can claim exclusive access to the slave device for any number of transfers by setting the 'lock' flag for all but the last request. This, for instance, allows read-modify-write accesses to be performed atomically.

A demultiplexer performs the opposite function, connecting a single master device to multiple slave devices. The selection is based upon the requested address. In addition, the demultiplexer allows the outputted slave address to be a non-unit function of the requested address, allowing address space transformations to be made. In certain situations, it also allows write transactions to be routed to multiple slaves at a time.

Figure 4.14 depicts how these blocks can be combined to form a conventional multi-master bus. In this diagram and in later ones we represent the slave end of an ρ-VEX bus connection with a filled arrow and the master end with a normal arrow.

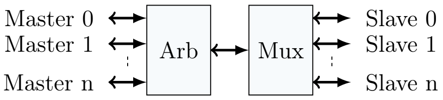


Figure 4.14: Diagram indicating how an ρ-VEX bus arbiter and demultiplexer can be used to construct a conventional multi-master bus.

In addition to the arbiter and demultiplexer, blocks are implemented that permit clock-domain crossings and/or insertion of registers into the request and response paths to break a critical path.

The timing of the ρ-VEX bus and the signals it consists of are depicted in Figure 4.15. Five transactions are shown as examples. Transaction A is a read that completes in a single cycle. Transaction B is also a read, but this read takes three cycles to complete. The slave indicates this by means of the 'busy' and 'ack' signals. While the slave reports that it is 'busy', the master must keep its request stable, preventing slaves from having to save the request themselves. Transaction C is a read yet again, but the slave device reports that it cannot complete it. It does this by asserting the 'fault' signal, while also asserting 'ack' to terminate the transaction. When the 'fault' signal is asserted, the read data signal may be used to provide an error code. Transaction D and E are writes. D completes successfully within a single cycle, whereas E takes two cycles to return a fault condition. Notice that, even though transaction E is a write, the read data signal may still be used to indicate the error code.

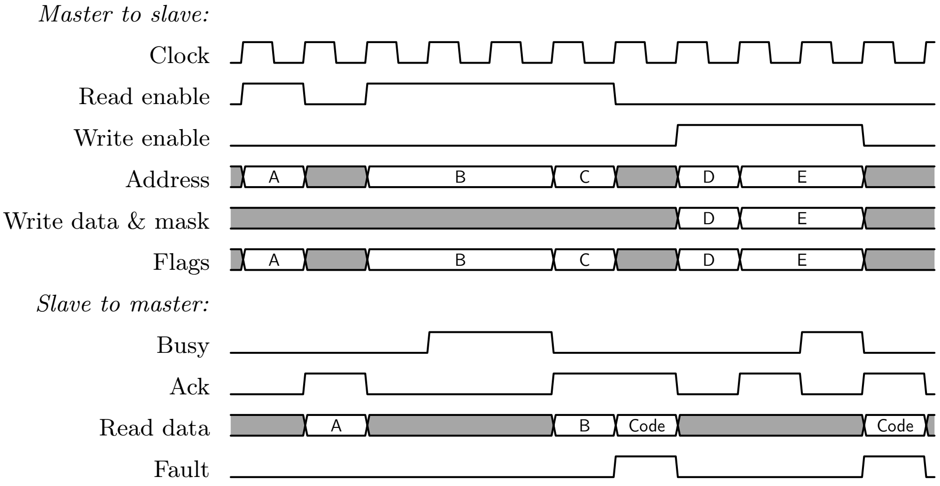


Figure 4.15: ρ-VEX bus timing example.

The 'flags' signal is a record consisting of burst identification signals and the 'lock' signal for arbiters. The burst signals are are not used by any ρ-VEX bus component, and are only intended for bridges to bus architectures that do support bursts.

The ρ-VEX bus is fixed to a width of 32 bits. Addresses must always be 32-bit aligned. It is not possible to read only a part of a 32-bit word. However, it is possible to write to 8-bit or 16-bit words by means of the write mask signal, which contains a mask bit for each of the four bytes in the word.

4.4.2 Standalone processing system

The standalone processing system is designed to provide a functional ρ-VEX system with minimal dependencies and deterministic timing for the memory system, to provide a platform for experimentation. It comes in two versions; one with a cache, and one without. These are shown in Figure 4.16.

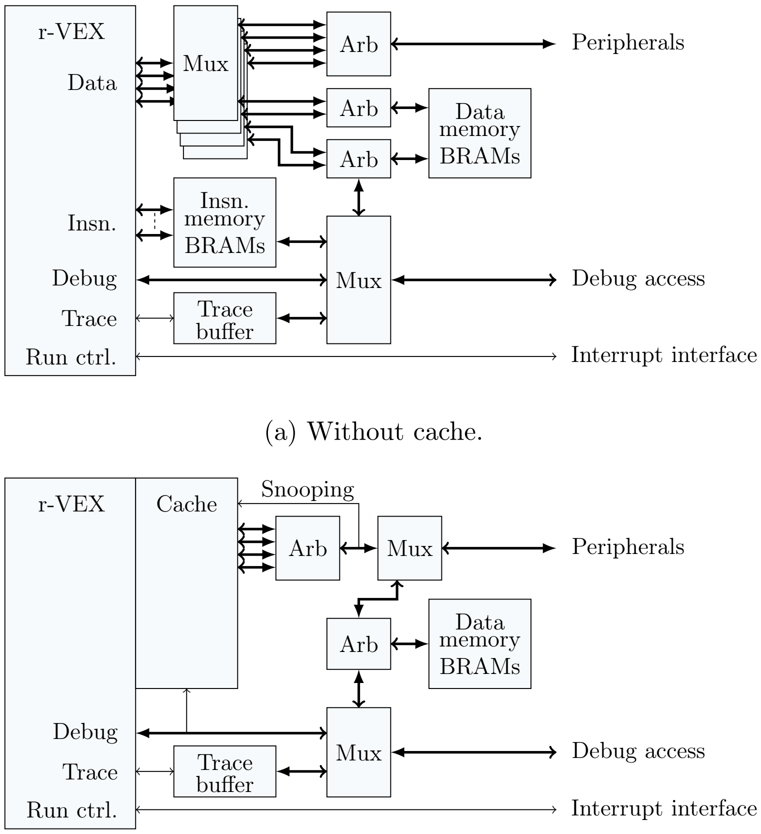
In the version without cache, as long as the debug access port is idle, instruction fetches are single-cycle. Memory operations only take more than one cycle when multiple requests are made simultaneously by either the first or the second half of the lane groups. This makes the platform suitable for testing the raw performance of the ρ-VEX, when not limited by the memory system.

The cached version is primarily intended for verifying the functionality of the cache in a controlled environment. The latency of the BRAM-based data memory is configurable at runtime using a control register. This control register can also be used to flush any cache block individually.

Both versions of the system are compatible with all design-time configuration options of the processor and the cache. In addition to that, the following things can be configured at design time:

- The sizes of the instruction and data memories.

- The memory map as observed by ρ-VEX memory operations.



(a) Without cache.

(b) With cache enabled.

Figure 4.16: Block diagram of the standalone processing.

- The memory map as observed by the debug access port.
- Which addresses should bypass the cache.

For more information about this system, refer to Section 9.3 of the ρ-VEX user manual (Appendix C).

4.4.3 GRLIB processing system

The GRLIB-based processing system is designed to allow the ρ-VEX to be connected to a wide variety of peripherals as well as the DDR3 memory of the FPGA development boards, by means of the GRLIB IP library [10]. The block diagram of this system is depicted in Figure 4.17.

The system control block contains control registers that can be used to reset the entire ρ-VEX system or flush any cache block individually. Note that the debug access port is an ρ-VEX bus interface, instead of an AHB or APB slave. Instead, an AHB-to-ρ-VEX-bus bridge is implemented as a separate unit. This allows multiple ρ-VEX slave devices to share the same AHB interface, similar to an AHB-APB bridge.

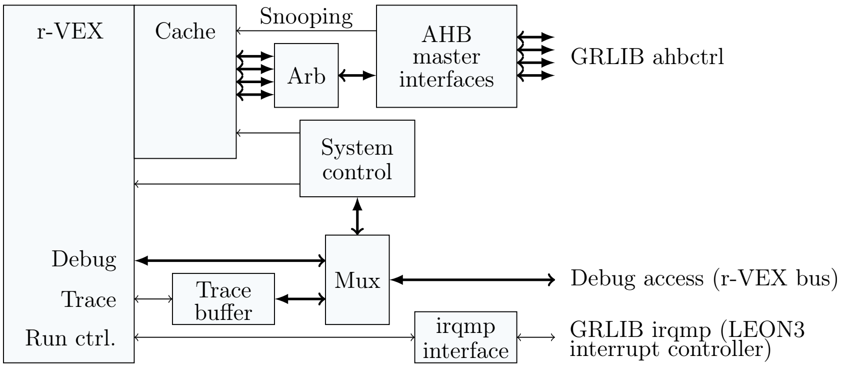


Figure 4.17: Block diagram of the GRLIB processing system.

For more information about this system, refer to Section 9.4 of the ρ-VEX user manual (Appendix C).

4.5 Debug UART

In this section, we discuss the implementation of the universal asynchronous receiver/-transmitter (UART) debug link peripheral. We first specify the communication protocol that will be used, and then describe the structure of the hardware.

4.5.1 Protocol

The UART protocol is an asynchronous serial protocol that encodes a stream of (usually) bytes using a single wire. Usually, two wires are used, one for either direction. Different frame formats exist to encode the frames [18, pp. 133]. We will be using the '8n1' frame format, i.e., 8 data bits, no parity bit, one stop bit. This format is shown in Figure 4.18.

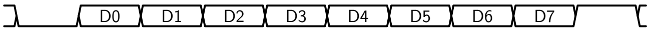


Figure 4.18: A UART frame.

Note that no clock signal is encoded. The transmitter and receiver must be configured to approximately the same frequency for transmission to be successful. When the receiver receives the start bit, it must wait 1.5 bit periods before sampling the first data bit, and then sample each subsequent bit at the expected bitrate.

As mentioned in Section 3.4.4, the debug UART is to be used both for executing bus transactions and to communicate with the application running on the ρ-VEX. Thus, we need to encode a packet stream as well as a byte stream, the latter emulating a normal serial port. Furthermore, error detection is desired for the packet stream.

To accomplish this, we first need a way to encode the start and end of a packet. We will arbitrarily reserve the characters `0xFD` and `0xFE` for these purposes. Furthermore, we reserve the character `0xFC` as an 'escape' character. Whenever this value is transmitted or received, the next value is to be one's-complemented. If any value between `0xFC` and `0xFE` is to be encoded as a data byte within the packet or application data stream, the 'escape sequence' is sent instead. This allows the packet receiver to reset itself to a defined state whenever `0xFD` or `0xFE` is received. Finally, in order to accelerate transmission of many subsequent packets, we specify that the end-of-packet code does not need to be sent if a second packet is started immediately after the first.

To add error detection to packets, we specify that the last byte in a packet is an 8-bit cyclic redundancy check (CRC) checksum, using $x^8 + x^2 + x^1 + x^0$ as the polynomial. The CRC is computed over the preceding data bytes of the packet without taking escape sequences into consideration. Note that the CRC byte itself may also need to be escaped. If a packet with an incorrect CRC is received, it is to simply be discarded.

Figure 4.19 shows an example transmission. The transmission represents the application data 'Hi' and two packets, `[0x00, 0x11, 0x22]` (CRC = `0xAC`) and `[0xFD]` (CRC = `0xFD`).

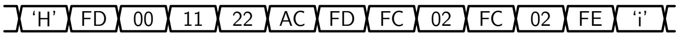


Figure 4.19: Example debug link protocol transmission.

Let us now specify the packet-level protocol. A transaction consists of a command and a reply packet. Transfers are always initiated by the host, so a packet sent from host to target is always a command, and vice versa. If the host sends a command but does not receive a reply within a set amount of time, one of the following three things happened: the command packet was received incorrectly by the target, the reply packet was received incorrectly by the host, or there is no connection at all. In any case, the host is to handle this case by resending the command packet until it does receive a reply, or until it has tried this a set amount of times. In the latter case, the connection is likely interrupted and the user is to be informed. Note that, while this protocol ensures that the command is executed or the user is informed, the command may be executed multiple times if one or more of the reply packets are corrupted.

The first byte of every packet constitutes the header. The high-order nibble of the header specifies a command code, whereas the low-order nibble is used as a sequence number. The header of a reply packet is always set to equal the header of the command packet it is replying to, allowing the host to identify which command packet a received packet was sent in response to.

Table 4.1 lists the commands supported by the debug link protocol. Observe that a distinction is made between single bus transactions and bulk transfers. The former is slow, as it requires two debug link transactions that cannot be parallelized. However, it supports all possible ρ-VEX bus transactions, can report bus faults, and it is guaranteed that the bus transaction is only performed once, regardless of retransmissions due to UART reply packet corruption.

The bulk read and write commands, on the other hand, are designed such that data can be transferred with as little overhead as possible, using only packets that fit within a 32-byte buffer. However, they are intended only for aligned 4 kiB blocks of memory. Furthermore, it is not guaranteed that the block is read/written in any specific order,

Table 4.1: List of debug link protocol command codes.

Code	Description
`0x0..0x9`	*Reserved.*
`0xA`	*Set bulk write page.* Sets the start address for subsequent bulk write commands. The command payload must be three bytes, specifying bit 31..8 in big endian order. The reply payload is undefined and should be ignored.
`0xB`	*Bulk write.* The payload from the host should be 5-29 bytes. The first byte sets the address: bits 31..12 are taken from the most recent bulk write page command, whereas bits 11 downto 0 are set to the byte value times 28. The remainder of the payload is interpreted as words which are to be written to the memory, starting at the initial address. The number of bytes written must be divisible by four, and bulk writes may not cross 4 kiB boundaries. The reply payload is undefined and should be ignored.
`0xC`	*Bulk read.* The payload from the host should be five bytes in size. The first four bytes should specify the initial address in big-endian order. The fifth byte should be set to what the LSB of the address will be when the bulk read should stop. This must be at most the LSB of the initial address plus 28, and at least the LSB plus four. Both addresses must be divisible by four. The reply payload contains the read data. Bulk reads may not cross 4 kiB boundaries.
`0xD`	*Prepare bus transaction.* The payload from the host should be nine bytes in size. The first four bytes specify the address of the bus transaction and the next four bytes specify the write data. The high-order nibble of the last byte specifies the write bytemask. Bit 3 must be set to perform a write and cleared to perform a read. Bits 2..0 must be zero. The bus transaction is not actually performed until the 'perform bus transaction' command is received. The reply payload is undefined and should be ignored.
`0xE`	*Perform bus transaction.* If the previously received command is a 'prepare bus transaction' command, the bus transaction described therein is performed. The payload from the host is undefined and should be ignored. The reply payload consists of five bytes. The first four bytes specify the read data. The fifth byte is set to 0 if the bus transaction completed successfully or to 1 if a bus fault occurred.
`0xF`	*Reserved.* This command code may result in the header byte needing to be escaped.

nor is it guaranteed that a memory location is only accessed once.

4.5.2 Structure

The structure of the debug UART is depicted in Figure 4.20. Notice that the interfaces are such that it can be connected directly to the standalone processing systems (recall Figure 4.16). The function of each block is described below.

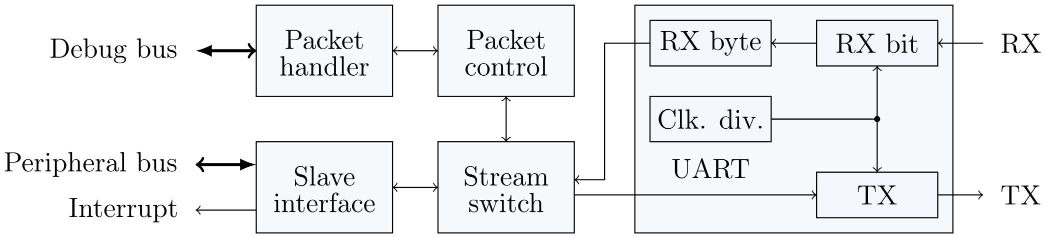


Figure 4.20: Structural overview of the debug UART.

Clk. div. This unit generates a strobe signal that is asserted eight times per UART bit period. A fractional divider is used to approximate the bitrate as closely as possible in the long run. The divisor and multiplier are generated automatically during synthesis by means of constant propagation. The debug UART uses 115200 bits/second by default, thus causing this unit to generate a pulse at a frequency of 921.6 kHz.

TX bit This unit serializes incoming to-be-transmitted bytes using the UART frame format.

RX bit This unit handles bit-level synchronization and performs glitch filtering. The unit synchronizes whenever the input signal has been stable for half a bit period after a transition. The value of a received bit is determined by means of majority voting between three samples centered around the expected center of the bit.

RX byte This unit handles frame-level synchronization. After it detects a start bit (the first low bit in Figure 4.18), it stores the subsequent 8 bits in a byte register and then forwards it to the stream switch block.

Stream switch This unit handles packet-level synchronization in the receive direction, and arbitration among the packet and application steam inputs in the transmit direction. It also handles the escape sequences.

Packet control This unit handles CRC checking and generation, and maintains the transmit and receive buffers. Both directions are double-buffered. Packets are 32 bytes in size at most, so 128 bytes of memory are used for the buffers. They are implemented such that the synthesizer infers distributed RAM for the buffers, thereby using only a minimal amount of area. Note that this is the primary reason for limiting the number of bytes per bulk transfer command to 28, instead of using a power-of-two amount.

Packet handler This unit contains the FSM that manages the ρ-VEX bus master interface. When the packet control unit signals that a new command is available, the receive packet buffer is swapped, allowing the FSM to decode and execute the command. While executing, the reply can be constructed in the transmit packet buffer. When the FSM is done and the previous reply has been completely sent, the transmit packet buffer is swapped, so the packet control unit can start transmitting it.

Slave interface This unit provides a slave interface for the application stream to the processor. It can also be tied directly to the UART block, in which case it forms a normal UART peripheral, although with a fixed bitrate. The interface was inspired by [36, pp. 182-208]. It contains a 16-byte first-in-first-out (FIFO) buffer for both the receive and transmit direction. It can be configured to generate an interrupt under the following circumstances:

- When the transmit FIFO buffer is empty.

- When the transmit FIFO buffer is not full.
- When the application wrote to the data register while the transmit FIFO buffer was full, in which case the written value was ignored.
- When there are at least 1, 4, 8 or 14 bytes available in the receive FIFO buffer. The amount is configurable using a 2-bit control register.
- When the receive line has been idle for at least one frame period.
- When a byte was received while the receive FIFO buffer was full, in which case the received byte is lost.

4.6 Debug software

As a custom external debug protocol is implemented, software needs to be written that makes use of this protocol. Three separate command-line utilities are implemented to fulfill this function. They are described in the following sections.

4.6.1 ρ-VEX server: `rvsrv`

The first of the three programs is called `rvsrv`, short for ρ-VEX server. It connects to the USB serial port of the FPGA development board and allows other programs to access it through two TCP server sockets. One of these servers acts as the other end of the virtual serial port peripheral that the ρ-VEX application can use freely. That is, any data or text sent by the ρ-VEX is broadcast to every connected TCP client, and any data from the TCP clients is merged and sent to the ρ-VEX. This is called the application port.

The other server, called the debug port, accepts memory access commands from any TCP client and returns the result to the client that sent it. If `rvsrv` receives commands from multiple clients at the same time, the commands are simply executed one by one.

TCP server sockets are used because of their ability to let multiple clients connect at the same time, and because they allow users to connect to the ρ-VEX through the Internet. This is particularly useful when the FPGA development used for the ρ-VEX is shared by multiple researchers. It is intended that `rvsrv` is run as a background process in this scenario, so a user connecting to `rvsrv` from the internet does not need execute or dialout permissions that would otherwise be needed to start it. For this purpose, `rvsrv` is written such that it will automatically reconnect to the serial port if the connection is lost, for instance because the FPGA board is power-cycled.

Connecting to the application port for manual interaction with the ρ-VEX can be done using `netcat`. This standard Linux tool can simply connect its `stdin` and `stdout` streams to a TCP server, thus essentially connecting the terminal it is run in to the ρ-VEX, as if the ρ-VEX application was run in that terminal.

4.6.2 ρ-VEX debugger: `rvd`

The commands that `rvsrv` expects on the debug port can theoretically be entered manually by the user. However, this is not very practical. Instead, a second program is created to do this, called `rvd` (short for ρ-VEX debugger). `rvd` is used as a command line tool, where every invocation typically only sends a single debug command before it terminates again. This makes it very easy to automate debug commands using shell scripts, makefiles, and so on.

The basic `rvd` commands are `rvd read` and `rvd write`. These commands may be used to read a register or a block of memory, or write to a register. Extensions of these commands allow file transfers to and from memory locations, allowing binaries to be uploaded to the ρ-VEX and result data to be downloaded. However, while these commands are technically enough to control the ρ-VEX and easier to use than writing `rvsrv` debug commands manually, it still is not very practical for debugging.

To improve things, additional commands are made for stopping and restarting the ρ-VEX, single stepping, printing the current state, and so on. As the ρ-VEX is intended to be a very configurable however, not every ρ-VEX processing system has the ρ-VEX control registers mapped to the same address, and not every ρ-VEX is the same either. To deal with this, such `rvd` commands are scripted through system-specific memory map configuration files. These files are specified using command line arguments. `rvd` is intended to be called either through an alias or through a shell script that appends these arguments to the user-specified ones automatically.

In order to support multiple contexts and in fact multiple processors, `rvd` allows the user to select one or more contexts using a command line parameter or through the `rvd select` command. Different addresses can be configured for different contexts in the memory map configuration files.

While `rvd` can be used reasonably efficiently on its own for small test programs, it has no concept of what program is being debugged. As such, you cannot, for instance, query `rvd` what the value of some C variable is or what line of code the program is currently stopped at: you have to cross-reference with the disassembled sources manually. For debugging more complex software, it is thus desirable to interface `rvd` with the existing (though buggy) ρ-VEX `gdb` port.

This is achieved using the `gdb` remote serial protocol. With the appropriate command line arguments, `gdb` can use this protocol to connect to the target using a TCP connection. Instead of connecting to the target directly, however, we connect it to `rvd`. Note that it is not desirable to simply implement the remote serial protocol in `rvsrv`, as `rvsrv` has no concept of memory maps or even a processor to be debugged. To prevent the user from having to set up the connection manually, the `rvd gdb` command that hosts the remote serial protocol server simply runs `gdb` as a child process with the suitable command line arguments.

4.6.3 ρ-VEX trace decoder: `rvtrace`

Software is also needed to handle tracing. In order to acquire the raw trace data from the trace buffer, `rvd trace` may be used, which enables tracing and then simply downloads data from the trace buffers to a file until some condition becomes true. However, this trace data is still in the raw binary format in this stage.

In order to decompress the trace data to a human-readable log file, `rvtrace` is created. `rvtrace` takes as input a raw trace file, the context index to interpret the data for, and a disassembly file. It then outputs the disassembly information for every executed instruction in the order of execution, while annotating the disassembly with the traced runtime information.

4.7 Conclusion

In this chapter, we have discussed the implementation of the complete ρ-VEX ecosystem.

In Section 4.1 we evaluated which hardware-description language is best suited for the project. We considered VHDL and SystemC in particular. We ended up choosing VHDL primarily due to the lack of toolchain support for SystemC.

The implementation of the processor was then extensively discussed in Section 4.2, starting with the reconfiguration system, and then going through the remainder of the components one at a time, roughly ordered by pipeline stage. We also described the trace buffer component at the end of the section.

Subsequently, the cache implementation was presented in Section 4.3. We first discussed the reconfigurable interconnect in detail, and then gave an overview of how the individual cache blocks are implemented.

In Section 4.4, we described ways in which the processor and cache can be tied together to form a complete processing system. In particular, we have seen the standalone system, designed to provide a controlled environment for doing experiments with the ρ-VEX, and the GRLIB-based system, which interfaces the ρ-VEX with the GRLIB IP library. In particular, the latter allows the ρ-VEX to use the DDR3 memories available on the FPGA development boards, thereby allowing large, complex applications to be run.

Finally, the protocol and implementation of the debug UART was described in Section 4.5, and the software used to control it was presented in Section 4.6.

Verification 5

In this chapter, we discuss how the functionality of the designed and implemented processing system is verified. This is done using a simulation-based conformance test and by running benchmarks on the synthesized design.

The conformance test suite for the processor is discussed in detail in the first section. In the subsequent section, a separate simulation-based verification platform for the cache is described, as the conformance test suite only tests the processor.

The remaining sections concern the synthesized design. First, the third section lists which design-time configurations are synthesized and how. It also lists the area and approximate operating frequency results. The fourth section lists the results of running the PowerStone benchmark suite [37] on the various synthesized designs. The fifth section explains how the functionality of the reconfiguration and trap systems is verified. The remaining two sections discuss the performance of the debug link and tracing respectively.

5.1 Conformance test suite

In order to test the processor, a simulation platform was constructed that automatically runs a set of conformance tests for various design-time configurations of the processor. It consists of a test runner implemented in behavioral VHDL, a set of test description files interpreted by the test runner, and a set of GNU `make` and `python` scripts that run the test suite in ModelSim, a simulation toolchain by Mentor Graphics, for each selected design-time configuration of the processor.

The conformance test is started using `make`. The `make` script first compiles any tests that are written in C, and then starts an instance of ModelSim for each design-time configuration. The `-j` command line switch allows multiple instances to be run on parallel to speed up the simulation. The ModelSim instances are automated using a TCL script, that compiles and loads the test runner VHDL, starts the simulation, and terminates ModelSim when the test runner signals completion. A `python` script filters the console output from ModelSim to write the result of each test to a log file. When all simulations complete, the `make` script returns whether any tests failed.

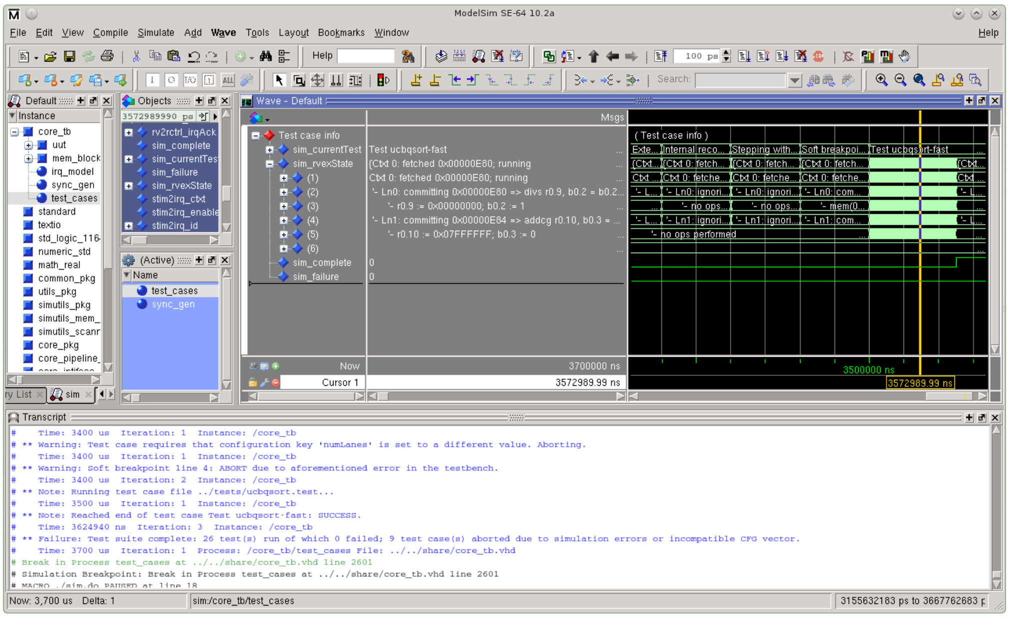
The VHDL test runner works by reading an index file that specifies a list of test description files. It then interprets the commands in these test files one by one. When the last test has been executed, the simulation is halted. Test description files may contain memory access commands, debug port commands, and the 'wait' command. The latter fulfills two purposes. First, it allows the execution of the next command to be delayed by a set amount of cycles. Secondly, it can make the test runner wait for a specified number of cycles for a specific event to occur. Events include the core starting or halting, instructions being fetched, or data memory being accessed by the core in a certain way. If the event does not occur in time, or (in some cases) a similar but different event occurs, the test case fails.

Initializing the memory may be done by means of commands that write integer values directly, a data file load command, or assembly commands. The latter allows syllables to be specified in a simple, human-readable way that is independent from the compiler toolchain. This is particularly important for testing error handling. Furthermore, the assembly syntax definitions are extracted directly from the opcode decoding table, and are therefore always in sync with the processor hardware description.

Test description files may also contain guards, that prevent the test case from being executed if an incompatible design-time configuration is used. This allows the same set of test cases to be used for various configurations, even if some test cases do not apply to some of them. When such a test case is encountered, the test runner marks the result of the test as inconclusive. It also does this if a test description is syntactically incorrect.

The instruction and data memories are modeled in such a way that they return explicitly undefined values for all cycles except for the ones in which the result is supposed to be valid. In addition, the access latencies are pseudo-randomly generated. This tests whether the timing of the memory interfaces is correct.

The conformance test suite may also be run in graphical user interface (GUI) mode. This allows the processor to be debugged should a test case fail. A screenshot of the GUI is depicted in Figure 5.1. Notice that the operations performed by the processor are available in a human-readable textual format in the waveform view. If a test fails or is inconclusive, the test runner also outputs a console message indicating what event caused the test to fail. This allows the test cases and the processor to be easily debugged.



```
Test ucbqsort-fast
{Ctxt 0: fetched 0x00000E80; running
Ctxt 0: fetched 0x00000E80; running
 '- Ln0: committing 0x00000E80 => divs r0.9, b0.2 = b0.2...
     '- r0.9 := 0x00000000; b0.2 := 1
 '- Ln1: committing 0x00000E84 => addcg r0.10, b0.3 = ...
     '- r0.10 := 0x07FFFFFF; b0.3 := 0
0
0
```

```
( Test case info )
Exte... Internal reco... Stepping with... Soft breakpoi... Test ucbqsort-fast
{Ctxt... {Ctxt 0: fetch... {Ctxt 0: fetch... {Ctxt 0: fetch... {Ctxt...
Ctxt ... Ctxt 0: fetche... Ctxt 0: fetche... Ctxt 0: fetche... Ctxt ...
'- L... '- Ln0: ignori... '- Ln0: ignori... '- Ln0: com... '- L...
... '- no ops... '- no ops... '- mem(0...
'- L... '- Ln1: ignori... '- Ln1: ignori... '- Ln1: com... '- L...
'- no ops performed
3500000 ns
3572989.99 ns
Now 3700000 ns
Cursor 1 3572989.99 ns
```

```
#    Time: 3400 us  Iteration: 1  Instance: /core_tb
# ** Warning: Test case requires that configuration key 'numLanes' is set to a different value. Aborting.
#    Time: 3400 us  Iteration: 1  Instance: /core_tb
# ** Warning: Soft breakpoint line 4: ABORT due to aforementioned error in the testbench.
#    Time: 3400 us  Iteration: 2  Instance: /core_tb
# ** Note: Running test case file ../tests/ucbqsort.test...
#    Time: 3500 us  Iteration: 1  Instance: /core_tb
# ** Note: Reached end of test case Test ucbqsort-fast: SUCCESS.
#    Time: 3624940 ns  Iteration: 3  Instance: /core_tb
# ** Failure: Test suite complete: 26 test(s) run of which 0 failed; 9 test case(s) aborted due to simulation errors or incompatible CFG vector.
#    Time: 3700 us  Iteration: 1  Process: /core_tb/test_cases File: ../../share/core_tb.vhd
# Break in Process test_cases at ../../share/core_tb.vhd line 2601
# Simulation Breakpoint: Break in Process test_cases at ../../share/core_tb.vhd line 2601
# MACRO ./sim.do PAUSED at line 18
```

```
Now: 3,700 us   Delta: 1   sim:/core_tb/test_cases   3155632183 ps to 3667762683 p
```

Figure 5.1: Screenshot of ModelSim after running the conformance test suite in GUI mode.

The following design-time configurations are currently tested by the conformance test suite:

- Configuration A: static two-way, fixed-length instructions (minimal configuration).
- Configuration B: eight-way, four lane groups, four contexts, fixed-length instructions, supporting both long immediate forwarding paths.
- Configuration C: eight-way, four lane groups, four contexts, variable-length instructions aligned by two syllables, only supporting long immediates forwarded from the other syllable in an aligned pair.

The created test cases and their results are shown in Table 5.1. They test the following things:

- Test 1 checks whether memory instructions and the `stop` instruction operate as expected in conjunction with the test runner.
- Tests 2 to 7 test the various register files and forwarding behavior, including the corner case where a register is read and written in the same cycle, and the behavior if a single register is written by multiple lanes at once.
- Tests 8 to 13 test the various long immediates routes, as well as trap behavior in case they are used in an incorrect or unsupported way. Only configuration C supports the long-immediate-from-previous-pair forwarding route, causing the different results.
- Tests 14 to 23 test the arithmetic logic unit (ALU) instructions for various inputs.
- Test 24 is a C program that computes the factorial for a number of inputs, thereby testing the multiplication unit.
- Tests 25 and 26 test various branch instructions.
- Test 27 tests the interrupt controller interface and the trap control registers.
- Tests 28 and 29 test the reconfiguration system. These are inconclusive for configuration A, because it only has one lane group and one context and thus does not support reconfiguration.
- Tests 30 and 31 test the debug interface. Both of these tests rely on reconfiguration.
- Test 32 consists of the PowerStone `ucbqsort` benchmark with a reduced input size. It intends to test compatibility with the compiler toolchain.

As can be seen in the table, all test cases succeed.

Table 5.1: List of test cases in the conformance test suite.

Test case	**Config. A**	**Config. B**	**Config. C**
1. sanity check	✓	✓	✓
2. GP. reg. consistency	✓	✓	✓
3. GP. reg. forwarding	✓	✓	✓
4. Br. reg. consistency	✓	✓	✓
5. Br. reg. forwarding	✓	✓	✓
6. link reg. consistency	✓	✓	✓
7. link reg. forwarding	✓	✓	✓
8. LIMMH from neighbor OK	✓	✓	✓
9. LIMMH from prev. OK	*n/a*	*n/a*	✓
10. LIMMH from prev. trap	✓	✓	*n/a*
11. LIMMH prefetching	*n/a*	*n/a*	✓
12. LIMMH to invalid lane trap	✓	✓	✓
13. LIMMH conflict trap	*n/a*	*n/a*	✓
14. ALU adder ops	✓	✓	✓
15. ALU bitwise ops	✓	✓	✓
16. ALU min/max/select ops	✓	✓	✓
17. ALU shift ops	✓	✓	✓
18. ALU single-bit ops/CLZ	✓	✓	✓
19. ALU sign/zero ext. ops	✓	✓	✓
20. ALU comparison ops	✓	✓	✓
21. ALU logical ops	✓	✓	✓
22. ALU accellerated division	✓	✓	✓
23. ALU link register moves	✓	✓	✓
24. factorial (multiplier)	✓	✓	✓
25. unconditional branch	✓	✓	✓
26. conditional branch	✓	✓	✓
27. interrupts/trap ID regs.	✓	✓	✓
28. externally-triggered reconf.	*n/a*	✓	✓
29. software-triggered reconf.	*n/a*	✓	✓
30. single step with reconf.	*n/a*	✓	✓
31. soft breakpoint trap	*n/a*	✓	✓
32. ucbqsort-fast	✓	✓	✓

5.2 Cache simulation

The conformance test suite discussed in the previous section uses a simulation model for the memories specially suited to test the processor, and thus does not include the cache. Therefore, a separate platform is constructed to test the cache.

This platform consists of a `make` script capable of compiling C programs for various ρ-VEX configurations, converting the binaries to a VHDL memory initialization file, and starting ModelSim to run the simulation. The simulation itself consists of a memory model for the cache to access with similar features as the conformance test suite, and a secondary memory model that monitors the instructions and data as accessed by the processor. Inconsistencies between the two are reported to the simulation console.

The cache simulation was run for various PowerStone benchmarks using an eight-way ρ-VEX with four lane groups for debugging purposes. However, doing extensive verification in this way is impractical due to the long simulation run time. The cache is mostly verified by means of the FPGA-based tests.

5.3 Synthesis

We now present synthesis results for the processor. Table 5.2 lists the designs that were synthesized. Note that all designs were synthesized using the default pipeline configuration. Experimentation with different configurations is left to future work.

Table 5.2: List of synthesized designs.

Design	Lanes	Lane groups	Contexts	Bundle alignment [syllables]	LIMMH from neighbor	LIMMH from previous	Lanes with multiplier	Lanes with memory unit	Instruction cache [kiB]	Data cache [kiB]	Memory [kiB]	ML605	VC707
gr96	8	4	4	2	✓		all	0,2,4,6	64	32	524288	✓	
gr24	8	4	4	2	✓		all	0,2,4,6	16	8	524288	✓	
sa96	8	4	4	2	✓		all	0,2,4,6	64	32	256	✓	✓
sa24	8	4	4	2	✓		all	0,2,4,6	16	8	256	✓	✓
4x2b2	8	4	4	2	✓		all	0,2,4,6	-	-	256	✓	✓
4x2b8	8	4	4	8	✓	✓	all	0,2,4,6	-	-	256	✓	✓
1x8b2	8	1	1	2	✓		all	0	-	-	256	✓	✓
1x8b8	8	1	1	8	✓	✓	all	6	-	-	256	✓	✓
1x2b2	2	1	1	2	✓		all	0	-	-	256	✓	✓

`gr24` and `gr96` are derived from the GRLIB version 1.3.7-b4144 LEON3 example project for the ML605 development board by replacing the LEON3 with the ρ-VEX GRLIB-based processing system (Section 4.4.3). The other designs use the standalone processing system (Section 4.4.2). The debug UART is included in all designs. The standalone designs also include a simple repetitive interrupt timer.

The designs are synthesized with Xilinx ISE 14.7. The standalone designs use the default optimization parameters. The GRLIB design is synthesized using the custom parameters that come with the example project, with the exception that retiming is disabled. This is done because the tools use an excessive amount of time and memory to synthesize the 'valid' memories in the cache otherwise.

The normal synthesis design flow requires the designer to set the operating frequency beforehand. The synthesis tools will then attempt to place and route the design such that it will function at that frequency. This flow is used for the GRLIB-based designs, using a 30 MHz clock. However, it is desired to determine how the design-time configuration of the processor affects the maximum frequency. Performance evaluation mode may be used to do this [38, pp. 103, 128]. In this mode, the synthesis tools ignore the user-specified timing constraints. Instead, the tools constrain themselves at runtime, increasing the

Table 5.3: Synthesis results.

Design		**Slices**	**LUTs**	**Registers**	**BRAMs**	**DSPs**	**Frequency**
`gr96`	ML605	32075	88018	58807	237	16	30.0 MHz
`gr24`	ML605	29469	76799	51152	213	16	30.0 MHz
`sa96`	ML605	19617	54125	22351	247	16	35.2 MHz
	VC707	23775	55388	22338	247	16	60.9 MHz
`sa24`	ML605	23883	66270	30047	271	16	35.3 MHz
	VC707	26662	68438	30032	271	16	61.5 MHz
`4x2b2`	ML605	18195	46975	16155	347	16	37.5 MHz
	VC707	20614	47458	16148	347	16	51.9 MHz
`4x2b8`	ML605	17591	46379	16044	347	16	34.0 MHz
	VC707	20262	47055	15972	347	16	57.8 MHz
`1x8b2`	ML605	11180	27321	8543	271	16	43.3 MHz
	VC707	12362	26040	8574	271	16	59.6 MHz
`1x8b8`	ML605	10781	26833	8134	271	16	44.4 MHz
	VC707	12345	27267	8143	271	16	61.7 MHz
`1x2b2`	ML605	4544	10679	3891	145	4	47.8 MHz
	VC707	4713	11057	3935	145	4	66.2 MHz

frequency constraint when possible and relaxing it when necessary. Still, the actual clock frequency that is generated by the frequency synthesizer must be specified at design time. It is set to 10 MHz, which is significantly lower than the expected maximum frequency, thereby making it highly probable that the synthesized design will work.

Table 5.3 lists an overview of the synthesis results. Figure 5.2 and Figure 5.3 depict the resource distributions of the processor and the cache respectively. Some notes:

- While the number of slice registers and lookup tables (LUTs) are similar for the ML605 and VC707, the VC707 design consistently occupies more slices. This is likely a result of lesser area constraints on the place and route tools, as the VC707 contains about twice as many slices as the ML605.

- Each of the synthesized eight-way processors use 128 block rams (BRAMs) for the general purpose-register file, and the two-way uses eight. The differing amounts of BRAMs between the reconfigurable and static designs is for the most part due to the the fact that the reconfigurable processor can access the instruction memory in more ways than the static processor. For instance, the first lane of a static eight-way ρ-VEX will only ever fetch 32-byte aligned words, while the first lane of a reconfigurable ρ-VEX can fetch any 8-byte aligned word.

- Each lane supporting multiplications uses two DSP blocks. This makes sense, as a single DSP block can only perform a 25x18 multiplication, and the ρ-VEX supports 32x16 multiplications.

- `4x2b8` is routed at a higher clock frequency than `4x2b2` on the VC707 as expected, but interestingly, this is not the case on the ML605. This is most likely due to the inherent randomness of the place-and-route heuristics.

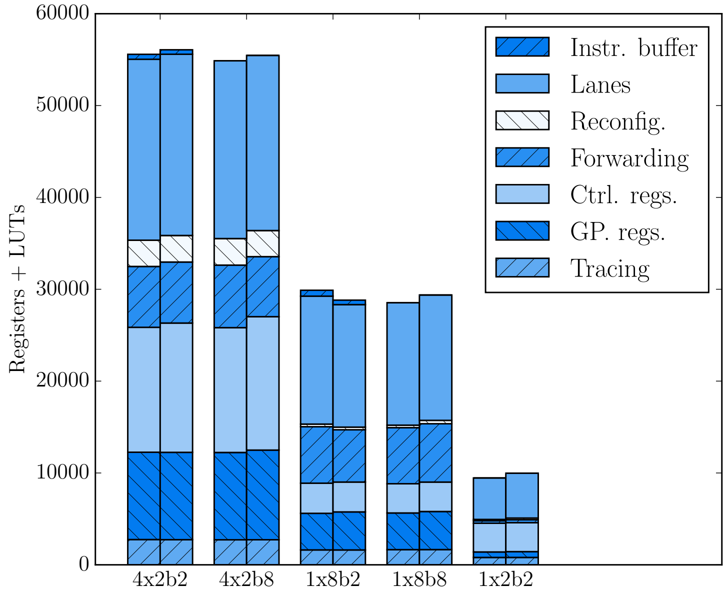
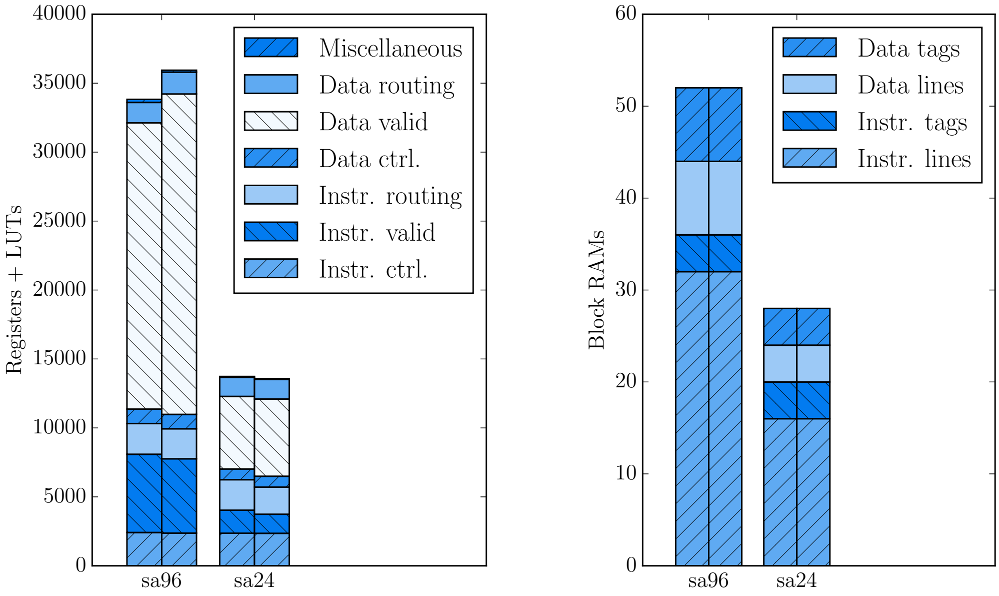


Figure 5.2: Register and LUT resource distribution of the ρ-VEX processor. The left bars represent the results for the ML605, the right bars represent the VC707.



(a) Registers and LUTs.

(b) Block RAMs.

Figure 5.3: FPGA resource of the reconfigurable cache. The left bars represent the results for the ML605, the right bars represent the VC707.

- The additional area needed to support variable-length instructions appears to be negligible.
- While the overhead of supporting reconfiguration does not seem to cost that much logic by itself, the area needed to store the additional contexts needed to support parallel execution is very significant.
- The majority of the register and LUT resources of the cache are devoted to the 'valid' memories. This is because they cannot be mapped to BRAMs or distributed RAM resources, as it needs to be possible to clear all bits at once to support cache flushing.
- The BRAM resources of the cache do not appear to scale linearly with the total size, implying that at least the `sa24` cache is not large enough to fill all the instantiated BRAMs. To understand this, consider that the cache is divided into four blocks. Therefore, each data cache block is only 4 kiB in size, while a BRAM is 36 kiB in size, explaining why `sa24` uses four BRAMs for the data lines. The same applies to the instruction memory, although here it is even worse, because the maximum width of a single-port BRAM is only 72 bits, while we need 256. Therefore, each instruction cache block needs four BRAMs, leading to sixteen BRAMs in total.

5.4 Benchmarks

In order to evaluate the performance and verify the correctness of the processor in hardware, benchmarks are run. The PowerStone benchmark suite [37] is used because it had already been ported to the ρ-VEX architecture. The initial reason for using PowerStone was because the applications in it are simple enough to not rely significantly on the C standard library, which had not yet been ported at the time[1]. The `auto` and `whetstone` benchmarks are missing as their sources could not be found. All benchmarks except `compress` and `des` were fitted with self-test code at the end that checks (part of) the computed result against the correct result. The result of the check is returned through the virtual serial port exposed by the debug link.

The benchmarks are compiled with HP VEX compiler version 3.43 [39]. In addition to the machine model and issue width, the following flags are specified: `-fno-xnop -fexpand-div -O2 -S`. The assembly output is then passed to the `vexparse python` program for the reconfigurable designs, which reschedules the assembly to allow it to be assembled as a generic binary. Finally, the code is assembled using the GNU `binutils` port for the ρ-VEX. `vexparse` and the `binutils` port were developed at the TU Delft Computer Engineering Lab and have not been published at the time of writing.

Compilation is done for each design individually, allowing the tools to optimize the code for the processor configuration. The benchmarks are compiled and run twice for the `sa24` and `gr24` platforms, once with the optimal settings, and once with the bundle length forced to eight syllables. This allows the behavior of fixed-length instructions to be tested on the design supporting variable-length instructions.

[1] At the time of writing, `newlib` has been partially ported, but it is still somewhat experimental.

Table 5.4: Benchmark results for the various configurations. The results of the individual benchmarks are combined. The designs marked with an * use code compiled with the bundle size fixed to eight syllables.

Design	Code size	Config.	Cycles	Stalls	NOPs	Insn. miss	Data miss	Speedup
`gr96`	514 kiB	8-way	8114 k	22.3%	25.4%	0.0%	7.3%	-
		4-way	8399 k	22.9%	25.4%	0.1%	8.0%	0.97
		2-way	10017 k	25.1%	25.4%	0.4%	9.8%	0.81
`gr24`	514 kiB	8-way	8848 k	28.7%	25.4%	0.5%	9.4%	0.92
		4-way	9344 k	30.7%	25.4%	0.7%	11.0%	0.87
		2-way	11710 k	36.0%	25.4%	1.6%	13.6%	0.69
`gr24*`	1392 kiB	8-way	9484 k	39.8%	74.5%	2.0%	9.4%	0.86
		4-way	27399 k	61.2%	74.5%	9.6%	10.8%	0.30
		2-way	53277 k	61.6%	74.5%	10.4%	13.6%	0.15
`sa96`	510 kiB	8-way	20465 k	69.2%	25.4%	0.0%	7.2%	0.47
		4-way	21659 k	70.1%	25.4%	0.1%	7.9%	0.44
		2-way	28507 k	73.7%	25.4%	0.4%	9.8%	0.33
`sa24`	510 kiB	8-way	27529 k	77.1%	25.4%	0.5%	9.3%	0.35
		4-way	30635 k	78.9%	25.4%	0.7%	10.9%	0.31
		2-way	47672 k	84.3%	25.4%	1.6%	13.6%	0.20
`sa24*`	1387 kiB	8-way	42377 k	86.5%	74.5%	2.0%	9.3%	0.23
		4-way	214208 k	95.0%	74.5%	9.6%	10.9%	0.04
		2-way	430756 k	95.3%	74.5%	10.4%	13.6%	0.02
`4x2b2`	510 kiB	8-way	6307 k	0.0%	25.4%	-	-	1.61
		4-way	6475 k	0.0%	25.4%	-	-	1.56
		2-way	7498 k	0.0%	25.4%	-	-	1.35
`4x2b8`	1387 kiB	8-way	5704 k	0.0%	74.5%	-	-	1.61
		4-way	10618 k	0.0%	74.5%	-	-	0.87
		2-way	20445 k	0.0%	74.5%	-	-	0.45
`1x8b2`	507 kiB	-	24999 k	0.0%	24.5%	-	-	0.47
`1x8b8`	1372 kiB	-	22641 k	0.0%	74.3%	-	-	0.53
`1x2b2`	579 kiB	-	28640 k	0.0%	23.1%	-	-	0.45

Table 5.4 lists the results of running the following benchmarks sequentially on the various designs: `qurt`, `crc`, `bcnt`, `blit`, `compress`, `des`, `engine`, `fir`, `g3fax`, `jpeg`, `pocsag`, `ucbqsort`, and `v42`. The self-test was completed successfully by all supporting benchmarks on all designs.

The 'config.' column in the table lists the runtime configuration with which the benchmarks were run. The 'stalls' column lists the percentage of the total cycles during which the processor is stalled due to the memory system. The 'NOPs' column lists the percentage of committed no-operation (NOP) syllables, providing an indication of code compression efficiency. The 'speedup' column lists the speedup of the designs with respect to `gr96`, computed using the operating frequency for the ML605 board listed in Table 5.3. The remaining columns should be self-explanatory. Some notes:

- The instruction cache miss rates are negligible in all but the sa24* and gr24* runs. This is due to the fact that the PowerStone benchmarks are all very short programs, that simply fit in the instruction cache in their entirety. The programs only become large enough to cause a significant amount of instruction cache lines to be evicted in the 16 kiB instruction caches when compiled with fixed-length bundles.
- The sa24 and sa96 designs were configured to simulate a memory access penalty of 20 cycles per 32-bit bus transaction. Evidently, the memory controller in the GRLIB platform is faster than that.
- The high amount of executed NOP syllables and the small difference in cycle count between 1x8b8 and 1x2b2 compared to the difference in execution resources indicate that the PowerStone benchmarks do not have a high amount of instruction level parallelism (ILP).

Table 5.5 lists the individual benchmark results for the gr24 design in 8-way mode, indicating the differences between the benchmarks. The code size of the benchmarks is dominated by the initialization code and the software floating point library. However, only qurt and fir use floating point arithmetic. This can be observed clearly from the instruction cache miss rate.

Table 5.5: Individual benchmark results for the gr24 design in 8-way mode.

Benchmark	Code size	Data size	Cycles	Stalls	NOPs	Insn. miss	Data miss
qurt	37 kiB	1 kiB	70 k	58.6%	20.7%	7.5%	1.5%
crc	36 kiB	1 kiB	18 k	10.7%	21.1%	0.4%	16.9%
bcnt	36 kiB	9 kiB	7 k	36.5%	8.8%	0.9%	81.9%
blit	37 kiB	9 kiB	33 k	54.2%	25.9%	0.4%	90.0%
compress	47 kiB	33 kiB	197 k	33.6%	20.8%	0.2%	26.5%
des	38 kiB	6 kiB	56 k	27.2%	12.0%	0.2%	9.1%
engine	40 kiB	1 kiB	764 k	9.0%	21.3%	0.0%	0.1%
fir	37 kiB	1 kiB	1103 k	55.2%	20.2%	6.2%	0.4%
g3fax	37 kiB	9 kiB	857 k	18.9%	24.4%	0.0%	0.8%
jpeg	39 kiB	79 kiB	2659 k	37.5%	25.2%	0.0%	15.4%
pocsag	39 kiB	2 kiB	25 k	20.7%	15.8%	0.7%	7.2%
ucbqsort	40 kiB	2 kiB	290 k	5.2%	28.7%	0.0%	0.8%
v42	44 kiB	40 kiB	2761 k	19.4%	29.2%	0.0%	10.2%

5.5 Runtime reconfiguration and interrupts

In the benchmark runs performed thus far, runtime reconfiguration and traps (aside from the stop trap caused by the stop instruction) are not used and are therefore not yet tested. Initially, the following method was used to test reconfiguration. First, four benchmarks are modified to run in a loop and output a single character to the serial port when a run completes. The character written depends on the benchmark and on whether the run was successful or not. The initialization code is then modified to start a different benchmark with a different stack pointer in each context. This program is compiled as a generic binary and run on the gr24 platform. While it runs, reconfigurations are requested manually through the debug interface. No erroneous runs were observed with extensive testing.

To test interrupts and simultaneously test runtime reconfiguration in a more scientific manner, a second program is made. This program includes two of the previously modified benchmarks, qurt and jpeg, which it runs in a loop for five seconds each. The number of runs and the actual time taken to run them are recorded (which may be slightly longer, as the individual runs are not interrupted when the time expires), so the average execution time of each benchmark can be determined afterward. qurt and jpeg are chosen specifically because qurt suffers from a significant amount of instruction cache misses and jpeg has the largest dataset.

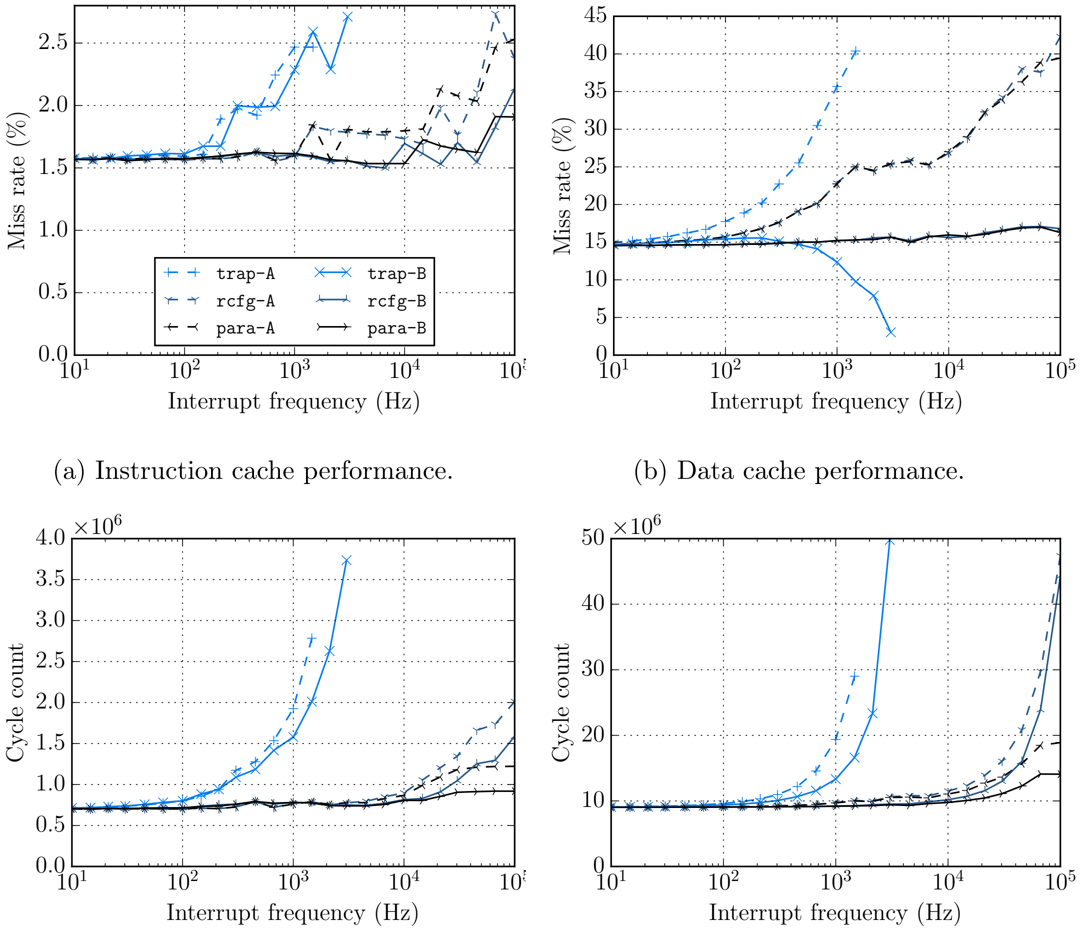
Before the first benchmark is started, the repetitive interrupt timer is configured to generate an interrupt at some specified rate. In the interrupt service routine, a reconfiguration is requested, such that the configuration with which the benchmarks are run toggles between two configurable configurations. The way in which the interrupt is serviced is also configurable; it may be serviced by trapping the context that runs the benchmarks, by using reconfiguring to interrupt the thread and run the handler in dedicated context, or by handling the interrupt in parallel to the benchmarks. The latter two methods use the interrupt-triggered reconfiguration system specified in the last paragraph of Section 3.2.4.

The program is run using the configurations listed in Table 5.6 at interrupt rates ranging from 10 Hz to 100 kHz. Notice that all configurations of the program toggle between running the benchmarks in a two-way and four-way configuration. The difference between configurations A and B is that the cache blocks used overlap in configuration B, but are different in A. This serves to test the measures taken to ensure cache coherence and consistency in a runtime reconfigurable system. Simultaneously, it allows the effectiveness of splitting and merging the caches as opposed to using different caches for each runtime configuration to be evaluated. Likewise, testing both interrupt handling methods serves to verify the functionality both systems, but also to evaluate the performance of either method.

Table 5.6: Configurations with which the reconfiguration/interrupt test program is run.

Name	Benchmark configs.	Interrupt handling method
trap-A	0x8808 and 0x0088	Regular trap
trap-B	0x8808 and 0x8800	Regular trap
rcfg-A	0x8818 and 0x1188	Interrupt by reconfiguring to 0x8880
rcfg-B	0x8818 and 0x8811	Interrupt by reconfiguring to 0x8088
para-A	0x8818 and 0x1188	Run in parallel by reconfiguring to 0x8810/0x1180
para-B	0x8818 and 0x8811	Run in parallel by reconfiguring to 0x8018/0x8011

The program is run using the sa24 design, using a simulated bus access latency of 20 cycles as before. No errors were detected while running the program. Figure 5.4 depicts the acquired performance results. Some notes:



(a) Instruction cache performance.

(b) Data cache performance.

(c) `qurt` average cycle count.

(d) `jpeg` average cycle count.

Figure 5.4: Performance results of the reconfiguration/interrupt test program.

- The data for the `trap` configurations ends at an interrupt frequency of around 3 kHz. This is because a single `jpeg` run at this point takes significantly more than five seconds to complete, causing the script that runs the tests to time out.

- As can be expected, dedicating a hardware context to only servicing interrupts is much more performant than handling the interrupts in the normal way.

- Handling the interrupts in parallel is slightly more performant than pausing the application. The effect is not very severe because the interrupt handler is very short and fits entirely in the cache.

- Configuration `B` almost always outperforms `A`, as expected.

- The increase in data cache performance at higher interrupt rates in configuration `trap-B` is due to the fact that the software context saving and restoring in the trap handler becomes more prominent.

5.6 Debug link and software

The debug link and software were used to debug and run all the previous tests, and have therefore already been implicitly verified to work. It is interesting to determine the performance of the link, however.

This was done by first uploading a 256 kiB file with randomized contents (generated using `dd` and `/dev/urandom`) to the memory of the `sa24` design, and subsequently downloading it again. The upload took 27.5 seconds, indicating a transfer speed of 9525 bytes per second. The download took 29.2 seconds, indicating 8980 bytes per second. Recall from Section 3.4.4 that the the theoretical bandwidth of the UART is 11520 bytes per second, although it has not been tested whether the USB to UART converter on the FPGA development boards is sufficiently buffered to sustain this speed.

5.7 Tracing

Finally, the functionality of the trace system is to be verified. This is done by tracing the benchmarks at various level-of-detail settings. The total cycle count and executed bundle count performance counters are then read when tracing completes to determine the average number of bundles executed per cycle. The results of these tests are listed in Table 5.7. The `gr96` design was used to run the tests.

Table 5.7: Performance of the trace unit using the UART debug link.

Level of detail	**Average bundles per second**
No tracing	20185102
Program counter only	8615
+ cache performance	1080
+ memory accesses	832
+ fetched instructions	150
+ all register writes	132

To give an indication of what the trace output from `rvtrace` looks like, the following listing shows a fully annotated instruction bundle from `qurt`.

```
# fetch for next bundle serviced by icache block 2: hit
    8530: 15 ac 08 80   stw 0x20[r0.1] = r0.22
                        # mem(0x000118C0) = 0xFD799000 (-42364928)
                        # Fetched syllable was 0x15AC0880
    8534: 59 00 68 42   orl b0.0 = r0.13, r0.2;;
                        # b0.0 = false
                        # Fetched syllable was 0x59006842
# write (full line) serviced by dcache block 3: hit
```

5.8 Conclusion

In this chapter, we have discussed the steps taken to verify the functionality of the ρ-VEX system, completing the final step of the method. In addition, the results of various performance measurement experiments were listed.

In Section 5.1, we have explained how the conformance test suite is implemented. Furthermore, we have listed 32 test cases, and have shown that all applicable test cases pass for three different design-time configurations of the processor.

As the conformance test suite does not include the cache, a separate simulation platform was created to test it. This platform is discussed briefly in Section 5.2.

Next, the synthesis results were listed for nine different designs in Section 5.3. Most of the designs were synthesized in performance evaluation mode, providing estimated maximum operating frequencies. These were found to range from 34.0 MHz to 47.8 MHz for the ML605 development board, and from 51.9 MHz to 66.2 MHz for the VC707 development board.

In Section 5.4, the PowerStone benchmark suite results were listed for the various synthesized designs. Then, in Section 5.5, we specified how the reconfiguration system was tested.

Finally, the performance of the debug link and the trace system were briefly discussed in Sections 5.6 and 5.7 respectively.

Conclusion 6

This chapter serves to conclude the thesis. In the first section, a summary is provided. The second section reiterates the problem statement and project requirements, and lists the main contributions made. The third section documents the additional contributions that were made during the course of the project. Finally, the fourth section lists recommendations for future work.

6.1 Summary

Chapter 2 presents the background information needed to understand this thesis.

In Section 2.1, field-programmable gate arrays (FPGAs) are introduced. Both the theory of operation of a generic FPGA and the features of the specific FPGA development board used in this project (the Xilinx ML605 and VC707) are discussed. Furthermore, it is described that FPGAs can be programmed using hardware description languages (HDLs), most notably VHDL, Verilog and SystemC. Finally, examples of other softcore processors are given, and use cases are presented.

Next, Section 2.2 details processor design. In particular, pipelining, trap handling, and debugging are discussed. Existing processor implementations are referenced to compound the theory. The ρ-VEX architecture and the existing implementations thereof are discussed in particular. At the end of the section, the components of the previous ρ-VEX implementations that are reused in the processor designed in this work are listed.

Finally, Section 2.3 discusses caches and other memory hierarchy components. Several cache implementation styles and their effect on cache consistency and coherence are described.

Chapter 3 describes the design of the main components of the new ρ-VEX processor.

Section 3.1 presents the design of the control registers. A distinction is made between context control registers and global control registers, the former being local to a thread and the latter being shared between all threads. The way in which the debug port is connected to the registers is also discussed.

Next, in Section 3.2, the runtime reconfiguration system is designed. It is illustrated how a configuration can be abstracted to a mapping from lane groups (the computational resources) to contexts (the register files and control logic), and how this can be encoded in a configuration word. Subsequently, the ways in which a reconfiguration may be requested are determined. Finally, the configuration word decoder and synchronization logic are designed.

The precise trap system is designed in Section 3.3. Here, it is determined that all traps can best be handled by the same set of trap handlers, regardless of the cause of the trap. Instead, it is defined that traps will be identified by means of a trap cause and a trap argument context control register. We then distinguish between the regular trap handler and the panic handler to deal with non-maskable nested traps. To allow interrupts to be postponed, an interrupt enable flag is specified. Finally, potential problems to do with returning from a trap handler after reconfiguration are illustrated, and the approach taken to solve these problems is described.

In Section 3.4, the debug system is designed. Here, it is decided that both self-hosted and external debugging are to be supported, and determined how this can be implemented. Particularly, a universal asynchronous receiver/transmitter (UART)-based debug interface peripheral is specified. Furthermore, the need for a trace unit is illustrated, and the required features are specified.

Subsequently, in Section 3.5, the design of the variable-length instruction system is described. It is determined that stop bits can be best used to encode the bundle boundaries, as the toolchain already supports this method. It is also argued that it is preferable to hide the complexity of fetching misaligned bundles from the memory system by implementing an instruction buffer in the processor.

The reconfigurable cache for the processor is designed in Section 3.6. Here, the requirement is set that a thread can only perform one memory operation per cycle. This allows the caches that belong to separate lane groups to be combined to form a larger, set associative cache when multiple lane groups are coupled. Next, it is discussed how transient cache performance after a reconfiguration can be optimized, although we specify that many optimizations will not be implemented and are left to future work. Finally, cache coherence and consistency problems caused by runtime reconfigurations are illustrated, and solutions to these problems are provided.

Finally, although not a requirement of the project, we specify additional requirements for design-time configurability in Section 3.7 that are relatively simple to implement, intended to accelerate future research.

Chapter 4 presents the implementation of the complete ρ-VEX ecosystem.

In Section 4.1, the hardware-description language used to implement the design is chosen based on the context of the project. VHDL and SystemC are considered in particular. VHDL is ultimately chosen, primarily due to the lack of SystemC toolchain support for the ML605 development board.

The implementation of the processor is extensively discussed in Section 4.2, starting with the reconfiguration system and then going through the remainder of the components one at a time, roughly ordered by pipeline stage. The trace buffer peripheral is also discussed along with the trace unit at the end of the section.

Subsequently, Section 4.3 discusses the cache implementation. The implementation of the reconfigurable interconnect between the cache blocks is first described in detail. The implementation of the cache blocks themselves is discussed in lesser detail, as they are not affected by any of the unique features of the processor.

Section 4.4 describes ways in which the processor and cache can be tied together to form complete processing systems. Two processing systems are implemented: the standalone system, designed to provide a controlled environment for conducting experiments with the ρ-VEX, and the GRLIB-based system, which interfaces the ρ-VEX with the GRLIB IP library. In particular, the latter allows the ρ-VEX to use the DDR3 memories available on the FPGA development boards, thereby allowing large, complex applications to be run.

Finally, the protocol and implementation of the debug UART are described in Section 4.5, and the software written to control the debug link is presented in Section 4.6.

Chapter 5 details the steps taken to verify the functionality of the system. In addition, the results of various performance measurement experiments are provided.

Section 5.1 explains how the conformance test suite is implemented. Furthermore, 32 test cases are described, and it is shown that all applicable test cases pass for three different design-time configurations of the processor.

As the conformance test suite does not include the cache, a separate simulation platform was created to test it. This platform is discussed briefly in Section 5.2.

Next, synthesis results are listed for nine different designs in Section 5.3. Most of the designs were synthesized in performance evaluation mode, providing estimated maximum operating frequencies. These are found to range from 34.0 MHz to 47.8 MHz for the ML605 development board, and from 51.9 MHz to 66.2 MHz for the VC707 development board.

In Section 5.4, the PowerStone benchmark suite results are listed for the various synthesized designs. Then, in Section 5.5, the ways in which the reconfiguration system is tested is described.

Finally, the performance of the debug link and the trace system is briefly discussed in Sections 5.6 and 5.7 respectively.

6.2 Main contributions

The problem statement of this thesis was:

> *How to design and implement a dynamically reconfigurable and parameterizable VLIW processor?*

We have answered this question by presenting a possible design, implementing it, and verifying the functionality on two FPGA development boards. Particularly, we have demonstrated that runtime reconfigurability can be accomplished by means of a reconfigurable interconnect structure between the register files (the contexts) and the functional units (the lane groups). The number of contexts, lane groups, and the issue width are parameterizable at design-time, among other things.

At the start of the project, a number of requirements were set. These are listed again here, along with a short summary of how the requirement was met.

1. *The design must be compatible with the current ρ-VEX compiler toolchain.*

   This was accomplished by making the new design binary-compatible with the design presented in [3].

2. *The design must be dynamically (runtime) reconfigurable.*

   As stated, this was accomplished by separating the processor into contexts and lane groups, connected by means of a reconfigurable interconnect. Reconfiguration is controlled by a state machine that first decodes the requested configuration, then determines which contexts and lane groups are affected by the reconfiguration, and subsequently halts these contexts in order to synchronize them before committing the new configuration. The reconfiguration overhead is only six cycles, measured from the last instruction issue before the reconfiguration to the first instruction issue after the reconfiguration.

3. *The design must support precise traps.*

   Trap preciseness is accomplished by flushing the pipeline when a trap occurs. Handling the trap is delayed until earlier instructions can no longer cause traps to ensure in-order trap handling. If a trap occurs while the reconfiguration controller is halting the running context, the program counter (PC) is set to the trap point, so the trap will be generated again after resumption. Special cases were identified with regards to returning to the application after handling a trap if a reconfiguration occurred during the execution of the handler. These cases and the way in which they are handled are discussed in detail in Section 3.3.5.

4. *The design must support debugging.*

   Both self-hosted and external debugging were implemented. In self-hosted mode, a trap is generated when a breakpoint or watchpoint is hit; in external debug mode, the context is halted. To allow external debug peripherals to access the processor, all register files are exposed through a bus slave interface.

   A UART-based debug interface peripheral was developed to facilitate external debugging. A custom communication protocol was developed for this interface. This protocol allows the application running on the ρ-VEX processor to use the debug peripheral as a normal UART, while at the same time allowing the host to give bus access commands. These commands are decoded and handled using a state machine in the peripheral, so they are transparent to the application. Transmission errors in the commands are detected using a cyclic redundancy check (CRC). Timeout and retransmission is employed to correct errors.

   A set of Linux tools was developed to communicate with the debug interface on the host side. This software can either be used for debugging on its own, or in conjunction with the existing `gdb` port [5].

   Finally, in addition to supporting traditional debugging, a trace system was constructed. This system allows various execution information to be recorded for later analysis, without needing to interrupt the program to analyze the current state.

5. *The design must support variable-length instructions.*

   Variable-length instructions are supported by allowing bundle boundaries to be explicitly marked using stop bits. When the stop bit is set in a syllable, subsequent syllables that are fetched in parallel are disabled, and the PC is incremented only by the amount of executed syllables. To handle the resulting misaligned instruction fetches, an 'instruction buffer' was developed. This buffer splits a misaligned fetch into two aligned instruction memory accesses, which are then performed sequentially. The instruction buffer always stores the result of the latest memory access, such that consecutive instruction fetches require at most one memory access. In other words, only misaligned branches require two memory accesses; any other instruction fetch can still be serviced in a single cycle.

6. *The design must include a coherent, dynamically reconfigurable cache.*

   This was accomplished by separating the instruction and data caches into multiple blocks, one for each cache/lane group pair. The blocks are connected to each other and to the processor through a reconfigurable interconnect, such that when lane groups are coupled in the processor, the associated blocks function as sets in a set associative cache. Coherence is achieved by using the write-through method in conjunction with bus snooping. Each block is equipped with a write buffer to hide the memory access latency for writes as much as possible. While the write buffer is filled, reconfigurations involving that buffer are blocked to ensure cache consistency.

In addition to those needed to meet the requirements, the following features were also designed and implemented:

- *Configuration system*

  To make the ρ-VEX design-time configurable beyond parameterization, a configuration system was developed, capable of generating parts of the hardware description. The relevant parts of the user manual, C/assembly header files, and debug interface memory map specifications are generated in tandem, to ensure that they remain synchronized. This system fully specifies the control register functionality and layout, and partially specifies instruction encoding, trap identification, and the pipeline layout.

- *GRLIB processing system*

  In order to test the cache with an off-chip memory controller, and simultaneously allow large applications to be run on the ρ-VEX, an AHB bus interface was constructed to allow the ρ-VEX to be used in conjunction with the GRLIB IP library and associated memory controller interface.

6.3 Additional contributions

During the course of the project, several additional contributions were made. These are listed in this section.

- *Vexparse*

  None of the C compilers currently available for the ρ-VEX can output code that can explicitly be assembled as a generic binary directly. Instead, it is the task of the assembler to schedule syllables accordingly if a generic binary is desired. However, the assembler was not able to do this in all cases, requiring the programmer to manually modify the assembly code. As a workaround, a `python` program called `vexparse` was developed by A. A. C. Brandon to do register renaming between the compilation and assembly phases. I have contributed to this program to allow it to fully reschedule basic blocks when needed, as register renaming alone was sometimes not enough.

- *ALMARVI project contribution*

  ALMARVI is a European research project aiming to construct a low-power many-core execution platform for low-power image processing [40]. The author has contributed to this project by constructing a wrapper for the standalone processing system compliant with the ALMARVI coprocessor interface specifications. In particular, an AXI slave to ρ-VEX bus bridge was developed. The system was tested using the Zedboard FPGA development board, based on the Xilinx Zynq XC7Z020-1CLG484 FPGA.

- *Conference papers*

  Two conference papers based on the processor implemented in this work were co-authored. The first of these papers discusses the implementation of the general-purpose register file in conjunction with runtime reconfiguration and multiple contexts ([7], Appendix A). The second describes the variable-length instruction encoding technique utilized ([8], Appendix B). Both papers were presented at the 2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig). Two additional papers were based on the implemented processor but were not co-authored [41][42].

- *User manual*

  In addition to this thesis, a manual was written to describe the functionality of the processor in more detail. It was written such that changes made in the configuration files for the processor are automatically reflected, as long as the documentation in the configuration files is kept up-to-date. It intends to make the processor easily understandable and usable by future researchers. The current version of the manual is attached to this thesis in Appendix C.

- *Technical support*

  The implemented processor has enabled several other MSc projects. Three of these projects have already been completed: [43], [44] and [45]. I have provided advise and technical support to these students during their projects to help them use and modify the processor. I was asked and am intending to continue this work for at least two more years as an employee of the TU Delft.

6.4 Future work

This section lists recommendations for future work.

6.4.1 Implementation

The following list shows examples of things that can be improved in the current hardware implementation.

- The opcode encoding used by the processor is defined in a configuration file. However, the assembler toolchain (GNU binutils) does not currently use this configuration. It may be useful to add support for this in the future, to make it even simpler to configure the instruction set.

- The line sizes and the number of bus access ports of the instruction and data caches are currently fixed. This was done merely to keep the initial implementation simple. It may be possible to improve the performance and area efficiency of the system by changing these parameters.

- The current variable-length instruction system requires a path from every 32-bit instruction memory word to every lane. It may be possible to optimize this by imposing additional restrictions on the placement of syllables within a bundle to reduce logic resources.

- When a breakpoint or watchpoint is hit, the current debug unit can only halt the context that caused it. It may be useful to add a feature that allows it to halt the entire processor as well, as this may be helpful in debugging parallel programs and synchronization methods.

- The trace unit currently has only limited use in debugging the cache and/or memory system, as it slows down execution and thus affects the timing of the cache/memory accesses. It would be useful to construct a secondary trace unit that is capable of logging data in bursts, either without stalling the processor or by clock-gating everything except for the trace data output.

- Floating point operations are very expensive on the ρ-VEX due to the lack of a hardware floating point unit. Implementing such a unit would make the ρ-VEX suitable for a wider range of applications.

- During the course this project, a memory management unit (MMU) was designed and implemented for the ρ-VEX in [45]. This MMU was integrated and tested with the latest version of the processor and cache at the time, but changes have since been made. Effort should thus be put into updating the MMU-enabled design.

- Similar to the MMU, a fault-tolerant version of the ρ-VEX was developed in [43], which is also based on an older minor version and should thus be updated. In particular, it may be integrated such that it can be enabled or disabled at design-time, allowing it to be merged into the mainline design without a hardware penalty in applications that do not need fault tolerance.

- In addition to an MMU, contemporary operating systems also require security features such as privilege levels from the hardware.

In addition, the following things could be improved on the software side.

- The current Linux kernel port uses an old MMU-less branch of the kernel. Now that an MMU is available, a more recent version can be ported.

- Two C standard libraries have been partially ported thus far: `uClibc` for programs compiled for the Linux kernel, and `newlib` for standalone applications. However, many system calls have not been implemented. This prevents complex programs from being compiled for the ρ-VEX, including many benchmark suites. Thus, work could be put into extending C standard library support.

- None of the compilers currently available for the ρ-VEX have C++ support that is verified to work.

- It is currently impossible to single-step through lines of C code using `gdb` or otherwise, as the assembler does not generate accurate debug symbols for anything but the start of a function. This is likely related to the fact that a very large instruction word (VLIW) processor is explicitly parallel and may thus be processing multiple lines of code at once. It should however be possible to prevent this behavior in the assembler when compiling with debug symbols and without optimizations.

6.4.2 Research opportunities

The following list shows examples of future research that can be conducted on the ρ-VEX processor.

- Only one pipeline configuration has been evaluated and verified to work. The effect of the configuration on the area, speed, and energy efficiency should be researched.

- The replacement policy of the cache when multiple blocks are coupled is simplistic in the current implementation. Using a better policy, such as least recently used (LRU), may improve cache performance significantly. It may also be possible to minimize the cache penalty of reconfiguration in cases where the program is aware of the reconfiguration in advance, by allowing the program to limit the blocks used to handle cache misses to those that will also be available in the later configuration. Thus, research should be conducted to determine an appropriate policy.

- The system currently does not support any primitive synchronization operation such as compare-and-swap or load-link/store-conditional. This limits synchronization between threads to less performant software-based approaches. Reconfiguration may affect the way in which such primitives can be implemented. Thus, research should be conducted to determine a suitable synchronization operation for the ρ-VEX.

- In Section 3.2.2, we argued in favor of an any-to-any interconnect network between contexts and lane groups in order to maximize the flexibility of the design. It may be worth researching whether such a network is worth the cost compared to more constrained solutions.

- It should be researched how competitive the ρ-VEX is compared to conventional architectures. In particular, it is hypothesized that the ρ-VEX can reach a similar energy efficiency as a conventional VLIW processor for applications with high instruction level parallelism (ILP), and similar performance as conventional multicore processors for applications with high thread level parallelism (TLP). In order to do this, the ρ-VEX must first be implemented as an application-specific integrated circuit (ASIC), as FPGAs are fundamentally less efficient than a custom silicon implementation.

Bibliography

[1] Xilinx, *Virtex-6 Family Overview*, Aug. 2015. DS150 v2.5. Available: `http://www.xilinx.com/support/documentation/data_sheets/ds150.pdf` (visited on Mar. 15, 2016).

[2] Xilinx, *7 Series FPGAs Overview*, May 2015. DS180 v1.17. Available: `http://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf` (visited on Mar. 15, 2016).

[3] R. A. E. Seedorf, "Fingerprint Verification on the VEX Processor," Master's thesis, Delft University of Technology, the Netherlands, 2010.

[4] T. v. As, "ρ-VEX: A Reconfigurable and Extensible VLIW Processor," Master's thesis, Delft University of Technology, the Netherlands, 2008.

[5] J. J. Hoozemans, "Porting Linux to the rVEX Reconfigurable VLIW Softcore," Master's thesis, Delft University of Technology, the Netherlands, 2014.

[6] P. Gavin, "CARPE Project." Available: `https://github.com/carpe-project/carpe` (visited on Mar. 14, 2016).

[7] J. Hoozemans and J. Johansen and J. V. Straten and A. Brandon and S. Wong, "Multiple Contexts in a Multi-Ported VLIW Register File Implementation," in *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pp. 1–6, Dec 2015.

[8] A. Brandon and J. Hoozemans and J. van Straten and A. Lorenzon and A. Sartor and A. C. Schneider Beck and S. Wong, "A sparse VLIW instruction encoding scheme compatible with generic binaries," in *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pp. 1–7, Dec 2015.

[9] S. Brown and Z. Vranesic, *Fundamentals of Digital Logic.* McGraw-Hill Europe, third ed., 2009.

[10] Cobham, *GRLIB IP Library User's Manual*, Jan. 2016. v1.5.0. Available: `www.gaisler.com/products/grlib/grlib.pdf` (visited on Mar. 15, 2016).

[11] Xilinx, *ML605 Hardware User Guide*, Oct. 2012. UG534 v1.8. Available: `http://www.xilinx.cofisher2005m/support/documentation/boards_and_kits/ug534.pdf` (visited on Mar. 15, 2016).

[12] Xilinx, *VC707 Evaluation Board for the Virtex-7 FPGA*, Sep. 2015. UG885 v1.6.1. Available: `http://www.xilinx.com/support/documentation/boards_and_kits/vc707/ug885_VC707_Eval_Bd.pdf` (visited on Mar. 15, 2016).

[13] Xilinx, *Virtex-6 FPGA Configurable Logic Block User Guide*, Feb. 2012. UG364 v1.2. Available: `www.xilinx.com/support/documentation/user_guides/ug364.pdf` (visited on Mar. 15, 2016).

[14] Xilinx, *7 Series FPGAs Configurable Logic Block User Guide*, Nov. 2014. UG474 v1.7. Available: http://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf (visited on Mar. 15, 2016).

[15] Xilinx, “Are Spartan-6, Virtex-6 and older devices supported in the Vivado design tools?,” Dec. 2012. AR# 53109. Available: http://www.xilinx.com/support/answers/53109.html (visited on Mar. 20, 2016).

[16] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, fourth ed., 2008.

[17] MIPS Technologies, *MIPS IV Instruction Set*, Sep. 1995. Revision 3.2. Available: http://www.cs.cmu.edu/afs/cs/academic/class/15740-f97/public/doc/mips-isa.pdf (visited on Mar. 16, 2016).

[18] Atmel, *8-bit Atmel with 8KBytes In-System Programmable Flash: ATmega8/L*, Feb. 2013. Rev. 2486AA-AVR-02/2013. Available: http://www.atmel.com/images/atmel-2486-8-bit-avr-microcontroller-atmega8_l_datasheet.pdf (visited on Mar. 22, 2016).

[19] ARM, *Cortex-M0 Devices Generic User Guide*, 2009. ARM DUI 0497A (ID112109). Available: http://infocenter.arm.com/help/topic/com.arm.doc.dui0497a/DUI0497A_cortex_m0_r0p0_generic_ug.pdf (visited on Mar. 22, 2016).

[20] STMicroelectronics, *ST200 VLIW Series ST231 Core and Instruction Set Architecture Manual*, Mar. 2004. ADCS 7645929A. Available: http://lipforge.ens-lyon.fr/docman/view.php/53/26/out.html (visited on Mar. 17, 2016).

[21] J.E. Smith and A.R. Pleszkun, “Implementing precise interrupts in pipelined processors,” *IEEE Transactions on Computers*, vol. 37, pp. 562–573, May 1988.

[22] M. Tan, “A Minimal GDB Stub for Embedded Remote Debugging,” Dec. 2002. Available: http://www.cs.columbia.edu/~sedwards/classes/2002/w4995-02/tan-final.pdf (visited on Mar. 16, 2016).

[23] Atmel, *Atmel AVR 8-bit Instruction Set Manual*, Jul. 2014. Rev. 0856J-AVR-07/2014. Available: www.atmel.com/images/atmel-0856-avr-instruction-set-manual.pdf (visited on Mar. 17, 2016).

[24] ARM, *Cortex-M0 Technical Reference Manual*, 2009. Revision r0p0. Available: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0432c/DDI0432C_cortex_m0_r0p0_trm.pdf (visited on Mar. 17, 2016).

[25] ARM, *CoreSight MTB-M0+ Technical Reference Manual*, 2012. Revision r0p1. ARM DDI 0486B (ID011213). Available: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0486b/DDI0486B_coresight_mtb_m0p_r0p1_trm.pdf (visited on Mar. 23, 2016).

[26] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, fifth ed., 2012.

[27] J. A. Fisher, P. Faraboschi, and C. Young, *Embedded Computing.* Morgan Kaufmann, 2005.

[28] J. W. van de Waerdt, *The TM3270 Media-processor.* PhD thesis, Delft University of Technology, the Netherlands, 2006.

[29] Q. Kong, "Interrupt Support on the ρ-VEX Processor," Master's thesis, Delft University of Technology, the Netherlands, 2011.

[30] A. A. C. Brandon and S. Wong, "Support for Dynamic Issue Width in VLIW Processors using Generic Binaries," in *Proc. Design, Automation & Test in Europe Conference & Exhibition*, (Grenoble, France), pp. 827 – 832, March 2013.

[31] "ST200 Micro Toolset Releases." Available: `http://ftp.stlinux.com/pub/tools/products/st200tools/index.htm` (visited on Feb. 12, 2016).

[32] D. Molka, D. Hackenberg, R. Schone, and M. S. Muller, "Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System," in *Parallel Architectures and Compilation Techniques, 2009. PACT '09. 18th International Conference on*, pp. 261 – 270, Sept. 2009.

[33] Opencores.org, *WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores*, Sep. 2002. Revision B.3. Available: `http://cdn.opencores.org/downloads/wbspec_b3.pdf` (visited on Mar. 19, 2016).

[34] ARM, *AMBA Specification*, 1999. Rev 2.0, ARM IHI 0011A.

[35] Cobham, *GRLIB IP Core User's Manual*, Jan. 2016. Version 1.5.0. Available: `http://www.gaisler.com/products/grlib/grip.pdf` (visited on Mar. 23, 2016).

[36] NXP Semiconductors, *LPC1311/13/42/43 User Manual*, Jun. 2012. Revision 5. UM10375. Available: `http://www.nxp.com/documents/user_manual/UM10375.pdf` (visited on Apr. 4, 2016).

[37] A. Malik, B. Moyer, and D. Cermak, "A Low Power Unified Cache Architecture Providing Power and Performance Flexibility," in *Low Power Electronics and Design, 2000. ISLPED '00. Proceedings of the 2000 International Symposium on*, pp. 241–243, 2000.

[38] Xilinx, *Command Line Tools User Guide*, Dec. 2009. UG628 v11.4. Available: `http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/devref.pdf` (visited on Apr. 7, 2016).

[39] Hewlett-Packard Company, "HP Labs : Downloads: VEX." Available: `http://www.hpl.hp.com/downloads/vex/` (visited on Feb. 12, 2016).

[40] "ALMARVI 621439." Available: `http://www.almarvi.eu` (visited on Apr. 22, 2016).

[41] J. S. P. Giraldo and A. L. Sartor and L. Carro and S. Wong and A. C. S. Beck, “Evaluation of Energy Savings on a VLIW Processor through Dynamic Issue-Width Adaptation,” in *2015 International Symposium on Rapid System Prototyping (RSP)*, pp. 11–17, Oct 2015.

[42] Q. Guo and A. Sartor and A. Brandon and A. C. S. Beck and X. Zhou and S. Wong, “Run-time Phase Prediction for a Reconfigurable VLIW Processor,” in *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 1634–1639, March 2016.

[43] K. Meun, “Fault Tolerance on the ρ-VEX Processor,” Master’s thesis, Delft University of Technology, the Netherlands, 2015.

[44] H. van der Wijst, “An Accelerator based on the ρ-VEX Processor: an Exploration using OpenCL,” Master’s thesis, Delft University of Technology, the Netherlands, 2015.

[45] J. Johansen, “Implementing Virtual Address Hardware Support on the ρ-VEX Platform,” Master’s thesis, Delft University of Technology, the Netherlands, 2016.

Glossary

arbitration *In the context of a bus:* the process of determining which master is allowed access to the bus, if multiple masters request access at the same time. 27

bitstream The configuration file for an FPGA. 7, 21, 50

block ram A dedicated memory component of a Xilinx FPGA. Block RAMs have two fully independent synchronous access ports, allowing two simultaneous accesses per clock cycle. In the Virtex-6 and Virtex-7 series, a single BRAM is 36 kib in size. 8, 73, 98

branch The operation of modifying the program counter (PC) register in a way other than incrementing it to point to the subsequent instruction. 9, 21, 44, 57, 70, 71, 73, 77

branch instruction An instruction that can modify the program counter (PC) register. 9, 11, 12, 21, 44

branch target The target of a branch instruction is the instruction that is to be executed after the branch if the branch is taken. 9, 11, 12, 69, 71, 72

breakpoint A breakpoint is an instruction address or line of source code that, when executed, causes the program to pause, to allow a software developer to inspect what the program was doing. 15–17, 22, 40, 41, 57, 71, 75, 110, 113

bundle A different name for a VLIW instruction, to disambiguate between syllables and bundles. A bundle is a set of syllables that are to be executed in parallel. xi, 19, 21, 22, 38–40, 42–44, 46, 47, 51, 53, 64–67, 69–72, 75, 77, 100, 102, 108, 110, 113

burst access A bus access requesting more than one word at a time. The words can then be transferred one every cycle, unaffected by latency. 23, 24, 27, 83

bus A data channel through which two or more devices can communicate with each other. 27–30, 41, 48–50, 78, 79, 81–83, 86, 87, 110, 111, 113

bus latency The time it takes for a request to propagate from a master to a slave, plus the time it takes for the reply from the slave to reach the master again. 27

bus width The number of physical wires in a bus dedicated to data. 27

bypass access *In the context of a cache:* a bypass access bypasses the cache entirely. Used for accessing peripherals or other addresses with often-changing values. 26, 50

cache A small but fast piece of memory that stores copies of results of previous accesses to a larger but slower memory, to prevent the same access from needing to be made again if the same memory is accessed in the near future. 4, 23–28, 30, 35, 42, 45–50, 53, 64, 74, 77–81, 83, 84, 91, 93, 96, 100, 102, 103, 106–109, 111, 113, 114

cache coherence Cache coherence is the requirement of a cache that a piece of data stored in the cache is either updated or invalidated when or shortly after another processor with its own cache writes to the data. 18, 26, 47, 48, 53, 103, 108, 111

cache consistency Cache consistency is the requirement of a cache that serviced memory accesses appear to be executed sequentially. That is, a write immediately followed by a read to the same address must return the written value. 26, 47, 49, 53, 103, 108

cache flush The action of marking all entries in a cache as invalid, effectively resetting the cache. 46, 49, 100

call instruction A special case of a branch instruction that unconditionally modifies the program counter (PC) register and stores the PC of the subsequent instruction in the link register. Used for function calls. 9, 20

configurable logic block Configurable logic blocks (CLBs) are the basic computational resource of an FPGA. They typically contain LUTs, registers and carry logic. 6

configuration word A specification of an ρ-VEX runtime configuration. 29, 34, 53, 107

context *In the context of software:* a context is the state of a running or halted thread, to be saved when switching to a different thread, and loaded when switching from a different thread. *In the context of the ρ-VEX processor:* the combination of all registers that constitute a software context. ρ-VEX contexts can be configured to run in parallel by reconfiguration. 31–35, 41, 46–48, 50, 53, 57–59, 61, 62, 70, 72, 74, 76–79, 90, 95, 100, 103, 104, 107, 109, 110, 112–114

context control register An ρ-VEX control register that is associated with and private to a certain context. The program counter is an example of such a register. 29, 30, 35, 36, 40, 41, 53, 57, 58, 71, 73, 74, 107, 108

critical path The critical path of a hardware design is the longest combinatorial path in terms of delay from the output of a register to the input of a register. This delay plus the register delays defines the minimum clock period, i.e., the maximum operating frequency for the design. 44, 82

critical section A critical section is a piece of code in a parallel program or a program with interrupts that may not be executed concurrently or interrupted. 37

debugging The process of finding the cause of a problem with a piece of software by inspecting the program while running it on the target hardware. 15

distributed RAM An alternative to BRAMs in Xilinx FPGAs. Unlike BRAMs, distributed RAM is spread out over the FPGA, which may allow for more efficient timing for small memories. Also unlike BRAMs, distributed RAM can be read asynchronously. 8, 30, 81, 88, 100

entity A VHDL construct containing a number of processes and entity instantiations to represent a logic block. 6, 7, 55, 56

external debugging In an external debugging environment, the to be debugged software runs on a different processor than the debugging software. The two are connected through some kind of physical link, such as JTAG or a serial port. The debugging software usually runs on a PC. 16, 40, 53, 89, 108, 110

fault A trap that is caused by a problem arising in the processor itself. 14, 36, 37, 71

field-programmable gate array An integrated circuit of which the hardware functionality can be reconfigured by reprogramming a bitstream. 2, 5, 32, 55, 107

forwarding The process of supplying computed data to the logic of preceding pipeline stages (executing later instructions) directly, to avoid having to delay later instructions that depend on the data until the data has been written to the register file. 13, 14, 21, 57, 73

frame pointer The value of the stack pointer at the start and end of a function. 10

fully associative *In the context of a cache:* a fully associative cache allows any memory location to be stored in any cache line. 24

generic A VHDL construct that serves as a configuration parameter for an entity. 7, 55

generic binary A software binary that can be run on multiple processors. In the context of the ρ-VEX, it is a binary that can run in eight-, four- or two-way mode. 22, 23, 38, 39, 44, 67, 69–71, 100, 103, 111

global control register An ρ-VEX control register that is global to the entire processor, as opposed to being related to a specific context. 29, 30, 53, 57–59, 73, 74, 107

GRLIB A library of mostly open-source HDL components developed by Aeroflex Gaisler [10]. 6, 41, 42, 81, 97

hazard A hazard is a dependency that has not been met yet in a pipelined processor, requiring the pipeline to be stalled to avoid incorrect execution. 12

hit *In the context of a cache:* a cache hit means that the requested memory was found in the cache, preventing the need for a main memory or next level cache access. 24, 25, 79, 81

host *In the context of debugging:* the program and/or the processor that is being debugged. 16, 40, 41, 86, 87

immediate A constant operand, embedded in the instruction word. 20

interrupt A trap that is requested by a source external to the processor, such as a peripheral. 14, 21, 22, 28, 33, 36–38, 40, 53, 71, 76, 88, 95, 103, 104, 108

invalidation The process of preventing an instruction to affect the state of the program in any way. 15, 22, 39

ISE An FPGA design suite by Xilinx. Supports all Xilinx FPGAs, but does not support high-level synthesis. 55

issue Verb. The process of starting execution of an instruction. 10

issue width of a VLIW processor: the number of instructions that can be issued in a single cycle. 19, 20, 30, 50, 51

JTAG A serial protocol intended for debugging hardware. 16, 41, 42

jump instruction A special case of a branch instruction that unconditionally modifies the program counter (PC) register. 9, 11, 14, 20, 36

lane The part of a VLIW that executes a syllable. VLIWs have one lane for each syllable that it can execute in parallel. 19, 21, 30, 32, 34, 35, 44, 45, 50, 51, 56–58, 62, 64, 66–70, 73–77, 98, 113

lane group A group of lanes in a runtime reconfigurable ρ-VEX processor that cannot be split over multiple programs. 30–35, 45–51, 53, 59–62, 64, 66–68, 73, 78, 79, 83, 95, 96, 107–111, 114

line *In the context of a cache:* a cache line is a piece of cache memory containing cached data and a tag specifying which memory address the data corresponds to and whether that data is up-to-date. 24, 25, 46, 48–50, 65, 66, 78, 79, 81, 102, 113

link register The register that stores the return address of a function call. 9, 10, 36

load instruction An instruction that copies data from memory to a register file. 9

long immediate A syllable in the ρ-VEX that supplies the upper 23 bits of an immediate to another syllable. It does not on its own perform any function. 20, 21, 39, 44, 51, 57, 66–68, 71, 95

master *In the context of a bus:* a device that initiates transfers with slaves on the bus. 27, 28

mirror *In the context of memory:* an address that maps to the same physical resource as another address. 27

miss *In the context of a cache:* a cache miss means that the requested memory was not found in the cache, meaning that the main memory or next level cache needs to be queried. 24, 25, 79, 102

ML605 A Virtex-6 FPGA development board manufactured by Xilinx. 8, 41, 106, 108, 109

ModelSim A toolchain by Mentor Graphics that allows VHDL, Verilog and SystemC hardware descriptions to be simulated.. 93

nibble Half a byte, i.e., 4 bits. 34

no-write allocate *In the context of a cache:* a no-write allocate cache does not allocate space for the written memory in the case of a write miss, and instead only forwards the write to the next level cache or memory. 25, 26

non-associative *In the context of a cache:* an non-associative cache maps a memory location to only one specific line. 24, 25

operating system A piece of software that manages system resources and provides services to applications running on it. 16, 40

page fault A fault caused by a virtual memory access or instruction fetch that is either not mapped to physical memory or is not allowed to be accessed in that way. 22, 27

panic handler The panic handler is a secondary trap handler in the ρ-VEX processor. It is used instead of the regular trap handler when the software running on the processor is not in a state that can be interrupted without losing information needed to return to the interrupted program. 36, 37, 40, 53, 71, 108

peripheral A device that connects to a processor to perform a special function, such as communicating with the outside world. 28, 29, 50, 88, 110

pipeline The pipeline of a processor is a set of stages of instruction execution that are processed sequentially in their own clock cycles, while at the same time a new instruction can be issued every cycle. 10–12, 18, 21, 44, 51, 57, 62, 68, 70, 72, 74–76, 110

preciseness *In the context of traps:* a precise trap is a trap where A) all instructions up to and optionally including the instruction at the trap point completely and correctly execute, B) all instructions following and optionally including the instruction at the trap point do not modify the state of the program, and C) the trap point points to exactly the instruction that caused the trap, if applicable [21]. 15, 22, 27, 110

process A VHDL construct containing a number of sequential statements that describe the functionality of a logic block. 6, 7

program counter The address of the current instruction being executed by a processor. 9, 29, 57, 110

reconfiguration The process of changing the configuration of a reconfigurable processor in order to allow it to more efficiently execute the task at hand. 21, 23, 30, 32–35, 38, 41, 45–47, 49, 53, 57–59, 61, 62, 72, 74, 93, 95, 100, 102, 103, 107–110, 112, 114

register file A piece of local data memory in a processor that has greater connectivity and is faster than main memory. 9, 10, 12, 13, 16, 20–22, 31, 35, 53, 57, 73, 74, 107, 109, 110

replacement policy *In the context of a set associative cache:* the algorithm used to determine which set is to be updated when a miss occurs. 25, 46–48

return instruction A special case of a branch instruction that unconditionally sets the program counter (PC) register to the link register. Used to terminate a function and return to the caller. 9, 10

self-hosted debugging In a self-hosted debugging environment, the to be debugged software runs on the same processor as the debugging software. The operating system provides the interface between the two processes. 16, 40, 41, 53, 108, 110

set *In the context of a set associative cache:* each set has exactly one line that corresponds to a given address. Sets are queried in parallel when a request is made to determine which set contains the requested value, if any. 24, 25, 46, 47, 111

set associative *In the context of a cache:* an n-way set associative cache maps a memory location to one specific line within each of its n sets. 24, 25, 46, 108, 111

signal A VHDL construct representing a physical connection or wire. 6, 7

slave *In the context of a bus:* a device that responds to transfers initiated by masters on the bus. 27, 28, 78, 110

slice A component used in Xilinx FPGAs that contains LUTs, registers and carry logic. Virtex-6 and Virtex-7 series FPGAs have two slices per configurable logic block. 8, 98

snooping *In the context of caches:* the process of monitoring writes made by other processors in a multiprocessing system, to invalidate or update the cache entries of the written memory. 26, 48, 49, 111

spatial locality A phenomenon exhibited by programs, stating that if a program uses a bit of data, it is likely to use data at nearby addresses in the near future. 24

stack First-in last-out data structure used to save and restore local function data. 10, 16, 36, 103

stack corruption The inadvertent process of modifying the stack causing a program to fail. 10

stack pointer Pointer to the ‘top’ of the stack, used to store how large the stack is. 10

stall Verb. The process of suspending execution of instructions while waiting for a dependency to be resolved. 12, 79, 81

stop bit A bit encoded in each syllable of a supporting VLIW architecture. The stop bit being set marks that it is the last syllable in the bundle. 19, 22, 23, 38, 43, 44, 53, 57, 108, 110

store instruction An instruction that copies data from a register file to memory. 9

syllable Word encoding a single operation. Multiple syllables make up an instruction bundle. 3, 19–21, 38, 39, 41–44, 47, 51, 56, 57, 64, 66–71, 74, 76, 77, 79, 94, 95, 100–102, 110, 111, 113

syscall A 'function call' from an application to the operating system. Normally implemented as a trap caused by a special instruction. 22

SystemC A hardware description language based on the C programming language. 6, 55, 91, 107, 108

tag *In the context of a cache:* a cache tag the part of a cache line that specifies which memory address the line corresponds to and whether the data is valid. 24, 25, 46, 48, 49, 79, 81

target *In the context of debugging:* the program and/or the processor that is being debugged. 16, 40, 86, 90

temporal locality A phenomenon exhibited by programs, stating that if a program uses a bit of data, it is likely to use it again in the near future. 24

testbench A piece of typically non-synthesizable HDL code that simulates the external environment of the UUT. 7

thread A task within a program that can execute in parallel to other tasks in a multiprocessing environment. 17, 35

timer A peripheral that generates an interrupt periodically or a certain amount of time after being triggered. 28

toplevel entity The VHDL entity that represents the design as a whole. 7

trap A trap is a transient condition that prevents a program from continuing normally. Normally handled by branching to a trap handler. 14–16, 21, 22, 27, 35–40, 53, 57, 66, 68, 70–77, 93, 95, 102, 107, 108, 110

trap argument One of the two trap identification registers in the ρ-VEX, the other being the trap cause. The significance of this 32-bit value is based on the value of the trap cause register. 35, 38, 53, 71, 76, 107

trap cause One of the two trap identification registers in the ρ-VEX, the other being the trap argument. This 8-bit register specifies the index of the trap that occurred. 35, 38, 53, 107

trap handler Also known as a trap service routine. A trap handler is a special function that performs the appropriate actions to resolve a trap such that normal program execution can be resumed, or alternatively signals an error if this is not possible. 14–16, 35–40, 53, 71, 75, 76, 104, 107, 108, 110

trap point The program counter (PC) of the instruction that was interrupted due to the occurrence of a trap. 14–16, 29, 36, 39, 67, 71, 77, 110

universal asynchronous receiver/transmitter A peripheral that communicates with another device serially using the RS232 protocol. 28, 41, 85, 108

VC707 A Virtex-7 FPGA development board manufactured by Xilinx. 8, 41, 55, 106, 109

Verilog A hardware description language. 6, 55, 107

VHDL The hardware description language that is used in this work. 6, 52, 55, 93, 107

Vivado An FPGA design suite by Xilinx supporting high-level synthesis. Only supports 7-series Xilinx FPGAs. 55

watchpoint A watchpoint is a data address that, when accessed, causes the program to pause, to allow a software developer to inspect what the program was doing. 16, 17, 41, 57, 71, 110, 113

word *In the context of a bus:* a datum that is defined to use the same amount of bits as the bus width. 27

write allocate *In the context of a cache:* a write allocate cache allocates space for the written memory in the case of a write miss, similar to a read miss. 25, 26, 49

write buffer A piece of hardware in a cache that stores a pending write request to the memory or next level cache, to allow the processor to continue executing before the write is complete. 25, 35, 47, 49, 50, 80, 111

write-back *In the context of a cache:* a write-back cache deals with writes by only saving the written value in the cache and setting a flag in the tag indicating that the value has been modified. The memory or next level cache is only updated when the line is removed from the cache. 25, 26

write-through *In the context of a cache:* a write-through cache deals with writes by saving the written value in the cache as well as to the memory or next level cache. 25, 26, 49, 111

A
Paper: context switching

The following paper was presented at the 2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig).

Multiple Contexts in a Multi-ported VLIW Register File Implementation

Joost Hoozemans, Jens Johansen, Jeroen van Straten, Anthony Brandon, Stephan Wong

Computer Engineering Lab, Delft University of Technology, The Netherlands
Email:{j.j.hoozemans, a.a.c.brandon, j.s.s.m.wong}@tudelft.nl
{j.johansen, j.vanstraten}@student.tudelft.nl

***Abstract*—The register file is an expensive component in the design of any processor, especially, when considering the additional ports that are needed to support multiple datapaths within a wide-issue VLIW processor. In a recent work, these additional resources were used to dynamically reconfigure the register file to support a dynamically reconfigurable VLIW core. The design can be perceived as a single 8-issue, two 4-issue, or four 2-issue VLIW cores. Consequently, the multi-ported design can operate in different modes, namely as one, two, or $four$ register files, respectively, corresponding to the active number of cores. The implementation of the register file design on FPGAs using Block RAMs still results in unused resources due to the coarseness of the Block RAMs.**

In this paper, we propose to re-purpose these unused BRAM resources to additionally support multiple contexts next to earlier-mentioned modes. In this manner, the 8-issue, 4-issue, and 2-issue cores have access to 4, 2, and 1 contexts, respectively. Consequently, we can avoid saving and restoring of the task states in a multi-task environment, turning context switching from a traditionally time-consuming event to an almost instantaneous event. The advantage of this is the reduction of interrupt latency and task switching latency, which are important in real-time and embedded systems.

Our results show that our technique can improve interrupt latency by a factor of $17.4\times$ compared to using a software register spill routine, depending on the behavior of the memory system. Likewise, the task switching time can be improved by $6.7\times$.

I. Introduction

The ρ-VEX processor [1] is a dynamically reconfigurable VLIW processor that can adapt its organization to the requirements of different workloads. One of its most important run-time parameters is the issue-width that allows for adaptation towards the ILP of the task(s) at hand. The design can be configured as a single 8-way (1 × 8-way), two 4-ways (2 × 4-way), four 2-way (4 × 2-way) VLIW processor core(s), or combinations of those: e.g., two 2-ways and one 4-way. This capability requires the design of an extensive register file to support these different modes. In the worst case, the register file must provide:

- 8 write ports and 16 read ports when running in the 1×8-way mode
- 4 architecturally separate register files when running in the 4×2-way mode

To design a register file that satisfies these requirements we use techniques such as Block RAM (BRAM) duplication and a Live Value Table (LVT), which we will discuss in Section II.

A major drawback of the current design is the large resource utilization. The BRAMs used to implement the register file on the FPGA need to be duplicated multiple times to provide the necessary amount of read and write ports. Every BRAM has a capacity of 512 32-bit words (2KiB); however, the architecture only requires 64 32-bit registers. Because of this, the resulting design has an enormous storage capacity of which at most an eighth is used by the processor in any particular configuration.

The design presented in this paper aims to convert the drawback of the high BRAM usage of the register file for wide-issue VLIW softcore processors into an advantage by using the overcapacity to store different execution contexts. The actual utilization of the BRAM storage capacity will increase from $\frac{1}{8}$ to $\frac{1}{2}$. Support for multiple contexts in hardware relieves the core from having to spill and restore its entire register file contents to and from memory in the event of a task switch or interrupt. In a multi-tasking environment, this concept changes task switches, which are traditionally very time-consuming, into a virtually instantaneous event. Faster context switching has advantages in numerous computing scenarios, as it will increase responsiveness for interactive workloads and improve interrupt latency and task switching speeds in real-time systems. In the following, we illustrate several cases in which our work can improve performance:

- Frequently used threads: Kernel threads, like schedulers, must be frequently executed. In a traditional core implementation, timers interrupt the core and trigger context switching in order to execute such threads. In our work, these threads can be maintained within the core and thereby remove the need for context switching. For example, an application is executing in the 8-issue mode using 1 out of 4 contexts. When the scheduler needs to execute, the current thread can be scheduled to run on a 4-issue core - this mode switch only takes several cycles when using generic binaries [2]. In the remaining 4-issue core, the execution of the scheduler can be resumed by using its own context that remained "dormant" within the core.
- Dynamic switching of execution by different cores: When threads require more resources, e.g., when their ILP

This work has been supported by the Almarvi European Artemis project nr. 621439.

978-1-4673-9406-2/15/$31.00 ©2015 IEEE

increases, our processor design allows for it to claim additional datapaths to execute the code more efficiently. This does mean that another thread must be stalled for a while. However, in our case, the context of the second thread does not need to be saved into the memory and can remain within the core until it is resumed. In the latter, another context switching operation is saved.

- Context-cycling after cache misses: When our processor is running in the 8-issue (4-issue) mode, it can have up 4 (2) contexts stored within each core. This means that when one thread is encountering a cache miss, thus execution is stalled, the core can easily switch to another thread (context) and continue execution, i.e., Switch-on-Event Multi-Threading SoEMT.
- Embedded real-time systems with multiple tasks that require stringent real-time constraints (e.g., control loops with sensors and actuators). A single core can process more events using multiple contexts [3]. Therefore, a softcore can be used as microcontroller on an FPGA which would save the designer from having to design hardware circuits to handle some events or having to resort to a multi-core system where distinct events are handled by a dedicated core.

The register file of our ρ-VEX is a complex topic, as it is also instrumental in supporting the core's dynamic reconfigurability [4]. We limit the scope of this paper to evaluating the benefits from multiple hardware contexts. It must therefore be noted that the costs of this design (see Table I) are paid not only for multiple contexts, but also to support the dynamic reconfigurability. Our approach in this paper gives us a 17.4× reduction in interrupt latency and 6.7× reduction in context switching time.

II. Background

The multi-ported register file is a challenging component in the design of softcore VLIW processors. Wide-issue VLIW processors like the ρ-VEX need register files with a large number of read and write ports. The VEX instruction set architecture (ISA) supports operations that use two source registers and one destination register. Because of this, the number of write ports required is equal to the issue-width, and the number of read ports is equal to twice the issue-width. Creating such complex register files using FPGA LUT resources is very expensive and scales very poorly with the number of ports. The reconfigurable ρ-VEX design and the implementation of its multi-ported register file are introduced in [5]. Moreover, in [6], the idea of using a Live Value Table (LVT) is discussed that enables the use of banked memories with duplication to create multi-ported BRAM memories. The ideas presented in this paper are built upon a register file design that is implemented using this technique. We will discuss the concepts and challenges briefly in this section.

Creating RAM memories that have more read ports is straightforward and achieved by duplicating the BRAM and writing data into each block simultaneously. In this way, each BRAM contains the same data, and their read ports can be used independently of each other. Increasing the number of write ports, however, is more difficult. Several solutions exist in literature. The simplest solution is to divide the register file into banks, each connected to one of the write ports [7]. This solution restricts the range of registers each write port can write to and thus reduces the freedom the compiler has to schedule instructions. Another solution introduced in [8] increases the size of each bank to the original register file size and renames the registers in between the compiler and assembler. This solution enables a banked design with the same scheduling freedom as an actual multi-ported register file but utilizes a multiple of the number of registers. Note that this technique does not necessarily require more BRAMs since their size is a lot larger than the 64 registers specified in the VEX ISA. It does, however, increase the number of bits required to specify the source and destination registers in instructions.

The register file used in the ρ-VEX uses the technique introduced by [6]. This scheme also duplicates the register file for each write port. However, instead of uniquely naming the registers in each bank, a Live Value Table (LVT) keeps track of which bank holds the most recent value of each register. It uses this information to multiplex the right bank to the read ports, as shown in Figure 1. The LVT needs to be implemented as a multi-ported LUT based RAM because it still needs one write port per register file write port. However, since it only needs to hold a bank address, it is much narrower than the original register file that the scheme seeks to replace. While this technique enables the register file to be implemented mostly with BRAMs instead of LUTs, it still scales poorly with the number of ports. The number of BRAMs required is equal to the product of the number of read and write ports. The depth of the LVT scales linearly with the number of registers in the register file while the width scales logarithmically with the number of write ports. The number of ports required for the LVT is equal to the number of ports on the register file.

III. Related work

In [9] the authors analyzed the high requirements that wide-issue VLIW processors pose on the register file. They discuss hypothetical FPGA primitives similar to existing BRAMs but featuring many more read and write ports. These primitives do not exist in current FPGAs, therefore, the use of large BRAM or LUT-based structures is required to emulate this behavior [6].

In [10], it is stated that "the context switch time is one of the most significant overhead factors in any operating system" and shows that high timer interrupt handling latency can impede schedulability of real-time tasks. In [3], it is measured that using a multi-threaded architecture with 4 register sets allows an autonomous guided vehicle to run at a 28% higher velocity. In [11], measurements were performed to quantify the interrupt latency of several embedded Linux distributions running on a Xilinx Microblaze.

There are numerous examples of processors which use the concept of multiple register files to enhance the context

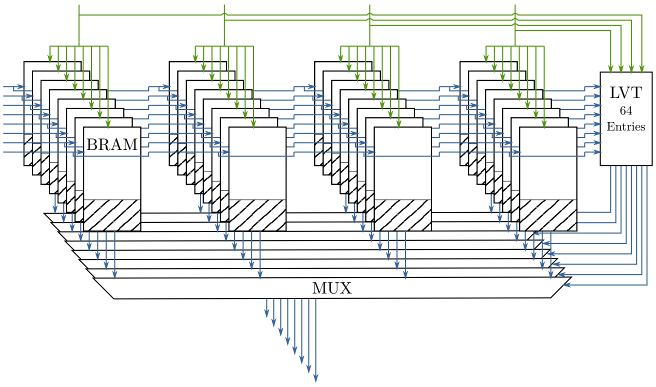


Figure 1. Block diagram of register file implementation using multiple banks of BRAMs. The green arrows indicate write ports, while the blue arrows indicate read ports. The shaded area represents the portion of the BRAM used for storing a single context.

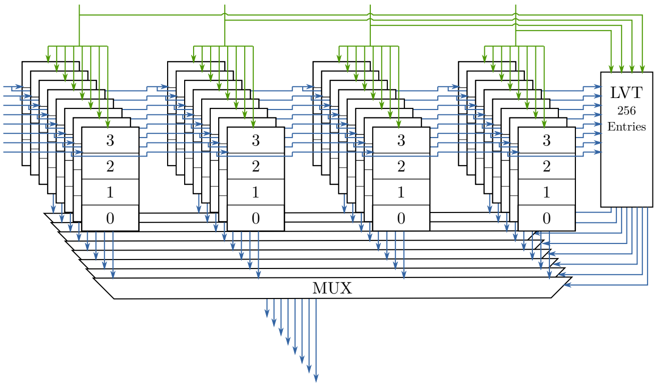


Figure 2. Block diagram of register file implementation supporting multiple contexts. Here the number of BRAMs is the same, but the LVT is larger.

switching time and interrupt latency in hardware. In [12], comparisons are made (by means of simulations) between increasing the number of cores and increasing the number of register sets in terms of increasing performance for a parallel workload. In [13], the MIPS architecture is extended by duplicating the register file multiple times and adding special instructions to switch between them when a context switch is required. In [14], the authors propose a novel architecture, which also supports holding multiple contexts in hardware simultaneously, and extend it with a dedicated cache to hold contexts to prevent spilling to main memory. Among other things the effects of the additional contexts on interrupt latency is investigated. Storing multiple contexts is also a requisite for (Simultaneous) Multi-Threading (SMT) [15]. An example of a VLIW processor with SMT support is the Itanium [16]. These technologies target high-end ASIC processors while this work targets the embedded (FPGA) domain.

The synthesizable ARPA-MT [17] and RTBlaze [18] processors also use SMT to improve schedulability and performance for embedded real-time systems. However, all the resource investments in this core are only used for SMT. The ARPA-MT core has a single execution pipeline. The fetch and decode circuits as well as the register file need to be duplicated for each thread slot.

In contrast, the ρ-VEX uses the additional resources to support: 1) a very wide VLIW to exploit ILP, 2) multiple hardware contexts and 3) a multi-core configuration (in other words, all contexts can be active and executing at the same time). Therefore, it uses the additional resources in a more efficient way compared to the previous work.

IV. Implementation

Figure 1 shows the implementation of a register file with four write ports and eight read ports ($4W \times 8R$), using BRAMs and an LVT. The $8W \times 16R$ version would be 4 times as large. The hatched area represents the part of the BRAM that is actually used to store the 64 registers used by the ρ-VEX. The figure shows that a large part of the BRAMs is unused. Because the ρ-VEX can be configured as four independent processors, it also needs four separate register files. However, the total number of read and write ports is the same for one large 8-issue processor or four separate 2-issue processors. Because of this characteristic, the same multi-ported register file can be used in each configuration. The number of registers, however, needs to be quadrupled, for a total of 256 registers, since each core needs a separate register file of 64 registers. The BRAM resources on contemporary FPGA boards provide more than sufficient storage capacity to accommodate this, so there is no added cost in BRAM resources. However, the LVT does need to increase in size, to keep track of the most recent location of all 256 registers.

Figure 2 shows how the multiple contexts can be stored in the previously unused space of the BRAMs. Creating four separate register spaces is a necessary cost to enable the ρ-VEX to be split into four separate processors. However, not all of the register spaces are used when the core is configured as a single 8-issue processor or two 4-issue processors. This creates the opportunity to re-purpose these unused register spaces as alternative register windows, which can be used to store the register context of inactive processes. Since the four register windows are implemented as a larger continuous address space, the uppermost bits can be used to select one of the four register windows.

The ρ-VEX utilizes more registers than just the 64 general purpose registers. It also has the following registers, that must be stored for a context switch:

1) A special 32-bit register used to store the return address for a function call (the link register).
2) Eight 1-bit registers used for conditional branching.
3) The program counter.
4) Various control registers, used for example for interrupt handling.

These registers cannot easily be stored in BRAMs, as the control logic needs to be able to access all these registers at once. Therefore, these registers are implemented in LUTs. To support running as 4×2-issue processors, all these registers need to be duplicated as well, and can thus be used as part of

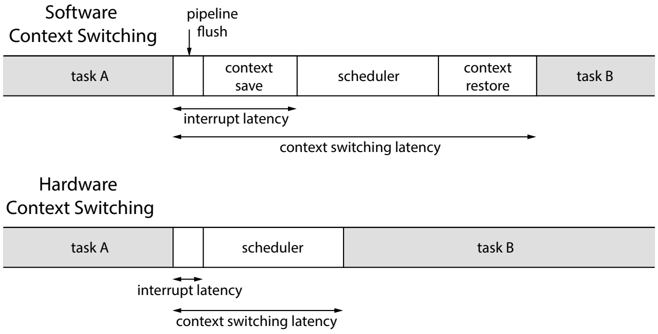


Figure 3. Context switching and interrupt latency definition.

the hardware contexts. Some additional hardware is required to use these registers for context switching, as not every lane would necessarily need access to all duplicates of the registers for reconfiguration only, while this is necessary for context switching. However, when this is done, the only registers which need to be spilled and restored are those registers which are used by the context switching routine, or scheduler itself. Because the additional hardware cost is small, our context switching design incorporates this feature.

A hardware context switch is not entirely free in terms of cycles in the current ρ-VEX design. To avoid complicating the forwarding logic, context switches are only possible when the pipeline is empty. Because the ρ-VEX has a five stage pipeline, five cycles are needed to flush the pipeline before a context switch can occur. In addition, the context switches are currently controlled by the dynamic reconfiguration controller, which takes three additional cycles to decode and commit a new configuration. Two of these are spent still executing instructions in the old context.

V. Experimental Setup

Our measurements are carried out using the ρ-VEX VLIW softcore processor clocked at 37.5 MHz running on a Xilinx ML605 development board, which incorporates an XC6VLX240T Virtex 6 FPGA. We use a timer connected to the interrupt request input of the processor to generate interrupts at different rates to measure the impact of our approach on the performance of the system.

We quantify the impact on performance by measuring two different values, namely:

1) *Interrupt Latency*: The number of cycles elapsed between the moment an interrupt request is received by the core, and the first instruction of the interrupt handler being executed.
2) *Context switching latency*: The number of cycles elapsed between the moment a context switch is requested (due to an interrupt), and the first instruction being executed in the new context.

Figure 3 shows what these latencies are made up of, namely: pipeline flushing, saving context registers, running the interrupt service routine (in our case the task scheduler), and finally restoring the context registers. By using hardware contexts the latency of saving and restoring registers can be eliminated. We measured these quantities by creating a workload of four programs. At every timer interrupt a scheduler selects a different program to execute, and performs the context switch to that program. The programs themselves have no impact on the measurements, since they are purely dependent on the time it takes to save and restore all context registers.

In order to measure the difference between hardware and software context switching, we wrote a software and a hardware context switching routine. The software version saves the complete context to the stack of the currently running task, stores the stack pointer to a predefined memory location, and starts executing the interrupt handler. The interrupt handler then calls the scheduler in order to schedule the next task. The current stack pointer is then replaced with the stack pointer of the new task. Next, the application context of the newly selected task is restored from the stack, after which control is handed back to the application. The hardware switch routine does not need to save or restore all registers. Instead it only has to do so for the registers used by the interrupt routine, in this case the scheduler.

The scheduler utilizes a linked list in memory to determine which task to switch to; each entry representing a task, with a mapping to another task. When a task completes, the linked list is rebuilt such that the context switching code does not switch back to the completed task, and a context switch is requested immediately using a software trap instruction. When the last task completes, it signals completion to the platform.

Because cache behavior will impact the latencies for saving and restoring the contexts we perform the measurements for different memory access latencies. We measure using latencies from 0 (single cycle memory access) to 30 cycle memory access on cache miss. The cache itself consists of a separate instruction and data cache, respectively 32KiB and 8KiB in size. The size has intentionally been kept small, because the programs under test had to be small as well for the entire memory to fit on the FPGA; it is assumed that, under normal circumstances, larger caches will be used, but the running programs will also use wider regions of more memory. Both caches have single-cycle hit latency for reads. The data cache has a two-cycle latency for writes for both hits and misses, as long as one of the four write buffers is vacant.

To evaluate the context switching overhead in multi-process time-sharing systems, overall performance of the multi-task system is tested on hardware using the cached system. The timer is used to generate an interrupt at a fixed frequency, often referred to as the system "tick," in which a context switch is performed. Clearly, the context switching overhead is directly related to the frequency of the system tick [10]. The frequency of the tick is usually in the order of 50 to 1000 Hz. A lower frequency will lead to lower switching overhead, but higher frequencies will result in a more responsive system. Systems that require more responsiveness will therefore have a higher tick frequency. For example, the Linux kernel uses a system tick of 1000 Hz for desktop systems, but this can be reduced to 100 Hz for server systems to reduce overhead. On the other

Table I
RESOURCE USAGE OF REGISTER FILE WITH AND WITHOUT SUPPORT FOR MULTIPLE CONTEXTS.

	Register File		Core	Increase over Core
	1 Context	4 Contexts		
Slice Registers	806	1392	8529	6.9%
Slice LUTs	10764	15591	35148	13.7%
RAMB18E1	128	128	147	0%
RAMB36E1	0	0	128	0%

hand, the Windows kernel uses 66 Hz. The frequency is varied between tests to evaluate its effect. In addition, the system is evaluated with varying bus latencies. The latencies used are estimates of what the average latency would be for a real off-chip memory system.

A cycle counter available within the ρ-VEX processor is used to measure the time from system reset to the program completion signal, which is given by the task switching implementation when all tasks have completed. For each timer and memory system configuration, both context switching implementations are evaluated. Because all other factors are kept constant, the difference in total execution time is only dependent on the context switching overhead. The speedup between the baseline and hardware context switching implementations is then determined to quantify this overhead.

VI. RESULTS

In Table I we show the increase in resource utilization of the register file when adding support for four contexts. As expected the number of BRAMs used does not increase. Only the number of registers and LUTs increases, since these are used to implement the LVT. While these increases seems large, when compared to the total usage of the core they are less significant. Additionally, note that this increase in resources in the register file is required to support the dynamic reconfigurability of the processor.

As we can observe in Table II, the interrupt latency is 87 cycles for software context switching. The interrupt latency when using hardware contexts is only 5 cycles, solely due to the pipeline flush performed by the trap handling logic. A full context switch, i.e., the time between a tick interrupt request and the execution of the first instruction in the new context, takes 174 cycles using the software implementation, compared to 26 cycles using the hardware contexts.

Table II
INTERRUPT AND CONTEXT SWITCHING LATENCY WITH SINGLE-CYCLE MEMORIES IN CYCLES.

	Software	Hardware	Reduction
Interrupt Latency	87	5	17.4×
Context Switch Latency	174	26	6.7×

In Table III, we can observe the results of the same experiments run using a cached memory system, with a bus latency of 20 cycles. We observe that the improvement due to the hardware context switching is greater in this system, with the improvement in interrupt latency increasing from 17.4 to 23.5×, and the improvement of context switching time increasing from 6.7 to 14.8×.

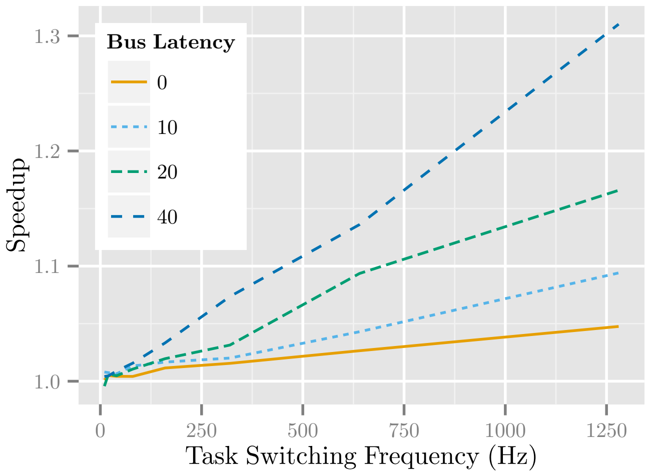


Figure 4. Speedup of the multi-task system due to the hardware context switching implementation.

Table III
INTERRUPT AND CONTEXT SWITCHING LATENCY WITH CACHE AND 20 CYCLES BUS LATENCY IN CYCLES.

	Software	Hardware	Reduction
Interrupt Latency	16798	713	23.5×
Context Switch Latency	31861	2148	14.8×

Figure 4 shows the speedup for different frequencies of the timer tick parameterized for different memory latencies, as measured on hardware using the cached system. It can be seen that in the region of higher task switching frequencies the difference between hardware and software context switching can be quite substantial depending on the memory system. A speedup of over 1.3× can be achieved for a bus latency of 40 cycles at a switching frequency of 1280 Hz.

VII. CONCLUSIONS

The concept of using additional register files to speed up multi-threading performance has been applied in numerous designs in the past. In this paper, we apply the concept to an existing design, exploiting the overcapacity of the BRAMs in the existing implementation of the multi-ported register file and the additional logic required by the parameterized reconfigurability of the ρ-VEX softcore. We have demonstrated that the proposed design can decrease the interrupt latency by a factor of over 20 times in a realistic environment. Likewise, the total context switching time can be decreased by a factor of over 10 times. In a simple multi-task system the effect of this is apparent as the decrease in overhead results in a speedup of 1.3× in the most extreme case evaluated. For applications with few real-time requirements, where the system tick frequency would be relatively low, the speedup is negligible, as the task switching code would not be executed as often. However, embedded real-time systems that need to process large numbers of events will benefit most from the improvements.

References

[1] S. Wong and F. Anjam, "The Delft Reconfigurable VLIW Processor," in *17th International Conference on Advanced Computing and Communications*, 12 2009, pp. 244–250.

[2] A. Brandon and S. Wong, "Support for dynamic issue width in VLIW processors using generic binaries," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, March 2013, pp. 827–832.

[3] U. Brinkschulte, C. Krakowski, J. Kreuzinger, and T. Ungerer, "Interrupt service threads-a new approach to handle multiple hard real-time events on a multithreaded microcontroller," *RTss WIP sessions, Phoenix*, pp. 11–15, 1999.

[4] F. Anjam, M. Nadeem, and S. Wong, "Targeting code diversity with run-time adjustable issue-slots in a chip multiprocessor," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, March 2011, pp. 1–6.

[5] S. Wong, F. Anjam, and F. Nadeem, "Dynamically Reconfigurable Register File for a Softcore VLIW Processor," in *Design, Automation Test in Europe Conference Exhibition*, March 2010, pp. 969–972.

[6] C. LaForest and J. Steffan, "Efficient Multi-ported Memories for FPGAs," in *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '10. ACM, 2010, pp. 41–50.

[7] M. Saghir and R. Naous, "A Configurable Multi-ported Register File Architecture for Soft Processor Cores," in *Reconfigurable Computing: Architectures, Tools and Applications*, ser. Lecture Notes in Computer Science, vol. 4419, 2007, pp. 14–25.

[8] F. Anjam, S. Wong, and F. Nadeem, "A Multiported Register File with Register Renaming for Configurable Softcore VLIW Processors," in *International Conference on Field-Programmable Technology (FPT), 2010*, Dec 2010, pp. 403–408.

[9] M. Purnaprajna and P. Ienne, "Making Wide-issue VLIW Processors Viable on FPGAs," *ACM Trans. Archit. Code Optim.*, vol. 8, no. 4, pp. 33:1–33:16, Jan. 2012.

[10] G. Buttazzo, *Hard Real-time Computing Systems: Predictable Scheduling Algorithms and Applications*. Springer, 2011, vol. 24.

[11] A. Ronnholm, "Evaluation of Real-Time Operating Systems for Xilinx MicroBlaze CPU," Master's thesis, Malardalens University, 6 2006.

[12] R. Thekkath and S. Eggers, "The Effectiveness of Multiple Hardware Contexts," *SIGOPS Oper. Syst. Rev.*, vol. 28, no. 5, pp. 328–337, Nov. 1994.

[13] N. Rafla and D. Gauba, "Hardware implementation of context switching for hard real-time operating systems," in *2011 IEEE 54th International Midwest Symposium on Circuits and Systems (MWSCAS)*, , Aug 2011, pp. 1–4.

[14] K. Tanaka, "PRESTOR-1: a Processor Extending Multithreaded Architecture," in *Innovative Architecture for Future Generation High-Performance Processors and Systems, 2005*, Jan 2005.

[15] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," in *ACM SIGARCH Computer Architecture News*, vol. 23, no. 2. ACM, 1995, pp. 392–403.

[16] R. Riedlinger, R. Bhatia, L. Biro, B. Bowhill, E. Fetzer, P. Gronowski, and T. Grutkowski, "A 32nm 3.1 billion transistor 12-wide-issue itanium processor for mission-critical servers," in *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2011 IEEE International*, Feb 2011, pp. 84–86.

[17] A. Oliveira, L. Almeida, and A. de Brito Ferrari, "The arpa-mt embedded smt processor and its rtos hardware accelerator," *Industrial Electronics, IEEE Transactions on*, vol. 58, no. 3, pp. 890–904, March 2011.

[18] T. P. Wijesinghe, "Design and implementation of a multithreaded softcore processor with tightly coupled hardware real-time operating system," Master's thesis, 2008. [Online]. Available: http://search.proquest.com/docview/250936948?accountid=27026

B Paper: sparse instruction encoding

The following paper was also presented at the 2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig).

A Sparse VLIW Instruction Encoding Scheme Compatible with Generic Binaries

Anthony Brandon*, Joost Hoozemans†, Jeroen van Straten‡, Arthur Lorenzon§, Anderson Sartor¶,
Antonio Carlos Schneider Beck||, Stephan Wong**

Computer Engineering Lab
Delft University of Technology
Email: {a.a.c.brandon*, j.j.hoozemans†, j.s.s.m.wong**}@tudelft.nl,
j.vanstraten@student.tudelft.nl‡

Institute of Informatics
Universidade Federal do Rio Grande do Sul
Email: {alorenzon§, alsartor¶, caco||}@inf.ufrgs.br

***Abstract*—Very Long Instruction Word (VLIW) processors are commonplace in embedded systems due to their inherent low-power consumption as the instruction scheduling is performed by the compiler instead by sophisticated and power-hungry hardware instruction schedulers used in their RISC counterparts. This is achieved by maximizing resource utilization by only targeting a certain application domain. However, when the inherent application ILP (instruction-level parallelism) is low, resources are under-utilized/wasted and the encoding of NOPs results in large code sizes and consequently additional pressure on the memory subsystem to store these NOPs.**

To address the resource-utilization issue, we proposed a dynamic VLIW processor design that can merge unused resources to form additional cores to execute more threads. Therefore, the formation of cores can result in issue widths of 2, 4, and 8. Without sacrificing the possibility of code interruptability and resumption, we proposed a generic binary scheme that allows a single binary to be executed on these different issue-width cores. However, the code size issue remains as the generic binary scheme even slightly further increases the number NOPS.

Therefore, in this paper, we propose to apply a well-known stop-bit code compression technique to the generic binaries that, most importantly, maintains its code compatibility characteristic allowing it to be executed on different cores. In addition, we present the hardware designs to support this technique in our dynamic core. For prototyping purposes, we implemented our design on a Xilinx Virtex-6 FPGA device and executed 14 embedded benchmarks. For comparison, we selected a non-dynamic/static VLIW core that incorporates a similar stop-bit technique for its code compression.

We demonstrate, while maintaining code compatibility on top of a flexible dynamic VLIW processor, that the code size can be significantly reduced (up to 80%) resulting in energy savings, and that the performance can be increased (up to a factor of three). Finally, our experimental results show that we can use smaller caches (2 to 4 times as small), which will further help in decreasing energy consumption.

I. Introduction

VLIW processors exploit ILP by means of a compiler, which statically analyzes the code and builds large instruction words (bundles) composed of instructions (syllables) that will execute in parallel. Since the compiler takes the burden of finding parallelism, VLIW processors occupy less area and dissipate less power when compared to superscalar processors. However, one of the major drawbacks of traditional VLIW processors is the large code size [1]. Because of this, instruction cache misses are more likely for VLIW processors when compared to conventional processors for a given cache size. Consequently, there will be more accesses to the main memory, which has higher delay and needs even more energy when compared to cache memories [2]. One solution would be increasing the cache size. However, this will also significantly increase the power dissipation, as several studies [3], [4] show that the energy consumption of the cache subsystem accounts for over 50% of the overall chip.

One reason for the large code size is the canonical instruction format that dictates the location of the syllables within each instruction word to correspond with the location of the functional units in the different datapaths. Consequently, unused datapaths must be issued NOPs that in turn must be encoded. One way to address this issue is to loosen the relationship between the instruction encoding and the datapath locations at the expense of having a more complex instruction decoder/scheduler. An intermediate solution maintains the canonical nature but encodes all NOPs at the end of an VLIW word with a single bit, the stop bit. These solutions reduce the code size and, consequently, reduce the pressure on the instruction cache (I-cache), reduce the number of I-cache misses, improve the performance of an application, and reduce energy consumption.

The code size issue is mainly due to the low inherent parallelism of the application, i.e., it is not possible to fill all parallel slots of a VLIW word. The nature of VLIW architectures automatically translates this issue into underutilized hardware resources. The ρ-VEX processor was introduced to deal with this low resource utilization by either power gating of datapaths to lower power consumption or merge datapaths together to form additional cores to execute parallel threads (if any). This capability allows the dynamic version of the ρ-VEX to switch at run-time between 2-, 4-, and 8-issue modes. One of the key innovations of the ρ-VEX processor

This work has been supported by the Almarvi European Artemis project nr. 621439.

is the introduction of the generic binary [5] that maintain code compatibility among different (dynamic) configurations of the processor allowing the code to be executable on any configuration (2-, 4-, or 8-issue) and interruptable at any point. The generic binary "suffers" from the same fate as their static counterparts when considering the code size.

In this paper, we propose a modified version of the variable length instruction bundle technique, which can be applied to the generic binaries of the dynamic VLIW processor (ρ-VEX). We also show how this compression technique leverages the current dispatch hardware and extra functional units to make its implementation more straightforward. More specifically, our approach is to apply the sparse instruction encoding (supported by the ISA of the processor) to the dynamic VLIW by using functional units required for supporting dynamic reconfiguration in order to reduce the complexity of dispatching sparse instruction bundles. In conclusion, we are proposing a new approach for code size reduction that marries the benefits of a well-known technique with the dynamic characteristics of the ρ-VEX processor. For comparison purposes, we use a static (non-reconfigurable) VLIW with an implementation of sparse instruction encoding similar to that found in the st200 [1]. The result is twofold: code sizes are reduced, and the difference in performance and energy between the static and dynamic versions decreases (slightly decreasing the price paid for adaptability).

Our contributions in this paper are:

- We implement support for the variable length instruction bundle, based on stop bits, in the dynamic ρ-VEX core to decrease code size, while maintaining compatibility for generic binaries, and therefore the run-time adaptability.
- By leveraging the additional functional units and dispatch logic already present, we show that the extra hardware complexity and overhead needed are insignificant.
- We compare the proposed approach to the non-dynamic version (with and without code compression) of the same processor, which is similar to a processor from the industry (STmicroelectronics' st200 VLIW). We demonstrate that the code compression is highly efficient in both versions, and that in most cases the dynamic version with compression has almost the same performance and energy consumption as the static one.

Comparing the dynamic version with and without the proposed technique, we achieve a code size reduction of over 50%, resulting in cache performance equivalent to that of a cache up to 4 times larger. Additionally, we can save up to 63% on energy consumption, while a maximum speedup of 3 times can be obtained in the best case.

II. Related Work

Conventional VLIW implementations had major drawbacks in the form of low instruction encoding efficiency (a large fraction of the code consisted of NOPs), which, together with the large number of operations that can be needed in a single cycle, resulted in enormous memory bandwidth requirements for instruction fetching. In order to address this issue, several approaches have been proposed:

- *Instruction mask bits*. The MAJC architecture [6] from Sun Microsystems exploits the parallelism at multiple levels: instruction, data, thread and process, through vertical and speculative multithreading, and chip multiprocessing. Mask bits indicate how many and what type of operations the instruction contains. In [7] and [8], the authors use a mask word that encodes which operations are present in the following bundle.
- *Instruction template bits*. These templates are used to limit code size, helping to decode and route the instructions, as used in the Itanium [9] and TM3270 media-processor [10]. The latter uses the templates to determine the compression of the next bundle, which relaxes the timing requirements of the decoding process.
- *Stop-bits*. A bit is reserved in each syllable, indicating whether it is the last syllable in a bundle or not, as presented in [11], [12], [13], [14], and [15].

However, none of the above-mentioned approaches are directly applicable to a dynamically reconfigurable VLIW processor. In order to apply variable length instruction bundle encoding to the ρ-VEX, we implement an extension of the stop-bit approach, which makes it suitable for variable issue widths, therefore maintaining the processor's dynamic adaptability.

III. Implementation

We will compare our implementation of sparse instruction encoding in a dynamic core to an existing encoding scheme applied to a static core. Both schemes are based on the stop-bit approach. The dynamic core is *run-time* reconfigurable in the number of datapaths. It can be configured as an 8-, 4-, or 2-issue core, whereas the static core has a fixed issue-width of 4 because the st200 toolchain which we use to compile applications for it only supports 4-issue.

A. Shared requirements

A number of properties are required by both cores. They will be outlined here and the precise implementations will be discussed in their respective sections. To support a sparse encoding, the hardware must support the following:

- Instruction bundles of variable length. That is, the location of the stop-bit determines which syllables should be executed, and also impacts the calculation of the next Program Counter (NextPC).
- Instruction bundles that cross a cache line boundary. An instruction buffer is used to store the relevant parts of the previous cache line to accommodate this in both designs.

B. Static Core

1) Overview: The VEX ISA is very closely related to STmicroelectronics' st200/Lx [11]. Both use an instruction encoding scheme with the following restrictions [16] that help to reduce the complexity of the fetch hardware [17]:

- Branch operations must be the first syllable in a bundle.

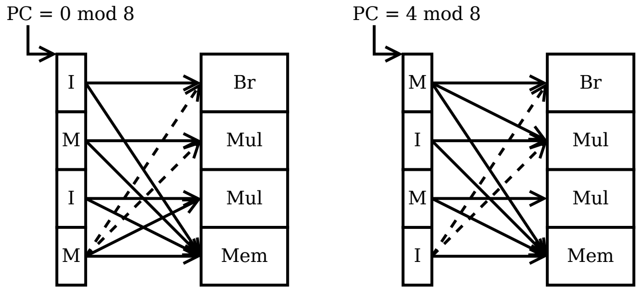


Fig. 1. Possibilities for the dynamic issue hardware using the st200 encoding scheme for bundles stored on even (left) and odd (right) word addresses.

- Multiplication operations must be stored at odd word addresses. This also restricts the number of multiplication operations to two per bundle.
- Long immediate extensions must be stored at even word addresses.

Considering these restrictions, the next section discusses the hardware modifications necessary for the implementation of the encoding technique.

2) Hardware: As opposed to the st200 implementation, our instruction cache does not support unaligned accesses and the core is designed to be generic (design-time configurable). To be generic, the design is very "lane-oriented", because the number of execution lanes or "datapaths" is generic, and every lane can be configured with multiple functional units (load/store, branch, ALU, and multiplication unit).

The first step in order to enable variable sized instruction bundles in the design is to add support for unaligned instruction cache accesses and bundles that cross a cache-line boundary. To this end, we add an instruction buffer between the cache and the fetch unit. This adds an additional stage to the datapath, increasing the branch delay by one cycle. The instruction buffer is similar to the design discussed in Section III-C2a.

The second step is to add dispatching logic that can send each syllable to a datapath that contains the functional unit that can execute the syllables operation type. The hardware required to fully support this can be quite complex. When every datapath needs to be able to accept an operation from any syllable slot in a bundle, a full crossbar is required [14]. Fortunately, the restrictions in the encoding scheme reduce this routing complexity to the diagram depicted in Fig. 1. The figure shows the locations of the MEM (load/store), MUL (multiplication) and BR (branch) units. Each datapath also contains an ALU, which is not depicted. The figure also depicts whether each syllable slot can contain a long immediate extension (I) or a multiplication (M) for even or odd instruction bundle start addresses. The arrows show which lanes it is possible to dispatch a particular syllable to. The dotted arrows depict an indirect requirement that is needed when two operations need to be swapped (e.g., if a MEM operation is located in syllable 0 and an ALU operation is located in slot 3, they will need to be swapped as the MEM unit is located in datapath 3).

TABLE I
LAYOUT OF FUNCTIONAL UNITS IN AN 4-ISSUE DYNAMIC ρ-VEX.

Datapath 0	Datapath 1	Datapath 2	Datapath 3
ALU MUL MEM	ALU MUL BR	ALU MUL MEM	ALU MUL BR

3) Discussion: As we can see in Fig. 1 the dispatch logic for a 4-issue core is already quite complex. Expanding the dispatch hardware discussed in this section to an 8-way VLIW would complicate the circuitry even more. The complexity would increase considerably, but not exponentially, because not all of the functional units (e.g. load/store, branch) are duplicated when doubling the issue width. As we will see in the following sections, our approach in the dynamic ρ-VEX core is able to dispatch operations with simpler logic that can be more efficiently scaled to an 8-way VLIW.

C. Dynamic Core

1) Overview: The dynamic core consists of multiple datapaths, which are divided into groups of two. Each datapath has a fixed set of functional units. Each group of two datapaths can function as a separate VLIW core, or can be combined with adjacent groups to form a larger VLIW core. In order to allow this, each group of two datapaths must have a functional unit layout that is identical to that of the the other groups. Additionally, to allow each datapah group to function as a separate VLIW core, each group must have a load/store unit and a branch unit. However, when the groups are configured to combine into a 4-issue VLIW, these additional load/store and branch units go unused. For example, the layout of functional units in each datapath is shown in Table I for a 4-issue configuration. By using this arrangement of functional units we are able to support sparse instruction encoding without decoding logic to dispatch instructions to different functional units.

2) Hardware: In addition to the requirements mentioned in Section III-A, in order to support sparse instruction encoding in the dynamic core, we also require the next PC calculation to be able to support dynamic switching of issue-width. This means that it should be possible to calculate either a single address, or multiple addresses based on the core configuration. Additionally, unlike in the static core, branch instructions must be able to appear in any lane to maintain the single/multiple core adaptability.

a) Instruction Buffer: When using sparse instruction encoding, bundles no longer have a fixed size. Because of this, the cache line size is no longer divisible by the bundle size, which means that instruction bundles can cross cache line boundaries. We handle this by fetching the next cache line and storing the previous one in a buffer in order to complete the instruction bundle.

Fig. 2 shows a diagram of the instruction buffer. Each lane-group has a register to store the previously fetched syllables

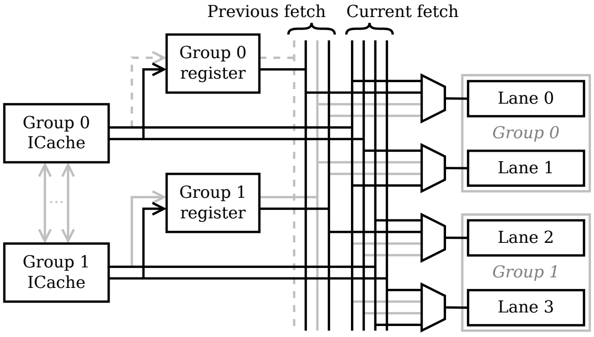


Fig. 2. This figure shows how the instruction buffer is implemented for a 4-issue dynamic core consisting of two lane groups.

(two syllables per group). The muxes select which syllable goes to which datapath based on the current configuration of the core (2-, 4-, or 8-issue), and the least significant bits of the program counter. The gray lines indicate paths that are only used when the core is configured in 4-issue mode. The dotted line indicates a path which would be used if the core were in 8-issue mode.

The instruction buffer registers are loaded whenever a new fetch address is sent to the cache. This can be determined by comparing the least significant bit of the current and previous fetch address, reducing the size of the required comparator significantly. Note that when a branch to an unaligned address occurs, the syllables in the instruction buffer are undefined. Additional logic is present in the branch unit to stall the core for an extra cycle in this case, during which an additional instruction fetch is performed in order to fill the instruction. This additional stall could adversely affect performance if it occurs often enough.

The muxes that select between the previously fetched data and the current fetched data are controlled by signals based on the current configuration (2-issue, 4-issue, or 8-issue) and the LSBs (Least Significant Bits) of the Program Counter (PC). For the case where the core is configured as the largest possible configuration (all lane-groups work together as one core) the mux select signals are equal to the least significant bits of the Program Counter.

b) Address Calculation: When using variable sized instruction bundles, the next value of the program counter depends on the size of the current fetched instruction bundle. This complicates the calculation of the next PC, which can now be PC + 4, 8, 12 or 16 (for a 4-issue VLIW). We deal with this by splitting the calculations into two parts:

- Calculation of the least significant bits (depicted in Fig. 3) is done for each lane to determine what the least significant bits would be if the instruction bundle ended in a particular lane. In each lane, a value corresponding to a different instruction bundle size is added to the least significant bits of the PC. For example, if the stop-bit is in the first instruction, that means the instruction bundle size is 4 bytes, and 4 is added to the PC. Additionally, the "align up" adder rounds up the calculated least significant bits to a cache line boundary. This will be used for the next fetch address.
- The most significant bits of the next PC are calculated in each branch unit (for the 4-issue configuration used in this paper that would be lanes 1 and 3). Finally, the least and most significant bits are combined to form the next PC based on the final position of the stop bit.

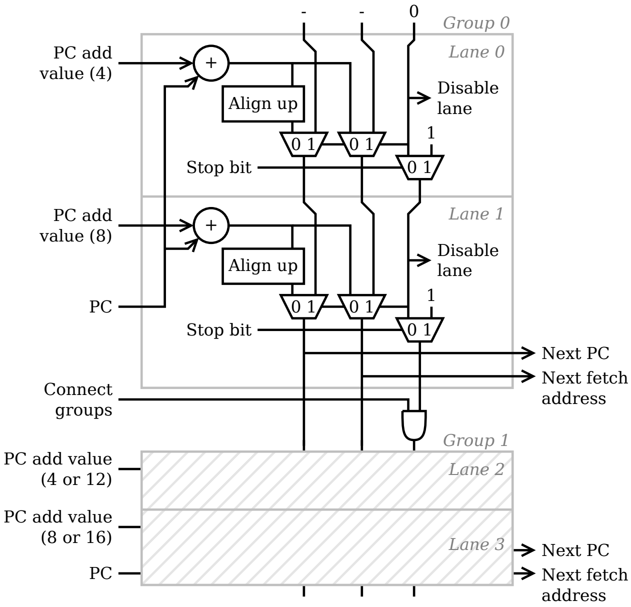


Fig. 3. This figure shows how the LSB of the next PC and next fetch address are calculated in each lane.

By splitting the program counter calculation in this way, we only need four 27-bit adders to calculate the most significant bits, of which only two are used in 4-way mode, and eight 3-bit adders for the least significant bits, instead of needing eight 32-bit adders.

c) Branch Instruction Dispatch: As mentioned in [5], generic binaries require that the branch instruction is always the last in a bundle, rather than the first. This requirement is in place to ensure that the bundle will still be executed completely by a core running in a 2-way configuration (otherwise, the syllables following the branch would be skipped because of the branch). In order to support this, additional logic is present to route the last instruction in a bundle to the last coupled lane if it is a branch instruction, so only one physical branch resource is used for executing branch instructions.

Because instruction bundles can now cross cache line boundaries, it is possible that after a branch only part of the next instruction bundle is fetched. The hardware detects this by checking if one of the fetched syllables (starting from the branch target address) has a stop-bit set. If not, the core stalls while the second part of the bundle is fetched. This means that for unaligned branches the branch delay is two cycles instead of one, which can cause performance degradation.

TABLE II

THE RESOURCE USAGE ON THE FPGA FOR THE DYNAMIC CORE WITH AND WITHOUT STOP-BIT IMPLEMENTATION.

Resource	Original	Stop-bit	Increase
Registers	30153	30537	1.3%
Luts	61927	62379	0.7%
BRAMs	125	125	0.0%

IV. Results

We evaluated four different versions of the processors: static baseline, static with stop-bit, dynamic baseline, and dynamic with stop-bit — all of them in their 4-issue configurations. We use the 4-issue configurations to provide a fair comparison between the static and dynamic cores. The difference between dynamic and static versions is that the binaries for the former are compiled as generic binaries. We considered instruction cache sizes ranging from 1KiB to 32KiB. These sizes were chosen so that at the largest cache size each of the programs fits in the instruction cache entirely.

The designs are implemented in VHDL and prototyped on a Xilinx Virtex 6 FPGA (ML605 Development board). With these prototypes, we use performance counters to determine the number of cache accesses, misses, and the number of running cycles. The cache stall time is 16 cycles per 4-byte bus access. We use the Cadence Encounter RTL Compiler to obtain power dissipation in ASIC (Application Specific Integrated Circuit), using a 65nm CMOS cell library from STMicroeletronics. The energy consumption of the memory subsystem was calculated with the Cacti Tool [18].

We use applications from the Powerstone benchmarks [19]. All sources are compiled with the HP VEX compiler [20] and assembled with either the ρ-VEX port of GNU as, or our modified version of the st200 assembler. The dynamic stop-bit versions are assembled with alignment turned off, so that instruction bundles are not padded at all. Since the processor lacks floating point operations, we use the floatlib library included with the HP VEX compiler (based on Berkeley SoftFloat [21]).

A. FPGA Resource usage

Table II shows the resource usage of the dynamic core on the FPGA. It shows that the increase is only 1.3% for the number of registers and 0.7% for the number of lookup tables. As we will show in the following sections, with this small increase in area we achieve significant improvements in performance, energy, and code size.

B. Code Size Reduction and Instruction Cache Miss Rate

In Table III, we show the reduction in code size for each of the 14 benchmarks used. We can see that the average reduction is around 50%. The reductions for the dynamic core in 8-way configuration are included for reference, and are even more extreme. These reductions will impact the cache behavior. In Fig. 4, we show the cache miss rates for the two different cores with and without sparse instruction encoding. The results show that both designs achieve a similar reduction in cache misses. In fact, with sparse instruction encoding the miss rates are similar to those of canonical encoding with a cache almost four times as large. This might seem like a larger improvement than expected, since the code size was only reduced by half. However, because loops account for a majority of the executed instructions, code size reduction that allow an entire loop body to fit into the cache will have a disproportionate impact on the cache miss rate.

TABLE III

THE CODE SIZE REDUCTION FOR EACH OF THE BENCHMARKS.

Program	code size reduction		
	static 4-way core	dynamic 4-way core	dynamic 8-way core
adpcm	49%	48%	73%
bcnt	35%	38%	64%
blit	47%	45%	67%
compress	53%	51%	74%
crc	48%	48%	71%
des	42%	44%	68%
engine	57%	54%	77%
fir	60%	54%	76%
g3fax	58%	55%	76%
jpeg	53%	51%	73%
pocsag	55%	51%	74%
qurt	67%	65%	82%
ucbqsort	57%	54%	76%
v42	56%	53%	75%
average	53%	51%	73%

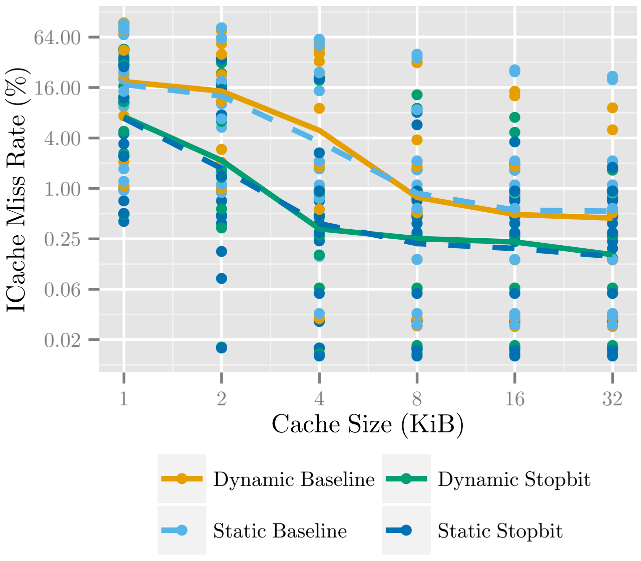


Fig. 4. Cache miss percentage for the dynamic and static cores, both with and without sparse instruction encoding for different instruction cache sizes. The dots represent the individual benchmarks, whereas the lines represent the average miss percentage for a particular configuration.

C. Execution Time

Fig. 5 shows the speedup in execution time achieved for both the dynamic and static cores. We can see that for larger cache sizes, the execution time of some benchmarks is larger

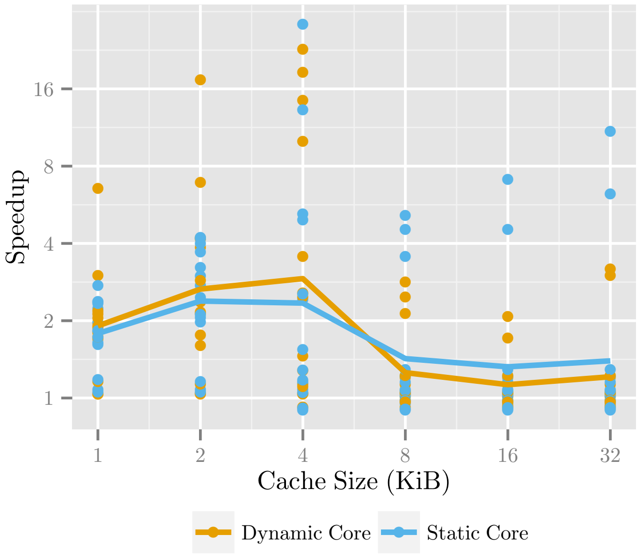


Fig. 5. Speedup for stop-bit implementation for different instruction cache sizes. The lines represent the average speedup for a particular cache size.

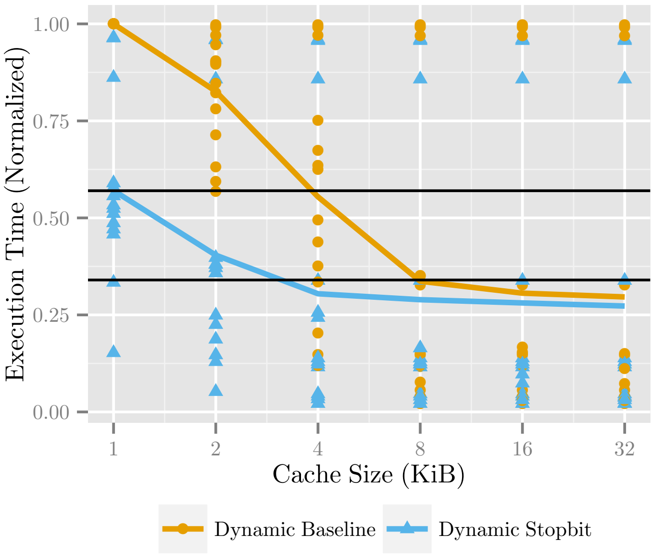


Fig. 6. Normalized execution times for the dynamic core

with sparse instruction encoding than without. This is because at those cache sizes the entire application fits in the cache, and the reduction in cache misses is offset by the penalty of having a longer branch delay. This could be remedied by inserting alignment NOPs to ensure that branch targets are always aligned, at the cost of an increase in cache misses.

In the same figure we also observe that for very small instruction cache sizes, the speedup is not as significant as it is for intermediate sizes. This is caused by the fact that the reduction in cache misses for intermediate cache sizes is far larger than for small cache sizes, as seen in Fig. 4.

Fig. 6 shows the normalized execution times for the dynamic core. The baseline is the average of the worst execution time of each application individually, executing on the dynamic baseline design with a cache size of 1KiB. This figure shows, for instance, that for smaller cache sizes, the dynamic stop-bit implementation performs equivalent to the dynamic version without stop-bit with a cache between 2 and 4 times larger.

D. Energy Results

Fig. 7 presents the total energy consumed by each of the benchmarks. The lines show the geometric mean of all applications at each cache size. We can see that for small cache sizes, due to the additional hardware required to support reconfiguration, the dynamic core consumes more energy than the static core. However, at large cache sizes the different designs are closer together in terms of energy consumption.

Fig. 8 depicts the energy consumption of the dynamic core relative to that of the static core (values greater than 1 mean that the static version consumes less energy than the dynamic one). We can see that the baseline dynamic design consumes far more energy at small cache sizes, whereas when using sparse instruction encoding the designs consume similar amounts of energy.

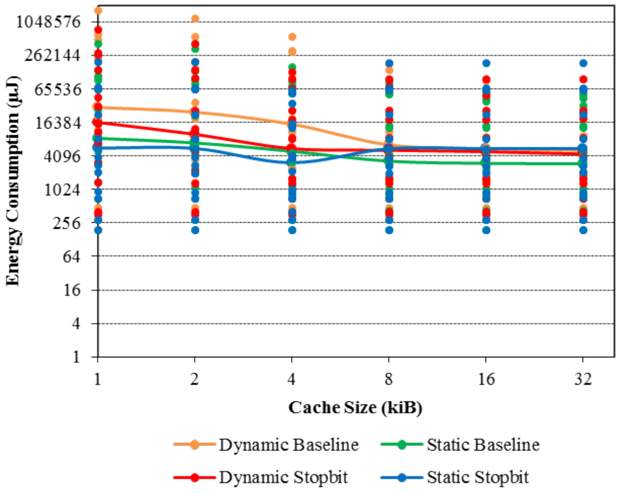


Fig. 7. Energy consumption for each of the benchmarks at different cache sizes.

As one can observe, the huge difference in energy consumption between the static and dynamic versions is significantly decreased when using the proposed stop-bit approach. Most notably, both processors consume approximately the same amount of energy at larger cache sizes. It means that one can take advantage of all the adaptability that the dynamic version provides, with limited additional costs in terms of energy.

V. Conclusion

In this paper, we extended the stop-bit technique for sparse instruction encoding to a dynamically reconfigurable VLIW processor. We showed that, by implementing this technique,

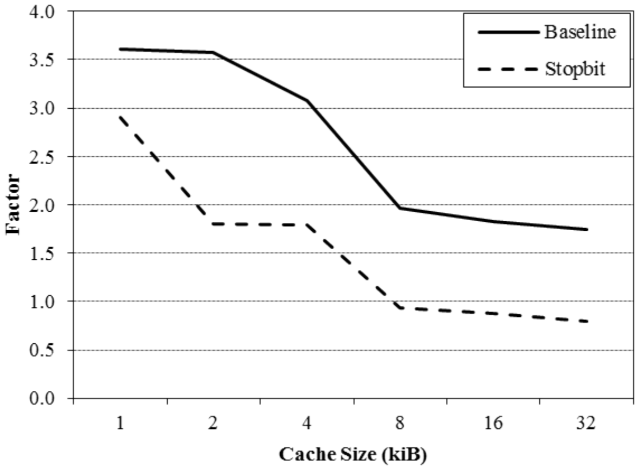


Fig. 8. Relative energy consumption between the static and dynamic cores (dynamic/static) at different cache sizes.

we reduce the cost of reconfigurability in terms of energy consumption and the performance overhead of cache misses. This is achieved without sacrificing the code compatibility of the generic binary and we thereby maintain full (dynamic) adaptability of the core. Using this technique, we bring the energy consumption of the dynamic core closer to that of the static design. Our results show that using the stop-bit technique in the dynamic core we can achieve similar performance and energy consumption with up to 4× smaller I-caches.

We do notice that for some applications at certain cache sizes the performance with stop-bit is slightly lower than without stop-bit due to the increased branch delay. Therefore, for future work we will investigate the effect of ensuring that branch target addresses are always correctly aligned. This would result in a slight increase in cache misses but also a decrease in delays due to branches.

References

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

[2] V. A. Korthikanti and G. Agha, "Towards optimizing energy costs of algorithms for shared memory architectures," in *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '10. New York, NY, USA: ACM, 2010, pp. 157–165. [Online]. Available: http://doi.acm.org/10.1145/1810479.1810510

[3] C. Zhang, F. Vahid, and W. Najjar, "A highly configurable cache for low energy embedded systems," *ACM Trans. Embed. Comput. Syst.*, vol. 4, no. 2, pp. 363–387, May 2005. [Online]. Available: http://doi.acm.org/10.1145/1067915.1067921

[4] W. Wang, P. Mishra, and S. Ranka, "Dynamic cache reconfiguration and partitioning for energy optimization in real-time multi-core systems," in *Proceedings of the 48th Design Automation Conference*, ser. DAC '11. New York, NY, USA: ACM, 2011, pp. 948–953. [Online]. Available: http://doi.acm.org/10.1145/2024724.2024935

[5] A. Brandon and S. Wong, "Support for dynamic issue width in vliw processors using generic binaries," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, March 2013, pp. 827–832.

[6] M. Tremblay, J. Chan, S. Chaudhry, A. W. Conigliaro, and S. S. Tse, "The majc architecture: A synthesis of parallelism and scalability," *IEEE Micro*, vol. 20, no. 6, pp. 12–25, 2000.

[7] Instruction storage method with a compressed format using a mask word, www.google.com/patents/US5057837.

[8] S. Jee and K. Palaniappan, "Performance evaluation for a compressed-vliw processor," in *Proceedings of the 2002 ACM symposium on Applied computing*. ACM, 2002, pp. 913–917.

[9] H. Sharangpani and K. Arora, "Itanium Processor Microarchitecture," *IEEE Micro*, pp. 24–43, Sep. 2000.

[10] J.-W. van de Waerdt, S. Vassiliadis, S. Das, S. Mirolo, C. Yen, B. Zhong, C. Basto, J.-P. van Itegem, D. Amirtharaj, K. Kalra *et al.*, "The TM3270 media-processor," in *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2005, pp. 331–342.

[11] J. A. Fisher, P. Faraboschi, and C. Young, *Embedded Computing: A VLIW Approach to Architecture, Compilers, and Tools*. 500 Sansome Street, Suite 400, San Francisco, CA 94111: Morgan Kaufmann Publishers, 2005.

[12] Method and apparatus for sequencing and decoding variable length instructions with an instruction boundary marker within each instruction, http://www.google.com/patents/US5881260.

[13] A. Suga and K. Matsunami, "Introducing the fr500 embedded microprocessor," *Micro, IEEE*, vol. 20, no. 4, pp. 21–27, 2000.

[14] T. M. Conte, S. Banerjia, S. Y. Larin, K. N. Menezes, and S. W. Sathaye, "Instruction fetch mechanisms for VLIW architectures with compressed encodings," in *Microarchitecture, 1996. MICRO-29. Proceedings of the 29th Annual IEEE/ACM International Symposium on*. IEEE, 1996, pp. 201–211.

[15] B. Hubener, G. Sievers, T. Jungeblut, M. Porrmann, and U. Ruckert, "Coreva: A configurable resource-efficient vliw processor architecture," in *Embedded and Ubiquitous Computing (EUC), 2014 12th IEEE International Conference on*. IEEE, 2014, pp. 9–16.

[16] ST231 Core and Instruction Set Architecture Manual.

[17] Instruction fetch apparatus for wide issue processors and method of operation, http://www.google.com/patents/US7028164.

[18] S. Thoziyoor, J. H. Ahn, A. Monchiero, J. B. Brockman, and N. P. Jouppi, "A comprehensive memory modeling tool and its application to the design and analysis of future memory hierarchies," in *Proc. 35th International Symposium on Computer Architecture (35th ISCA'08)*. Beijing: ACM SIGARCH, Jun. 2008.

[19] Powerstone Benchmarks, http://www.cprover.org/goto-cc/examples/index.php.

[20] The HP VEX toolchain, http://www.hpl.hp.com/downloads/vex/.

[21] http://www.jhauser.us/arithmetic/SoftFloat.html.

C

ρ-VEX user manual

The remainder of this thesis contains a copy of the ρ-VEX user manual at the time of the author's thesis defense. If you have received this thesis in print, it is not included, in consideration of the environment.

ρ-VEX user manual

Jeroen van Straten, TU Delft

April 18, 2016

Git revision: `5cd37f2`
Core version tag: `z1KS3dj`

Contents

1 Introduction **C-7**

2 Overview of the ρ-VEX processor **C-9**

2.1 Reconfiguration C-9

2.2 Generic binaries C-10

2.3 Intended applications C-10

3 Instruction set architecture **C-13**

3.1 Assembly syntax C-13

3.2 Registers C-14

3.2.1 General purpose registers C-14

3.2.2 Branch registers C-14

3.2.3 Link register C-15

3.2.4 Global and context control registers C-15

3.3 Memory C-15

3.4 Syllable resource classes and delays C-16

3.4.1 ALU class C-16

3.4.2 Multiply class C-16

3.4.3 Memory class C-17

3.4.4 Branch class C-17

3.4.5 Long immediate class C-17

3.5 Generic binaries C-18

3.5.1 Generating generic binaries C-19

3.6 Stop bits C-19

3.6.1 Design-time configuration C-20

3.7 Instruction set C-21

3.7.1 ALU arithmetic instructions C-24

3.7.2 ALU barrel shifter instructions C-27

3.7.3 ALU bitwise instructions C-27

3.7.4 ALU single-bit instructions C-29

3.7.5 ALU boolean instructions C-30

3.7.6 ALU compare instructions C-32

3.7.7 ALU selection instructions C-36

3.7.8 ALU type conversion instructions C-38

3.7.9 ALU miscellaneous instructions C-39

3.7.10 Multiply instructions C-40

3.7.11 Memory instructions C-43

3.7.12 Branch instructions C-46

3.7.13 Long immediate instructions C-48

4 Control registers **C-49**
4.1 Global control registers C-49
4.1.1 CR_GSR - Global status register C-50
4.1.2 CR_BCRR - Bus reconfiguration request register C-50
4.1.3 CR_CC - Current configuration register C-51
4.1.4 CR_AFF - Cache affinity register C-51
4.1.5 CR_CNT - Cycle counter register C-51
4.1.6 CR_CNTH - Cycle counter register high C-51
4.1.7 CR_LIMCn - Long immediate capability register n C-52
4.1.8 CR_SICn - Syllable index capability register n C-52
4.1.9 CR_GPS1 - General purpose register delay register B C-53
4.1.10 CR_GPS0 - General purpose register delay register A C-53
4.1.11 CR_SPS1 - Special delay register B C-54
4.1.12 CR_SPS0 - Special delay register A C-54
4.1.13 CR_EXT2 - Extension register 2 . . . . C-55
4.1.14 CR_EXT1 - Extension register 1 . . . . C-56
4.1.15 CR_EXT0 - Extension register 0 . . . . C-56
4.1.16 CR_DCFG - Design-time configuration register C-57
4.1.17 CR_CVER1 - Core version register 1 . . . . C-57
4.1.18 CR_CVER0 - Core version register 0 . . . . C-58
4.1.19 CR_PVER1 - Platform version register 1 . . . . C-58
4.1.20 CR_PVER0 - Platform version register 0 . . . . C-58
4.2 Context control registers C-59
4.2.1 CR_CCR - Main context control register C-60
4.2.2 CR_SCCR - Saved context control register C-62
4.2.3 CR_LR - Link register C-63
4.2.4 CR_PC - Program counter C-63
4.2.5 CR_TH - Trap handler C-63
4.2.6 CR_PH - Panic handler C-64
4.2.7 CR_TP - Trap point C-64
4.2.8 CR_TA - Trap argument C-64
4.2.9 CR_BRn - Breakpoint n C-65
4.2.10 CR_DCR - Debug control register 1 . . . . C-65
4.2.11 CR_DCR2 - Debug control register 2 . . . . C-67
4.2.12 CR_CRR - Context reconfiguration request register C-68
4.2.13 CR_WCFG - Wakeup configuration C-69
4.2.14 CR_SAWC - Sleep and wake-up control register C-69
4.2.15 CR_SCRPn - Scratchpad register n C-70
4.2.16 CR_RSC - Requested software context C-70
4.2.17 CR_CSC - Current software context C-70
4.2.18 CR_RSCn - Requested swctxt on hwctxt n C-71
4.2.19 CR_CSCn - Current swctxt on hwctxt n C-71
4.2.20 CR_CYC - Cycle counter C-71
4.2.21 CR_STALL - Stall cycle counter C-72
4.2.22 CR_BUN - Committed bundle counter C-72

4.2.23 CR_SYL - Committed syllable counter C-72
4.2.24 CR_NOP - Committed NOP counter C-72
4.2.25 CR_IACC - Instruction cache access counter C-73
4.2.26 CR_IMISS - Instruction cache miss counter C-73
4.2.27 CR_DRACC - Data cache read access counter C-73
4.2.28 CR_DRMISS - Data cache read miss counter C-73
4.2.29 CR_DWACC - Data cache write access counter C-74
4.2.30 CR_DWMISS - Data cache write miss counter C-74
4.2.31 CR_DBYPASS - Data cache bypass counter C-74
4.2.32 CR_DWBUF - Data cache write buffer counter C-74
4.3 Performance counter registers C-74

5 Traps and interrupts C-77
5.1 Trap sources C-77
5.2 Trap and panic handlers C-78
5.3 Trap identification C-79
5.4 State saving and restoration C-82

6 Reconfiguration and sleeping C-83
6.1 Configuration word encoding C-83
6.2 Requesting a reconfiguration C-84
6.3 Sleep and wake-up system C-85
6.3.1 Power saving C-86
6.3.2 Decreasing interrupt latency C-86

7 Debugging ρ-VEX software C-89
7.1 Setting up C-89
7.2 Connecting to a remote machine C-90
7.3 Connecting to the FPGA C-90
7.4 Running programs C-90
7.5 Debugging programs C-91
7.6 Tracing execution C-91

8 Design-time configuration C-93
8.1 Control register files C-94
8.1.1 .tex command reference C-95
8.1.2 Language-agnostic code (LAC) sections C-98
8.1.3 LAC type system C-98
8.1.4 LAC predefined constants C-101
8.1.5 LAC literals C-101
8.1.6 LAC object objects and references C-102
8.1.7 LAC operators C-103
8.1.8 LAC statements C-103
8.2 Instruction set C-104
8.2.1 .tex file structure C-104
8.2.2 .tex command reference C-105

8.2.3 encoding.ini reference C-106
8.3 Pipeline C-107
8.4 Traps C-107
8.4.1 .tex file structure C-107
8.4.2 .tex command reference C-107

9 Instantiation C-109
9.1 Data types C-109
9.2 Bare ρ-VEX processor C-109
9.2.1 Instantiation template C-110
9.2.2 Interface description C-111
9.3 Standalone processing system C-122
9.3.1 Instantiation template C-122
9.3.2 Interface description C-123
9.4 GRLIB processing system C-127
9.4.1 Instantiation template C-127
9.4.2 Interface description C-129

Bibliography C-133

Introduction 1

This manual intends to document the ρ-VEX reconfigurable VLIW processor. It is intended for software developers using the processor and hardware developers who are only interested in instantiating the processor in their system. It does not aim to document the internal workings of the processor; the comments in the source code are a better source of documentation for this. It also does not document the design choices that were made in the construction of the core; for this, readers are referred to [1].

The next chapter gives a top-level overview of the processor. The third chapter documents the instruction set architecture (ISA) in detail. The fourth chapter lists all the control registers of the processor. The fifth chapter handles the trap and interrupt system of the processor. The sixth handles reconfiguration, the component of the ρ-VEX that makes it special. The seventh documents how the core may be debugged using a computer. The final two chapters are intended for the hardware developers only, documenting the design-time configuration options of the core and how it may be instantiated.

Overview of the ρ-VEX processor 2

Let us begin by defining some terminology. The ρ-VEX processor is a Very Large Instruction Word (VLIW) processor, which means that each instruction can specify multiple independent operations. Such operations are called *syllables*; a full instruction is called a *bundle*. *Instruction* may be used for either a bundle or a syllable, depending on context. A VLIW processor capable of executing n syllables per cycle is called an *n-way* VLIW processor.

Because the amount of syllables in a bundle is usually[1] not fixed, the processor needs a way to tell which syllables belong to which bundle. In the VEX architecture, this is done by means of a *stop bit* in each syllable. If the stop bit is set, the next syllable in the program starts a new bundle. Otherwise, the next syllable is part of the same bundle.

When a VLIW processor executes a bundle, each syllable will be routed to its own *(pipe)lane*. Note the 'a' in lane; this is not a typo for pipeline (although each pipelane, confusingly, does contain its own pipeline). In other words, the pipelane is the thing that contains the computational resources to execute a syllable.

2.1 Reconfiguration

What makes the ρ-VEX processor special compared to other VLIW processors, is that while the total number of pipelanes is obviously fixed, the pipelanes can be distributed between different programs, running in parallel. This distribution can be changed at runtime by means of *reconfiguration*.

Note that 'reconfiguration' here is used to describe a process within the system described by a single FPGA bitstream. In other words, the FPGA bitstream does not need to be fully or partially reloaded when the ρ-VEX processor reconfigures itself. This allows reconfiguration to be done in a single cycle in theory, although it comes at the cost of needing FPGA slice muxes or LUTs to permit reconfiguration, instead of using the FPGA fabric directly.

Not all pipelanes are seperable by means of reconfiguration. Groups of inseperable pipelanes are called *lane groups*. Sometimes they are also referred to as *lanepairs* when a lane group contains two pipelanes, which is the most common configuration.

In order to be able to run multiple programs on a single ρ-VEX processor core at the same time, an ρ-VEX processor supports multiple *contexts*. Formally, a context contains the complete state of a program, from program counter to register file. However, a

[1] It is uncommon for the compiler to find enough parallelism in a program to fill an entire bundle. Therefore, if the bundle size is fixed, a lot of syllables will be NOP. While a fixed bundle size results in much simpler hardware, the size of the binary will be excessive. While main memory footprint is not so much an issue nowadays, memory throughput and latency is; the efficiency of the instruction coding directly affects execution speed as the memory is usually the bottleneck.

more useful way to think of ρ-VEX contexts is as virtual processor cores. By means of reconfiguration, the amount of lane groups dedicated to each virtual core can be changed. In fact, it is possible to completely pause such a virtual core by simply assigning zero lane groups to it.

2.2 Generic binaries

To compile for a VLIW processor, the compiler needs to be aware of what the maximum number of syllables per bundle is. However, reconfiguration changes this value at runtime, which would imply that each program should be compiled multiple times, for each bundle size possible with reconfiguration. This would severely limit the usefulness of reconfiguration, as it would be extremely difficult to reconfigure in the middle of program execution. At best, the program counters would be the only things that would not match between the two binaries.

The solution to this problem is a *generic binary* [2]. Generic binaries are compiled for the largest possible bundle size at which they may execute, referred to as the *generic bundle size*. This allows the compiler to extract as much parallelism as may ever be used. The difference between a normal binary compiled for the generic bundle size and a generic binary lies in additional rules imposed to the program by the assembler. These rules are carefully picked to ensure that, for instance, a bundle with four syllables in it still runs correctly if the two syllable pairs are run sequentially. Unless otherwise specified, an ρ-VEX generic binary refers to a binary compiled such that it runs correctly on 8-way, 4-way and 2-way ρ-VEX processor cores.

2.3 Intended applications

On the short term, the current version of the ρ-VEX processor is still primarily intended for research. The VHDL is written in a highly flexible and configurable way, thus making modifications for experiments relatively easy. At the same time, several complex features have been added to the core, in order to make it possible to, for instance, run Linux on it. Most notably, precise trap support has been added since the previous ρ-VEX version, necessary for adding a memory-management unit.

This combination of flexibility and complexity comes at a cost: speed. The current version of the ρ-VEX processor only runs at 37.5 MHz on a high-end Virtex 6 FPGA using the default configuration, while almost completely filling it up. Much more interesting is what the ρ-VEX architecture is capable of on the long run when better optimized, or even ported to an ASIC.

In general, VLIW processors are well-suited for executing highly parallel programs, such as those found in digital signal processing (DSP). In particular, the reconfiguration capabilities of the ρ-VEX processor allow it to be used in places where multiple DSP algorithms run in parallel in a real-time system, such that each task has its own deadlines.

To demonstrate, consider a hypothetical audio/video decoder DSP with the following characteristics as an example.

- The audio and video decoders do not depend on each other and can thus be executed in parallel. The decoders themselves are not multithreaded.
- Both tasks run 1.5x as fast when running on a 4-way VLIW compared to a 2-way VLIW.
- The execution times of both tasks are data dependent. For example, if there is a lot of movement in the video, then the video task will take longer to complete.
- It is possible to heuristically predict whether or not either decoder will meet its deadline at its current execution speed before the deadline, in a way that does not cost an excessive amount of additional computation. This can be done, for example, by decoding audio and video a few frames in advance, and assuming that if the current frame is computationally intensive, the next one will probably be too (locality).
- The audio task takes priority over the video task, as choppy audio is perceived as more intrusive than choppy video.
- For simplicity, assume that while the video decoder is decoding a single frame, the audio decoder has to decode a frame's worth of audio samples. In other words, the audio and video decoding tasks start at the same time and have the same deadline. In addition, assume that both tasks need an approximately equal amount of processing time for a single frame.

Let us now analyze the performance of this system if it were implemented on two 2-way VLIW processors. Each processor is simply assigned to one of the tasks. The primary downside to this system in the context of this discussion is that if the audio is overly complex, the audio decoder will miss its deadline, regardless of the whether the video processor was fully utilized or not.

To prevent this from happening, one may instead choose to implement the system on a single 4-way VLIW with a real-time operating system (RTOS) kernel. Notice that this system has the same amount of compute resources as the previous system. Now, the RTOS will ensure that the audio decoder runs before the video decoder. Because the audio decoder runs 1.5x as fast, it will likely meet its deadline now. While unlikely, it is possible that the video decoder will also complete in time now, but even if it does not, choppy video was considered favorable over choppy audio. The major downside of this system is that it is effectively much slower than the 2x2-way system, as the decoders do not actually run twice as fast when given twice as many computational resources, as the instruction level parallelism just is not always there.

The power of the ρ-VEX processor is that it can basically switch between these two implementations at runtime, depending on the actual load each task experiences. When neither task is in danger of failing to meet its deadline, the ρ-VEX processor could run in 2x2-way mode. However, if one of the tasks starts falling behind the other because it is more computationally intensive, the ρ-VEX processor could reconfigure to 1x4-way mode for that task. When it catches up, it will switch back to 2x2-way mode, as that is more efficient.

3 Instruction set architecture

The instruction set architecture (ISA) of the ρ-VEX processor is primarily based on VEX, an example VLIW architecture used in [3] to explain VLIW concepts. A compiler was developed for this architecture by HP, which is available as a free download for noncommercial use [4]. In addition, some enhancements were made to the instruction set to be compatible with the Lx architecture, introduced in [5]. Lx was designed by HP and STMicroelectronics for use in SoC (system-on-chip) designs, among which is the ST200 family of processors. This architecture comes with an Open64-based compiler and Linux port, available for download under the GNU GPL [6]. These tools together provide the basis for the toolchain used for the ρ-VEX. The instruction encoding for this version of the ρ-VEX processor is based on that of the previous major version of the ρ-VEX [7].

Instead of noting the differences between these architectures, this section functions as a reference for the ρ-VEX processor ISA in its current state, to save the reader from cross-referencing.

3.1 Assembly syntax

The following listing shows the syntax for a single instruction bundle.

```
start:
  c0 stw  0x10[$r0.1] = $r0.53
  c0 add  $r0.3       = $r0.0, -32
  c0 and  $b0.2       = $r0.0, $r0.10
  c0 call $l0.0       = interrupt
;;
```

The first line represents a label, as it ends in a colon. Each non-empty line that does not start with a semicolon and is not a label represents a syllable. The first part of the syllable, `c0`, is optional. It specifies the cluster that the syllable belongs to. Since the ρ-VEX processor currently does not support clusters, only cluster zero is allowed if specified. The second part represents the opcode of the syllable, defining the operation to be performed. The third part is the parameter list. Anything that is written to is placed before the equals sign, anything that is read is placed after. Finally, a double semicolon is used to mark bundle boundaries.

The syntax for a general purpose register is `$r0.`*index*, where *index* is a number from 0 to 63. The first 0 is used to specify the cluster, which, again, is not used in the ρ-VEX processor. Branch registers and the link register have the same syntax, substituting the 'r' with a 'b' or an 'l' respectively. The *index* for branch registers ranges from 0 to 7. For link registers only 0 is allowed.

Most instructions also accept a literal as their second operand. Literals may be a decimal or hexadecimal number (using `0x` notation), a label reference, or a basic C-like integer expression. Literals represent 32-bit values with undefined signedness, i.e., `0xFFFFFFFF` and -1 specify the same value.

Finally, the load and store instructions require a memory reference as one of their operands. Memory references use the following syntax: *literal*`[$r0.`*index*`]`. At runtime, the literal is added to the register value to get the address, i.e. base + offset addressing is used.

A port of the GNU assembler (`gas`) is used for assembly. Please refer to its manual for information on target-independent directives or more information on the expressions mentioned above.

In general, the C preprocessor is used to preprocess assembly files. This allows usage of the usual C-style comments, includes, definitions, etc. In particular, the control registers may be easily referenced as long as the appropriate files are included.

3.2 Registers

The ρ-VEX processor has five distinguishable register files. Each is described below.

3.2.1 General purpose registers

The ρ-VEX core contains 64 32-bit general purpose registers for arithmetic.

Register 0 is special, as it always reads as 0 when used by the processor. Writing to it does however work; the debug bus can read the latest value written to it. This allows the register to be used for debugging on rare occasions.

Register 1 is intended to be used as the stack pointer. The `RETURN` and `RFI` instructions can add an immediate value to it for stack adjustment, but otherwise it behaves just as any other general purpose register.

Register 63 can optionally be mapped to the link register at design time using generics. This allows arithmetic instructions to be performed on the link register without needing to use `MOVFL` and `MOVTL`, at the cost of a general purpose register.

There are no explicit move or load-immediate operations, as the following syllables are already capable of these operations.

```
c0 or $r0.dest = $r0.0, $r0.src        // Move src to dest
c0 or $r0.dest = $r0.0, immediate      // Load immediate
```

3.2.2 Branch registers

The ρ-VEX core contains 8 1-bit registers used for branch conditions, select instructions, divisions, and additions of values wider than 32 bits.

All arithmetic operations that output a boolean value can write to either a general purpose register (in which case they will write 0 for false and 1 for true) or a branch register. These include all integer comparison operations and select boolean operations.

Moving a branch register to another branch register cannot be done in a single cycle, but loading an immediate into a branch register or moving to or from a general purpose register can be done as follows.

```
c0 cmpeq $b0.dest = $r0.0, $r0.0       // Load true
c0 cmpne $b0.dest = $r0.0, $r0.0       // Load false
c0 cmpne $b0.dest = $r0.0, $r0.src     // Move general purpose to branch
c0 slctf $r0.dest = $b0.src, $r0.0, 1  // Move branch to general purpose
```

Branch register can also not be loaded from or stored into memory on their own. However, to improve context switching speed slightly, the `LDBR` and `STBR` instructions are available. These load or store a byte containing all eight branch registers in a single syllable.

3.2.3 Link register

The link register is a 32-bit register used to store the return address when calling. It can also be used as the destination address for an unconditional indirect jump or call, in cases where the branch offset field is too small or when the jump target is determined at runtime.

When general purpose register 63 is not mapped to the link register, the `MOVTL` and `LDW` instructions can be used to load the link register from a general purpose register or memory respectively. `MOVFL` and `STW` perform the reverse operations.

3.2.4 Global and context control registers

These two register files contain special-purpose registers. The global control registers contain status information not specific to any context, whereas the context control registers are context specific.

The processor can access these register files through memory operations only. All these accesses are single-cycle. 1 kiB of memory space has to be reserved for this purpose, usually mapped to `0xFFFFFC00..0xFFFFFFFF`. The location of the block is design-time configurable. Note that it is impossible for the processor to perform actual memory operations to this region, so the location of the block should be chosen wisely.

The global register file is read-only from the perspective of the program. The context register file is writable, but it should be noted that each program can only access its own hardware context register file. If an application requires that programs can write to the global register file or the other context register files, the debug bus can be made accessible for memory operations by the bus interconnect outside the core. In most platforms this happens coincidentally, as the processor can access the main bus of the platform, and the debug bus is wired as a slave peripheral on this bus. For more information about the debug bus, refer to Section 9.2.2.7.

For more information about the control registers in general, refer to Section 4.

3.3 Memory

Each lane group of the ρ-VEX processor currently has exactly one memory unit. The configurability of this may be extended in the future, as memory operations commonly end up being the critical path when extracting instruction-level parallelism. However, doing so would require significant modifications to the ρ-VEX core and the reconfigurable cache.

The ρ-VEX processor is big endian. This means that when accessing a 32-bit or 16-bit word, the most significant byte will reside in the lowest address. This is the opposite of what you may be used to coming from x86.

The ρ-VEX processor is capable of reading and writing 32-bit, 16-bit and 8-bit words. Seperate read instructions exist for reading 16-bit and 8-bit words in signed or unsigned mode. All n-bit accesses must be n-bit aligned. If an access is improperly aligned, a `MISALIGNED_ACCESS` trap will be caused.

Note that a 1 kiB block of the external memory space must be selected to be remapped to the control register file internally. This prevents the processor from being able to access the block. Refer to Section 3.2.4 for more information.

3.4 Syllable resource classes and delays

Some syllables take more than one cycle to complete. In this case, they are always pipelined; no multi-cycle syllable will stall the rest of the bundle. While this is good for performance, it does require the attention of the programmer in order to write properly functioning code.

In addition, not all pipelanes support execution of all syllables, and as such, requirements are imposed on the position of certain syllables within a bundle in the binary. The assembler will normally ensure that these requirements are met, unlike the delay requirements, which it cannot detect. However, it is still important for the user to know them in order to be able to write assembly.

It is also important that the assembler is configured in the same way as the core. If there are discrepancies, the assembler may still output binaries that the core cannot execute. If this happens, the core will produce an `TRAP_INVALID_OP` trap, with the index of the offending pipelane as the trap argument.

There are five distinguishable classes of syllables. These classes are ALU, multiply, memory, branch and long immediate.

3.4.1 ALU class

ALU syllables are the basic ρ-VEX instructions. They can be processed by every lane. In the default pipeline and forwarding configuration of the ρ-VEX, their results are available after a single cycle. That is, the bundle immediately following can use their results.

3.4.2 Multiply class

Multiply syllables are only allowed in lanes that are configured to have a multiplier. This configuration is done at design time using generics. In the default configuration, every lane has a multiplication unit.

In the default pipeline and forwarding configuration of the ρ-VEX, multiply instructions are two-cycle pipelined. That is, two bundle boundaries are needed between the syllable producing the value and a syllable that uses it.

3.4.3 Memory class

Memory syllables are only allowed in lanes that are configured to have a memory unit. In addition, in most configurations, only one memory unit can be active per context at a time, even if multiple are available. This is due to the fact the data cache can only perform one operation per cycle per context. In theory, it is still permissible in such a system to perform a single memory operation and a single control register operation at the same time, but there is currently no toolchain support for this.

In the default pipeline and forwarding configuration of the ρ-VEX, memory load instructions are two-cycle pipelined. That is, two bundle boundaries are needed between a load syllable and the first syllable that uses the value. However, there is no store to load delay; if a bundle with a load of a certain address immediately follows a bundle that stores a value at that address, the newly written value is loaded.

3.4.4 Branch class

All syllables that affect the program counter are considered branch syllables. Only one branch syllable is permitted per cycle, and in almost all design-time core configurations, it must be the last syllable in a bundle.

In the previous ρ-VEX version, a delay was needed between a syllable producing a branch register or link register value and branch operations. This is not the case in the default pipeline and forwarding configuration of this ρ-VEX version, as the ALU and branch operations are initiated in the same pipeline stage.

3.4.5 Long immediate class

Sometimes, one syllable does not contain enough information for one pipelane to execute. The only time when this happens in the ρ-VEX processor is when an immediate outside the range -256..255 is to be specified. For this purpose, `LIMMH` syllables exist. These syllables perform no operation in their own pipelane, but instead send 23 additional immediate bits to another lane, which allows a 32-bit immediate to be used in a single cycle.

Any ALU, multiply or memory syllable that supports an immediate can receive a long immediate. However, long immediates can *not* be used to extend the branch offset field.

`LIMMH` syllables are automatically inferred by the assembler. However, each `LIMMH` syllable inferred means that one less functional syllable can be scheduled in a single bundle. In addition, a certain pipelane can not 'send' a long immediate to any other pipelane.

The ρ-VEX supports two routes for long immediates to take. They are called 'long immediate from neighbor' and 'long immediate from previous pair'. One or both of these methods may be enabled at design time using generics.

Long immediate from neighbor

This is the most common route, as it is supported in all ρ-VEX configurations. This allows all pipelanes to forward a long immediate to their immediate neighbor within a pair of pipelanes. This is depicted in Figure 3.2 for an 8-way ρ-VEX processor.

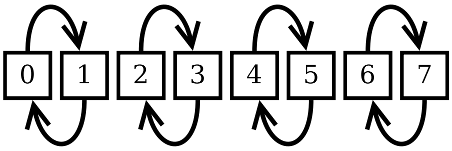


Figure 3.1: Long immediate from neighbor routing.

Long immediate from previous pair

This provides an alternative place where a long immediate can be placed for lanes 2 and up; when this route is enabled lane n can send a long immediate to lane $n + 2$. This is depicted in Figure 3.2 for an 8-way ρ-VEX processor. However, due to limitations in the instruction fetch unit, this system is incompatible with the stop bit system. For these reasons, it can only be effectively used in cores with at least four lanes that are not configured to support stop bits.

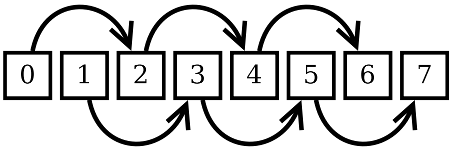


Figure 3.2: Long immediate from previous pair routing.

3.5 Generic binaries

Generic binaries are binaries that can be correctly run on different core configurations, even if the core reconfigures during execution. They were introduced in [2]. Typically, a generic binary refers to a binary that can be run with two pipelanes (2-way), four pipelanes (4-way) or eight pipelanes (8-way).

A generic binary is typically compiled in the same way as a regular 8-way binary. It is the task of the assembler to ensure that the generic binary requirements are met. For the standard generic binary, these rules are the following.

- *The single branch instruction allowed per bundle must end up in the last execution cycle in 2-way and 4-way execution.* The ρ-VEX processor imposes the even stricter requirement that branch syllables must always be the last syllable in a bundle.
- *RAW hazards must be avoided in all runtime configurations.* That is, for example, a register that is written in one of the first two syllables may not be read in subsequent slots. This is because the old value of the register would be read in 8-way mode, but the newly written value would be read in 2-way mode.

Extrapolating these rules to the general case should be trivial.

3.5.1 Generating generic binaries

In order to generate generic binaries, the `-u` flag needs to be passed to the assembler. By default, the assembler will only try to move syllables around within bundles in order to meet the requirements imposed above. However, often this is not possible without further processing.

There are two ways to process the assembly files to meet the requirements. The first one can be done by the assembler as well. If the `-autosplit` flag is passed, it will attempt to split bundles that it cannot schedule directly. This solves most problems at the cost of runtime performance. Refer to [2] for more information.

The second way involves running a python script called `vexparse` on the assembly compilation output, before passing them to the assembler. Depending on its configuration, `vexparse` will extract a dependency graph of all syllables in a basic block[1] from the assembly code, and then completely reschedule all instructions. As a side effect, it will fix hand-written assembly code that failed to take multiply and load instruction delays into consideration.

Being a python script, `vexparse` is much slower than the `-autosplit` option of the assembler. However, it generates more efficient code, as it is not limited to merely splitting bundles.

3.6 Stop bits

The stop bit system is the colloquial name for the binary compression algorithm that the core may be design-time configured to support. It refers to a bit present in every syllable, which, if set, marks the syllable as the last syllable in the current bundle. In contrast, when the stop bit system is not used, bundle boundaries are based on alignment; each bundle is expected to start on an alignment boundary of the maximum size of a bundle. `NOP` instructions are then used to fill the unused words. The stop bit should then still be set in the last syllable, as failing to do so will cause a trap if the bundle contains a branch syllable.

The major advantage of stop bits is the decreased size of the binary. This does not only mean that the memory footprint of a program will be smaller; memory is cheap, so this is usually not an issue. More importantly, it means that the processor will need to do less instruction memory accesses for the same amount of computation; memory bandwidth and caches *are* expensive.

There is an additional benefit when combined with generic binaries. When a generic binary without stop bits runs in 8-way mode, the `NOP` instructions needed for bundle alignment do not cause any delays in execution, aside from the implicit delays due to the strain on the instruction memory system. However, when the binary is run in 2-way mode, these alignment `NOP`s may actually cost cycles. To illustrate, imagine an 8-way generic binary bundle with only two syllables used. When this bundle is executed in 2-way mode, execution will necessarily still take four cycles, because the processor still

[1] A basic block is a block of instructions with natural scheduling boundaries at the start and end of it. The prime example of such boundaries are branch instructions.

needs to work through eight syllables.[2] When stop bits are enabled, such alignment `NOP`s do not exist, so they will naturally never waste cycles.

The major disadvantage of using stop bits is its hardware complexity. Without stop bits, the core naturally always fetches a nicely aligned block of instruction memory to process. Each 32-bit word in this block can be wired directly to the syllable input of each lane. In contrast, when stop bits are fully enabled, a bundle may start on any 32-bit word boundary. Thus, a new module is needed between the instruction memory (which expects accesses aligned to its access size) and the pipelanes. This module must then be capable of routing any incoming 32-bit word to any pipelane, based on the lower bits of the current program counter and even the syllable type, as branch syllables always need to be routed to the last pipelane. It must also store the previous fetch to handle misaligned bundles, and when a branch to a misaligned address occurs, it must stall execution for an additional cycle, as it will have to fetch both the memory block before and after the crossed alignment boundary.

On the plus side, the large multiplexers involved in this instruction buffer do not increase in size when adding reconfiguration capabilities to an 8-way core with stop bits. Some additional control logic is obviously required, but nothing more.

3.6.1 Design-time configuration

The ρ-VEX processor core allows the designer to make a compromise between the large binary size without stop bits and the additional hardware needed with stop bits. Instead of simply supporting stop bits or not, the stop bit system is configured by specifying the bundle alignment boundaries that the core may expect. When the bundle alignment boundaries equal the size of the maximum bundle size, stop bits are effectively disabled. When the alignment boundary is set to 32-bit words, stop bits are fully enabled. Midway configuration are supported equally well.

Every time the bundle alignment boundary is halved, the multiplexers in the syllable dispatch logic double in size. The complexity of program counter generation increases with each step as well, as does the instruction fetch buffer size. Meanwhile, the number of alignment `NOP`s required in the binary decreases with each step.

The default 8-way reconfigurable core with stop bits enabled have the bundle alignment boundary set to 64-bit. Going all the way to 32-bit boundaries does not increase 2-way execution performance of an 8-way generic binary further, and most `NOP`s have already been eliminated, so doubling the hardware complexity once more is generally not justifiable.

[2] It is certainly possible to avoid this without a complete stop bit system. For example, for the previous version of the ρ-VEX processor, it was proposed to use the stop bits to mark the end of the useful part of a bundle, instead of the actual boundaries. In the case of our 8-way bundle with only two syllables used, assuming the two syllables can be placed in the first two slots, the stop bit would be set in the second syllable instead of the eighth. When this code is executed in 2-way mode, the ρ-VEX processor would recognize that it can jump to the next 8-way bundle alignment boundary, thus skipping the six `NOP` syllables.

3.7 Instruction set

The ρ-VEX instruction set consists of 169 instructions. These instructions are defined by two bitfields in the syllable, called `opcode` and `imm_sw`. The `opcode` field is 8 bits in size, ranging from bit 31 to 24 inclusive, allowing for 256 different operations to be performed. `imm_sw` is a single bit (bit 23) that specifies if the second operand is a register or an immediate. This thus allows a total of 512 different instructions in theory.

However, not all operations support both register and immediate mode. In addition, some instructions have operand fields that extend into the `opcode`, requiring a single instruction to use multiple opcodes. Taking these things into consideration, the ρ-VEX instruction set has 113 `opcodes` that are not yet mapped.

There are two additional fields with a fixed function within the instruction set. The first is the stop bit, bit 1. This bit determines where the bundle boundaries are. Refer to Section 3.6 for more information. The second field, bit 0, is reserved for cluster end bits. The toolchain currently always outputs a 0 bit, and the processor ignores it completely.

The following table lists all the instructions in the ρ-VEX instruction set ordered by opcode. The subsequent sections document each instruction, ordered by function. If you are reading this document digitally, you can click any instruction in the table to jump to its documentation.

31 30 29 28 27 26 25 24	23	22 21 20 19 18 17	16 15 14 13 12 11	10 9 8 7 6 5	4 3 2	1	0	
0 0 0 0 0 0 0 0	0	d	x	y		S		`mpyll $r0.d = $r0.x, $r0.y`
0 0 0 0 0 0 0 0	1	d	x	imm		S		`mpyll $r0.d = $r0.x, imm`
0 0 0 0 0 0 0 1	0	d	x	y		S		`mpyllu $r0.d = $r0.x, $r0.y`
0 0 0 0 0 0 0 1	1	d	x	imm		S		`mpyllu $r0.d = $r0.x, imm`
0 0 0 0 0 0 1 0	0	d	x	y		S		`mpylh $r0.d = $r0.x, $r0.y`
0 0 0 0 0 0 1 0	1	d	x	imm		S		`mpylh $r0.d = $r0.x, imm`
0 0 0 0 0 0 1 1	0	d	x	y		S		`mpylhu $r0.d = $r0.x, $r0.y`
0 0 0 0 0 0 1 1	1	d	x	imm		S		`mpylhu $r0.d = $r0.x, imm`
0 0 0 0 0 1 0 0	0	d	x	y		S		`mpyhh $r0.d = $r0.x, $r0.y`
0 0 0 0 0 1 0 0	1	d	x	imm		S		`mpyhh $r0.d = $r0.x, imm`
0 0 0 0 0 1 0 1	0	d	x	y		S		`mpyhhu $r0.d = $r0.x, $r0.y`
0 0 0 0 0 1 0 1	1	d	x	imm		S		`mpyhhu $r0.d = $r0.x, imm`
0 0 0 0 0 1 1 0	0	d	x	y		S		`mpyl $r0.d = $r0.x, $r0.y`
0 0 0 0 0 1 1 0	1	d	x	imm		S		`mpyl $r0.d = $r0.x, imm`
0 0 0 0 0 1 1 1	0	d	x	y		S		`mpylu $r0.d = $r0.x, $r0.y`
0 0 0 0 0 1 1 1	1	d	x	imm		S		`mpylu $r0.d = $r0.x, imm`
0 0 0 0 1 0 0 0	0	d	x	y		S		`mpyh $r0.d = $r0.x, $r0.y`
0 0 0 0 1 0 0 0	1	d	x	imm		S		`mpyh $r0.d = $r0.x, imm`
0 0 0 0 1 0 0 1	0	d	x	y		S		`mpyhu $r0.d = $r0.x, $r0.y`
0 0 0 0 1 0 0 1	1	d	x	imm		S		`mpyhu $r0.d = $r0.x, imm`
0 0 0 0 1 0 1 0	0	d	x	y		S		`mpyhs $r0.d = $r0.x, $r0.y`
0 0 0 0 1 0 1 0	1	d	x	imm		S		`mpyhs $r0.d = $r0.x, imm`
0 0 0 0 1 0 1 1	0			y		S		`movtl $l0.0 = $r0.y`
0 0 0 0 1 0 1 1	1			imm		S		`movtl $l0.0 = imm`
0 0 0 0 1 1 0 0	0	d				S		`movfl $r0.d = $l0.0`
0 0 0 0 1 1 0 1	1		x	imm		S		`ldw $l0.0 = imm[$r0.x]`
0 0 0 0 1 1 1 0	1		x	imm		S		`stw imm[$r0.x] = $l0.0`
0 0 0 1 0 0 0 0	1	d	x	imm		S		`ldw $r0.d = imm[$r0.x]`
0 0 0 1 0 0 0 1	1	d	x	imm		S		`ldh $r0.d = imm[$r0.x]`
0 0 0 1 0 0 1 0	1	d	x	imm		S		`ldhu $r0.d = imm[$r0.x]`

31	30	29	28	27	26	25	24	23	22 21 20	19 18 17	16 15 14 13 12 11	10 9 8 7 6 5	4 3 2	1	0	
0	0	0	1	0	0	1	1	1	d		x	imm		S		`ldb $r0.d = imm[$r0.x]`
0	0	0	1	0	1	0	0	1	d		x	imm		S		`ldbu $r0.d = imm[$r0.x]`
0	0	0	1	0	1	0	1	1	d		x	imm		S		`stw imm[$r0.x] = $r0.d`
0	0	0	1	0	1	1	0	1	d		x	imm		S		`sth imm[$r0.x] = $r0.d`
0	0	0	1	0	1	1	1	1	d		x	imm		S		`stb imm[$r0.x] = $r0.d`
0	0	0	1	1	0	0	0	0	d		x	y		S		`shr $r0.d = $r0.x, $r0.y`
0	0	0	1	1	0	0	0	1	d		x	imm		S		`shr $r0.d = $r0.x, imm`
0	0	0	1	1	0	0	1	0	d		x	y		S		`shru $r0.d = $r0.x, $r0.y`
0	0	0	1	1	0	0	1	1	d		x	imm		S		`shru $r0.d = $r0.x, imm`
0	0	0	1	1	0	1	0	0	d		x	y		S		`sub $r0.d = $r0.y, $r0.x`
0	0	0	1	1	0	1	0	1	d		x	imm		S		`sub $r0.d = imm, $r0.x`
0	0	0	1	1	0	1	1	0	d		x			S		`sxtb $r0.d = $r0.x`
0	0	0	1	1	1	0	0	0	d		x			S		`sxth $r0.d = $r0.x`
0	0	0	1	1	1	0	1	0	d		x			S		`zxtb $r0.d = $r0.x`
0	0	0	1	1	1	1	0	0	d		x			S		`zxth $r0.d = $r0.x`
0	0	0	1	1	1	1	1	0	d		x	y		S		`xor $r0.d = $r0.x, $r0.y`
0	0	0	1	1	1	1	1	1	d		x	imm		S		`xor $r0.d = $r0.x, imm`
0	0	1	0	0	0	0	0	offs						S		`goto offs`
0	0	1	0	0	0	0	1							S		`igoto $l0.0`
0	0	1	0	0	0	1	0	offs						S		`call $l0.0 = offs`
0	0	1	0	0	0	1	1							S		`icall $l0.0 = $l0.0`
0	0	1	0	0	1	0	0	offs					bs	S		`br $b0.bs, offs`
0	0	1	0	0	1	0	1	offs					bs	S		`brf $b0.bs, offs`
0	0	1	0	0	1	1	0	stackadj						S		`return $r0.1 = $r0.1, stackadj, $l0.0`
0	0	1	0	0	1	1	1	stackadj						S		`rfi $r0.1 = $r0.1, stackadj`
0	0	1	0	1	0	0	0							S		`stop`
0	0	1	0	1	1	0	0	0	d		x	y		S		`sbit $r0.d = $r0.x, $r0.y`
0	0	1	0	1	1	0	0	1	d		x	imm		S		`sbit $r0.d = $r0.x, imm`
0	0	1	0	1	1	0	1	0	d		x	y		S		`sbitf $r0.d = $r0.x, $r0.y`
0	0	1	0	1	1	0	1	1	d		x	imm		S		`sbitf $r0.d = $r0.x, imm`
0	0	1	0	1	1	1	0	1			x	imm		S		`ldbr imm[$r0.x]`
0	0	1	0	1	1	1	1	1			x	imm		S		`stbr imm[$r0.x]`
0	0	1	1	0	bs			0	d		x	y		S		`slctf $r0.d = $b0.bs, $r0.x, $r0.y`
0	0	1	1	0	bs			1	d		x	imm		S		`slctf $r0.d = $b0.bs, $r0.x, imm`
0	0	1	1	1	bs			0	d		x	y		S		`slct $r0.d = $b0.bs, $r0.x, $r0.y`
0	0	1	1	1	bs			1	d		x	imm		S		`slct $r0.d = $b0.bs, $r0.x, imm`
0	1	0	0	0	0	0	0	0	d		x	y		S		`cmpeq $r0.d = $r0.x, $r0.y`
0	1	0	0	0	0	0	0	1	d		x	imm		S		`cmpeq $r0.d = $r0.x, imm`
0	1	0	0	0	0	0	1	0		bd	x	y		S		`cmpeq $b0.bd = $r0.x, $r0.y`
0	1	0	0	0	0	0	1	1		bd	x	imm		S		`cmpeq $b0.bd = $r0.x, imm`
0	1	0	0	0	0	1	0	0	d		x	y		S		`cmpge $r0.d = $r0.x, $r0.y`
0	1	0	0	0	0	1	0	1	d		x	imm		S		`cmpge $r0.d = $r0.x, imm`
0	1	0	0	0	0	1	1	0		bd	x	y		S		`cmpge $b0.bd = $r0.x, $r0.y`
0	1	0	0	0	0	1	1	1		bd	x	imm		S		`cmpge $b0.bd = $r0.x, imm`
0	1	0	0	0	1	0	0	0	d		x	y		S		`cmpgeu $r0.d = $r0.x, $r0.y`
0	1	0	0	0	1	0	0	1	d		x	imm		S		`cmpgeu $r0.d = $r0.x, imm`
0	1	0	0	0	1	0	1	0		bd	x	y		S		`cmpgeu $b0.bd = $r0.x, $r0.y`
0	1	0	0	0	1	0	1	1		bd	x	imm		S		`cmpgeu $b0.bd = $r0.x, imm`
0	1	0	0	0	1	1	0	0	d		x	y		S		`cmpgt $r0.d = $r0.x, $r0.y`
0	1	0	0	0	1	1	0	1	d		x	imm		S		`cmpgt $r0.d = $r0.x, imm`
0	1	0	0	0	1	1	1	0		bd	x	y		S		`cmpgt $b0.bd = $r0.x, $r0.y`
0	1	0	0	0	1	1	1	1		bd	x	imm		S		`cmpgt $b0.bd = $r0.x, imm`
0	1	0	0	1	0	0	0	0	d		x	y		S		`cmpgtu $r0.d = $r0.x, $r0.y`
0	1	0	0	1	0	0	0	1	d		x	imm		S		`cmpgtu $r0.d = $r0.x, imm`

31	30	29	28	27	26	25	24	23	22 21 20	19 18 17	16 15 14 13 12 11	10 9 8 7 6 5	4 3 2	1	0	
0	1	0	0	1	0	0	1	0		bd	x	y		S		cmpgtu $b0.bd = $r0.x, $r0.y
0	1	0	0	1	0	0	1	1		bd	x	imm		S		cmpgtu $b0.bd = $r0.x, imm
0	1	0	0	1	0	1	0	0	d		x	y		S		cmple $r0.d = $r0.x, $r0.y
0	1	0	0	1	0	1	0	1	d		x	imm		S		cmple $r0.d = $r0.x, imm
0	1	0	0	1	0	1	1	0		bd	x	y		S		cmple $b0.bd = $r0.x, $r0.y
0	1	0	0	1	0	1	1	1		bd	x	imm		S		cmple $b0.bd = $r0.x, imm
0	1	0	0	1	1	0	0	0	d		x	y		S		cmpleu $r0.d = $r0.x, $r0.y
0	1	0	0	1	1	0	0	1	d		x	imm		S		cmpleu $r0.d = $r0.x, imm
0	1	0	0	1	1	0	1	0		bd	x	y		S		cmpleu $b0.bd = $r0.x, $r0.y
0	1	0	0	1	1	0	1	1		bd	x	imm		S		cmpleu $b0.bd = $r0.x, imm
0	1	0	0	1	1	1	0	0	d		x	y		S		cmplt $r0.d = $r0.x, $r0.y
0	1	0	0	1	1	1	0	1	d		x	imm		S		cmplt $r0.d = $r0.x, imm
0	1	0	0	1	1	1	1	0		bd	x	y		S		cmplt $b0.bd = $r0.x, $r0.y
0	1	0	0	1	1	1	1	1		bd	x	imm		S		cmplt $b0.bd = $r0.x, imm
0	1	0	1	0	0	0	0	0	d		x	y		S		cmpltu $r0.d = $r0.x, $r0.y
0	1	0	1	0	0	0	0	1	d		x	imm		S		cmpltu $r0.d = $r0.x, imm
0	1	0	1	0	0	0	1	0		bd	x	y		S		cmpltu $b0.bd = $r0.x, $r0.y
0	1	0	1	0	0	0	1	1		bd	x	imm		S		cmpltu $b0.bd = $r0.x, imm
0	1	0	1	0	0	1	0	0	d		x	y		S		cmpne $r0.d = $r0.x, $r0.y
0	1	0	1	0	0	1	0	1	d		x	imm		S		cmpne $r0.d = $r0.x, imm
0	1	0	1	0	0	1	1	0		bd	x	y		S		cmpne $b0.bd = $r0.x, $r0.y
0	1	0	1	0	0	1	1	1		bd	x	imm		S		cmpne $b0.bd = $r0.x, imm
0	1	0	1	0	1	0	0	0	d		x	y		S		nandl $r0.d = $r0.x, $r0.y
0	1	0	1	0	1	0	0	1	d		x	imm		S		nandl $r0.d = $r0.x, imm
0	1	0	1	0	1	0	1	0		bd	x	y		S		nandl $b0.bd = $r0.x, $r0.y
0	1	0	1	0	1	0	1	1		bd	x	imm		S		nandl $b0.bd = $r0.x, imm
0	1	0	1	0	1	1	0	0	d		x	y		S		norl $r0.d = $r0.x, $r0.y
0	1	0	1	0	1	1	0	1	d		x	imm		S		norl $r0.d = $r0.x, imm
0	1	0	1	0	1	1	1	0		bd	x	y		S		norl $b0.bd = $r0.x, $r0.y
0	1	0	1	0	1	1	1	1		bd	x	imm		S		norl $b0.bd = $r0.x, imm
0	1	0	1	1	0	0	0	0	d		x	y		S		orl $r0.d = $r0.x, $r0.y
0	1	0	1	1	0	0	0	1	d		x	imm		S		orl $r0.d = $r0.x, imm
0	1	0	1	1	0	0	1	0		bd	x	y		S		orl $b0.bd = $r0.x, $r0.y
0	1	0	1	1	0	0	1	1		bd	x	imm		S		orl $b0.bd = $r0.x, imm
0	1	0	1	1	0	1	0	0	d		x	y		S		andl $r0.d = $r0.x, $r0.y
0	1	0	1	1	0	1	0	1	d		x	imm		S		andl $r0.d = $r0.x, imm
0	1	0	1	1	0	1	1	0		bd	x	y		S		andl $b0.bd = $r0.x, $r0.y
0	1	0	1	1	0	1	1	1		bd	x	imm		S		andl $b0.bd = $r0.x, imm
0	1	0	1	1	1	0	0	0	d		x	y		S		tbit $r0.d = $r0.x, $r0.y
0	1	0	1	1	1	0	0	1	d		x	imm		S		tbit $r0.d = $r0.x, imm
0	1	0	1	1	1	0	1	0		bd	x	y		S		tbit $b0.bd = $r0.x, $r0.y
0	1	0	1	1	1	0	1	1		bd	x	imm		S		tbit $b0.bd = $r0.x, imm
0	1	0	1	1	1	1	0	0	d		x	y		S		tbitf $r0.d = $r0.x, $r0.y
0	1	0	1	1	1	1	0	1	d		x	imm		S		tbitf $r0.d = $r0.x, imm
0	1	0	1	1	1	1	1	0		bd	x	y		S		tbitf $b0.bd = $r0.x, $r0.y
0	1	0	1	1	1	1	1	1		bd	x	imm		S		tbitf $b0.bd = $r0.x, imm
0	1	1	0	0	0	0	0							S		nop
0	1	1	0	0	0	1	0	0	d		x	y		S		add $r0.d = $r0.x, $r0.y
0	1	1	0	0	0	1	0	1	d		x	imm		S		add $r0.d = $r0.x, imm
0	1	1	0	0	0	1	1	0	d		x	y		S		and $r0.d = $r0.x, $r0.y
0	1	1	0	0	0	1	1	1	d		x	imm		S		and $r0.d = $r0.x, imm
0	1	1	0	0	1	0	0	0	d		x	y		S		andc $r0.d = $r0.x, $r0.y
0	1	1	0	0	1	0	0	1	d		x	imm		S		andc $r0.d = $r0.x, imm
0	1	1	0	0	1	0	1	0	d		x	y		S		max $r0.d = $r0.x, $r0.y

31 30 29 28 27 26 25 24	23	22 21 20 19 18 17	16 15 14 13 12 11	10 9 8 7 6 5	4 3 2	1	0	
0 1 1 0 0 1 0 1	1	d	x	imm		S		`max $r0.d = $r0.x, imm`
0 1 1 0 0 1 1 0	0	d	x	y		S		`maxu $r0.d = $r0.x, $r0.y`
0 1 1 0 0 1 1 0	1	d	x	imm		S		`maxu $r0.d = $r0.x, imm`
0 1 1 0 0 1 1 1	0	d	x	y		S		`min $r0.d = $r0.x, $r0.y`
0 1 1 0 0 1 1 1	1	d	x	imm		S		`min $r0.d = $r0.x, imm`
0 1 1 0 1 0 0 0	0	d	x	y		S		`minu $r0.d = $r0.x, $r0.y`
0 1 1 0 1 0 0 0	1	d	x	imm		S		`minu $r0.d = $r0.x, imm`
0 1 1 0 1 0 0 1	0	d	x	y		S		`or $r0.d = $r0.x, $r0.y`
0 1 1 0 1 0 0 1	1	d	x	imm		S		`or $r0.d = $r0.x, imm`
0 1 1 0 1 0 1 0	0	d	x	y		S		`orc $r0.d = $r0.x, $r0.y`
0 1 1 0 1 0 1 0	1	d	x	imm		S		`orc $r0.d = $r0.x, imm`
0 1 1 0 1 0 1 1	0	d	x	y		S		`sh1add $r0.d = $r0.x, $r0.y`
0 1 1 0 1 0 1 1	1	d	x	imm		S		`sh1add $r0.d = $r0.x, imm`
0 1 1 0 1 1 0 0	0	d	x	y		S		`sh2add $r0.d = $r0.x, $r0.y`
0 1 1 0 1 1 0 0	1	d	x	imm		S		`sh2add $r0.d = $r0.x, imm`
0 1 1 0 1 1 0 1	0	d	x	y		S		`sh3add $r0.d = $r0.x, $r0.y`
0 1 1 0 1 1 0 1	1	d	x	imm		S		`sh3add $r0.d = $r0.x, imm`
0 1 1 0 1 1 1 0	0	d	x	y		S		`sh4add $r0.d = $r0.x, $r0.y`
0 1 1 0 1 1 1 0	1	d	x	imm		S		`sh4add $r0.d = $r0.x, imm`
0 1 1 0 1 1 1 1	0	d	x	y		S		`shl $r0.d = $r0.x, $r0.y`
0 1 1 0 1 1 1 1	1	d	x	imm		S		`shl $r0.d = $r0.x, imm`
0 1 1 1 0 bs	0	d	x	y	bd	S		`divs $r0.d, $b0.bd = $b0.bs, $r0.x, $r0.y`
0 1 1 1 1 bs	0	d	x	y	bd	S		`addcg $r0.d, $b0.bd = $b0.bs, $r0.x, $r0.y`
1 0 0 0 tgt			imm			S		`limmh tgt, imm`
1 0 0 1 0 0 0 0	0		x	y		S		`trap $r0.x, $r0.y`
1 0 0 1 0 0 0 0	1		x	imm		S		`trap $r0.x, imm`
1 0 0 1 0 0 0 1	0	d	x			S		`clz $r0.d = $r0.x`
1 0 0 1 0 0 1 0	0	d	x	y		S		`mpylhus $r0.d = $r0.x, $r0.y`
1 0 0 1 0 0 1 0	1	d	x	imm		S		`mpylhus $r0.d = $r0.x, imm`
1 0 0 1 0 0 1 1	0	d	x	y		S		`mpyhhs $r0.d = $r0.x, $r0.y`
1 0 0 1 0 0 1 1	1	d	x	imm		S		`mpyhhs $r0.d = $r0.x, imm`

3.7.1 ALU arithmetic instructions

The ρ-VEX ALU has a 32-bit adder for arithmetic. Some exotic instructions are available to make efficient multiplications by small constants and to speed up software divisions.

`add $r0.d = $r0.x, $r0.y`

`add $r0.d = $r0.x, imm`

31 30 29 28 27 26 25 24	23	22 21 20 19 18 17	16 15 14 13 12 11	10 9 8 7 6 5	4 3 2	1	0
0 1 1 0 0 0 1 0	0	d	x	y		S	
0 1 1 0 0 0 1 0	1	d	x	imm		S	

Performs a 32-bit addition. Notice that `ADD` instructions may be used as move or load immediate operations when `x` is set to 0. While the `OR` instruction is often used instead, there is no functional difference between the two when used inthis way.

```
$r0.d = $r0.x + [$r0.y|imm];
```

sh1add $r0.d = $r0.x, $r0.y

sh1add $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	0	1	1	0	d						x						y									S	
0	1	1	0	1	0	1	1	1	d						x						imm									S	

Performs a 32-bit addition. `$r0.x` is first left-shifted by one.

```
$r0.d = ($r0.x << 1) + [$r0.y|imm];
```

sh2add $r0.d = $r0.x, $r0.y

sh2add $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	1	0	0	0	d						x						y									S	
0	1	1	0	1	1	0	0	1	d						x						imm									S	

Performs a 32-bit addition. `$r0.x` is first left-shifted by two.

```
$r0.d = ($r0.x << 2) + [$r0.y|imm];
```

sh3add $r0.d = $r0.x, $r0.y

sh3add $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	1	0	1	0	d						x						y									S	
0	1	1	0	1	1	0	1	1	d						x						imm									S	

Performs a 32-bit addition. `$r0.x` is first left-shifted by three.

```
$r0.d = ($r0.x << 3) + [$r0.y|imm];
```

sh4add $r0.d = $r0.x, $r0.y

sh4add $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	1	1	0	0	d						x						y									S	
0	1	1	0	1	1	1	0	1	d						x						imm									S	

Performs a 32-bit addition. `$r0.x` is first left-shifted by four.

```
$r0.d = ($r0.x << 4) + [$r0.y|imm];
```

sub $r0.d = $r0.y, $r0.x

sub $r0.d = imm, $r0.x

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0	1	0	0	d						x						y									S	
0	0	0	1	1	0	1	0	1	d						x						imm									S	

Performs a 32-bit subtraction. Note that, unlike all other instructions, the immediate must be specified first. This allows `SUB` to be used to subtract a register from an immediate.

Notice that `SUB` reduces to two's complement negation when `x` or `imm` equal zero.

```
$r0.d = [$r0.y|imm] + $r0.x;
```

addcg $r0.d, $b0.bd = $b0.bs, $r0.x, $r0.y

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	bs			0	d						x						y						bd			s	

Primitive for additions of integers wider than 32 bits. Addition is performed by first setting a scratch branch register to false using CMPNE for the carry input. Then ADDCG can be used to add up words together one by one with increasing significance, using the scratch branch register for the carry chain.

Subtractions can be performed by setting the carry input to 1 using CMPEQ and ones-complementing one of the inputs using XOR.

Notice that ADDCG reduces to rotate left by one through a branch register when x equals y. This may be used for for shift left by one operations on integers wider than 32 bits.

```
long long tmp = $r0.x + $r0.y + ($b0.bs ? 1 : 0);
$r0.d = (int)tmp;
$b0.bd = (tmp & 0x100000000) != 0;
```

divs $r0.d, $b0.bd = $b0.bs, $r0.x, $r0.y

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	bs			0	d						x						y						bd			s	

Primitive for integer divisions. The following assembly code performs a single nonrestoring division step.

```
addcg  s_lo, shift_bit    = s_lo, s_lo, <zero>
;;
divs   s_hi, quotient_bit = s_hi, divisor, shift_bit
;;
addcg  quotient, <unused> = quotient, quotient, quotient_bit
;;
```

Here, s_lo and s_hi represent the partial remainder, initialized to the dividend before the first division step. The first ADDCG and the DIVS instruction together perform a 64-bit shift-left-by-1 operation. Depending on the sign of the partial remainder before the shift (i.e., the MSB which was shifted out), the dividend is added to or subtracted from the partial remainder. If an addition was performed, quotient_bit is set to 1, representing that the current quotient bit is -1 in binary signed digit representation. Otherwise, the current quotient bit is 1. The final ADDCG stores the result by left-shifting the bit into quotient. Post-processing is required to convert the quotient from binary signed digit representation to two's complement.

It should be noted that a division step can be done in a single cycle using just two syllables. This division step has to be applied many times in a loop, and benefits from modulo-scheduling (also known as software pipelining), allowing each step to be performed in a single cycle. Furthermore, the quotient_bits can be shifted into s_lo instead of a different register, as the zero bits shifted into s_lo are unused. This eliminates the need for the second ADDCG, as well as the zero and scratch branch registers.

The prologue and epilogue code for various divisions are beyond the scope of this manual, as the compilers take care of expanding divisions.

```
int tmp = ($r0.x << 1) | ($b0.bs ? 1 : 0);
bool flag = ($r0.x & 0x80000000) != 0;
$r0.d = flag ? (tmp + $r0.y) : (tmp - $r0.y);
$b0.bd = flag;
```

3.7.2 ALU barrel shifter instructions

The ρ-VEX ALU includes a barrel shifter. It should be noted that the shift amount input to the barrel shifter is 8-bit unsigned, not 32-bit as one might expect. That is, the upper 24 bits of the shift amount are discarded, and for instance a left shift by a negative amount will *not* simply result in a right shift.

`shl $r0.d = $r0.x, $r0.y`

`shl $r0.d = $r0.x, imm`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	1	1	1	0	d						x						y									s	
0	1	1	0	1	1	1	1	1	d						x						imm									s	

Performs a left-shift operation. Zeros are shifted in from the right.

```
$r0.d = $r0.x << [$r0.y|imm];
```

`shr $r0.d = $r0.x, $r0.y`

`shr $r0.d = $r0.x, imm`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0	0	0	0	d						x						y									s	
0	0	0	1	1	0	0	0	1	d						x						imm									s	

Performs a signed right-shift operation. That is, the sign bit of `$r0.x` is shifted in from the left.

```
$r0.d = $r0.x >> [$r0.y|imm];
```

`shru $r0.d = $r0.x, $r0.y`

`shru $r0.d = $r0.x, imm`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0	0	1	0	d						x						y									s	
0	0	0	1	1	0	0	1	1	d						x						imm									s	

Performs an unsigned right-shift operation. That is, zeros are shifted in from the left.

```
$r0.d = (unsigned int)$r0.x >> [$r0.y|imm];
```

3.7.3 ALU bitwise instructions

The ρ-VEX ALU supports a subset of bitwise operations in a single cycle.

and $r0.d = $r0.x, $r0.y

and $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	0	1	1	0	d						x						y									s	
0	1	1	0	0	0	1	1	1	d						x						imm									s	

Performs a bitwise AND operation.

```
$r0.d = $r0.x & [$r0.y|imm];
```

andc $r0.d = $r0.x, $r0.y

andc $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	0	0	d						x						y									s	
0	1	1	0	0	1	0	0	1	d						x						imm									s	

Performs a bitwise AND operation, with the first operand one's complemented.

```
$r0.d = ~$r0.x & [$r0.y|imm];
```

or $r0.d = $r0.x, $r0.y

or $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	0	0	1	0	d						x						y									s	
0	1	1	0	1	0	0	1	1	d						x						imm									s	

Performs a bitwise OR operation. Notice that OR instructions reduce to move or load immediate operations when x is set to 0.

```
$r0.d = $r0.x | [$r0.y|imm];
```

orc $r0.d = $r0.x, $r0.y

orc $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	0	1	0	0	d						x						y									s	
0	1	1	0	1	0	1	0	1	d						x						imm									s	

Performs a bitwise OR operation, with the first operand one's complemented. Notice that ORC instructions reduce to one's complement when y or imm is set to 0.

```
$r0.d = ~$r0.x | [$r0.y|imm];
```

xor $r0.d = $r0.x, $r0.y

xor $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	1	0	d						x						y									s	
0	0	0	1	1	1	1	1	1	d						x						imm									s	

Performs a bitwise XOR operation.

```
$r0.d = $r0.x ^ [$r0.y|imm];
```

3.7.4 ALU single-bit instructions

The ρ-VEX ALU supports several bitfield operations in a single cycle. Note that the bit selection logic follows the same rules as the shift amount in the barrel shifter. That is, only the least significant byte of the bit selection operand is used to select the bit, the rest is ignored.

sbit $r0.d = $r0.x, $r0.y

sbit $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	1	0	0	0	d						x						y									S	
0	0	1	0	1	1	0	0	1	d						x						imm									S	

Sets a given bit in a 32-bit integer.

```
$r0.d = $r0.x | (1 << [$r0.y|imm]);
```

sbitf $r0.d = $r0.x, $r0.y

sbitf $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	1	0	1	0	d						x						y									S	
0	0	1	0	1	1	0	1	1	d						x						imm									S	

Clears a given bit in a 32-bit integer.

```
$r0.d = $r0.x & ~(1 << [$r0.y|imm]);
```

tbit $r0.d = $r0.x, $r0.y

tbit $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	0	0	0	d						x						y									S	
0	1	0	1	1	1	0	0	1	d						x						imm									S	

Copies a given bit to an integer register.

```
$r0.d = ($r0.x & (1 << [$r0.y|imm])) != 0;
```

tbit $b0.bd = $r0.x, $r0.y

tbit $b0.bd = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	0	1	0				bd			x						y									S	
0	1	0	1	1	1	0	1	1				bd			x						imm									S	

Copies a given bit to a branch register.

```
$b0.bd = ($r0.x & (1 << [$r0.y|imm])) != 0;
```

tbitf $r0.d = $r0.x, $r0.y

tbitf $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	d						x						y									s	
0	1	0	1	1	1	1	0	1	d						x						imm									s	

Copies the complement of a given bit to an integer register.

```
$r0.d = ($r0.x & (1 << [$r0.y|imm])) == 0;
```

tbitf $b0.bd = $r0.x, $r0.y

tbitf $b0.bd = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	1	0				bd			x						y									s	
0	1	0	1	1	1	1	1	1				bd			x						imm									s	

Copies the complement of a given bit to a branch register.

```
$b0.bd = ($r0.x & (1 << [$r0.y|imm])) == 0;
```

3.7.5 ALU boolean instructions

As well as supporting many bitwise operations, the ρ-VEX ALU also supports some boolean operations in a single cycle. The boolean operations are defined in the same way as C boolean operations are defined. That is, the value 0 represents false, and any other value represents true.

andl $r0.d = $r0.x, $r0.y

andl $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	0	1	0	0	d						x						y									s	
0	1	0	1	1	0	1	0	1	d						x						imm									s	

Performs a boolean AND operation and stores the result in an integer register.

```
$r0.d = $r0.x && [$r0.y|imm];
```

andl $b0.bd = $r0.x, $r0.y

andl $b0.bd = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	0	1	1	0				bd			x						y									s	
0	1	0	1	1	0	1	1	1				bd			x						imm									s	

Performs a boolean AND operation and stores the result in a branch register.

```
$b0.bd = $r0.x && [$r0.y|imm];
```

orl $r0.d = $r0.x, $r0.y

orl $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	0	0	0	0	d						x						y									S	
0	1	0	1	1	0	0	0	1	d						x						imm									S	

Performs a boolean `OR` operation and stores the result in an integer register.

```
$r0.d = $r0.x || [$r0.y|imm];
```

orl $b0.bd = $r0.x, $r0.y

orl $b0.bd = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	0	0	1	0				bd			x						y									S	
0	1	0	1	1	0	0	1	1				bd			x						imm									S	

Performs a boolean `OR` operation and stores the result in a branch register.

```
$b0.bd = $r0.x || [$r0.y|imm];
```

nandl $r0.d = $r0.x, $r0.y

nandl $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	1	0	0	0	d						x						y									S	
0	1	0	1	0	1	0	0	1	d						x						imm									S	

Performs a boolean `NAND` operation and stores the result in an integer register.

```
$r0.d = !($r0.x && [$r0.y|imm]);
```

nandl $b0.bd = $r0.x, $r0.y

nandl $b0.bd = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	1	0	1	0				bd			x						y									S	
0	1	0	1	0	1	0	1	1				bd			x						imm									S	

Performs a boolean `NAND` operation and stores the result in a branch register.

```
$b0.bd = !($r0.x && [$r0.y|imm]);
```

norl $r0.d = $r0.x, $r0.y

norl $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	1	1	0	0	d						x						y									S	
0	1	0	1	0	1	1	0	1	d						x						imm									S	

Performs a boolean `NOR` operation and stores the result in an integer register.

```
$r0.d = !($r0.x || [$r0.y|imm]);
```

norl $b0.bd = $r0.x, $r0.y

norl $b0.bd = $r0.x, imm

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td colspan="3"></td><td colspan="3">bd</td><td colspan="6">x</td><td colspan="6">y</td><td colspan="3"></td><td>S</td><td></td></tr>
<tr><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td colspan="3"></td><td colspan="3">bd</td><td colspan="6">x</td><td colspan="9">imm</td><td>S</td><td></td></tr>
</table>

Performs a boolean NOR operation and stores the result in a branch register.

```
$b0.bd = !($r0.x || [$r0.y|imm]);
```

3.7.6 ALU compare instructions

The ρ-VEX ALU supports all 32-bit possible integer comparison operations in a single cycle. The immediate version of CMPNE that writes to a branch register is used to load an immediate branch register.

cmpeq $r0.d = $r0.x, $r0.y

cmpeq $r0.d = $r0.x, imm

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td colspan="6">d</td><td colspan="6">x</td><td colspan="6">y</td><td colspan="3"></td><td>S</td><td></td></tr>
<tr><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td colspan="6">d</td><td colspan="6">x</td><td colspan="9">imm</td><td>S</td><td></td></tr>
</table>

Determines whether the first operand is equal to the second operand and stores the result in an integer register.

```
$r0.d = $r0.x == [$r0.y|imm];
```

cmpeq $b0.bd = $r0.x, $r0.y

cmpeq $b0.bd = $r0.x, imm

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td colspan="3"></td><td colspan="3">bd</td><td colspan="6">x</td><td colspan="6">y</td><td colspan="3"></td><td>S</td><td></td></tr>
<tr><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td colspan="3"></td><td colspan="3">bd</td><td colspan="6">x</td><td colspan="9">imm</td><td>S</td><td></td></tr>
</table>

Determines whether the first operand is equal to the second operand and stores the result in a branch register.

```
$b0.bd = $r0.x == [$r0.y|imm];
```

cmpge $r0.d = $r0.x, $r0.y

cmpge $r0.d = $r0.x, imm

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td colspan="6">d</td><td colspan="6">x</td><td colspan="6">y</td><td colspan="3"></td><td>S</td><td></td></tr>
<tr><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td colspan="6">d</td><td colspan="6">x</td><td colspan="9">imm</td><td>S</td><td></td></tr>
</table>

Determines whether the first operand is greater than or equal to the second operand and stores the result in an integer register.

```
$r0.d = $r0.x >= [$r0.y|imm];
```

`cmpge $b0.bd = $r0.x, $r0.y`

`cmpge $b0.bd = $r0.x, imm`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	1	1	0				bd			x						y									s	
0	1	0	0	0	0	1	1	1				bd			x						imm									s	

Determines whether the first operand is greater than or equal to the second operand and stores the result in a branch register.

```
$b0.bd = $r0.x >= [$r0.y|imm];
```

`cmpgeu $r0.d = $r0.x, $r0.y`

`cmpgeu $r0.d = $r0.x, imm`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	0	d						x						y									s	
0	1	0	0	0	1	0	0	1	d						x						imm									s	

Determines whether the first operand is greater than or equal to the second operand in unsigned arithmetic and stores the result in an integer register.

```
$r0.d = (unsigned int)$r0.x >= (unsigned int)[$r0.y|imm];
```

`cmpgeu $b0.bd = $r0.x, $r0.y`

`cmpgeu $b0.bd = $r0.x, imm`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	1	0				bd			x						y									s	
0	1	0	0	0	1	0	1	1				bd			x						imm									s	

Determines whether the first operand is greater than or equal to the second operand in unsigned arithmetic and stores the result in a branch register.

```
$b0.bd = (unsigned int)$r0.x >= (unsigned int)[$r0.y|imm];
```

`cmpgt $r0.d = $r0.x, $r0.y`

`cmpgt $r0.d = $r0.x, imm`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	1	0	0	d						x						y									s	
0	1	0	0	0	1	1	0	1	d						x						imm									s	

Determines whether the first operand is greater than the second operand and stores the result in an integer register.

```
$r0.d = $r0.x > [$r0.y|imm];
```

`cmpgt $b0.bd = $r0.x, $r0.y`

`cmpgt $b0.bd = $r0.x, imm`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	1	1	0				bd			x						y									s	
0	1	0	0	0	1	1	1	1				bd			x						imm									s	

Determines whether the first operand is greater than the second operand and stores the result in a branch register.

```
$b0.bd = $r0.x > [$r0.y|imm];
```

`cmpgtu $r0.d = $r0.x, $r0.y`

`cmpgtu $r0.d = $r0.x, imm`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	0	d						x						y									S	
0	1	0	0	1	0	0	0	1	d						x						imm									S	

Determines whether the first operand is greater than the second operand in unsigned arithmetic and stores the result in an integer register.

```
$r0.d = (unsigned int)$r0.x > (unsigned int)[$r0.y|imm];
```

`cmpgtu $b0.bd = $r0.x, $r0.y`

`cmpgtu $b0.bd = $r0.x, imm`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	1	0				bd			x						y									S	
0	1	0	0	1	0	0	1	1				bd			x						imm									S	

Determines whether the first operand is greater than the second operand in unsigned arithmetic and stores the result in a branch register.

```
$b0.bd = (unsigned int)$r0.x > (unsigned int)[$r0.y|imm];
```

`cmple $r0.d = $r0.x, $r0.y`

`cmple $r0.d = $r0.x, imm`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	1	0	0	d						x						y									S	
0	1	0	0	1	0	1	0	1	d						x						imm									S	

Determines whether the first operand is less than or equal to the second operand and stores the result in an integer register.

```
$r0.d = $r0.x <= [$r0.y|imm];
```

`cmple $b0.bd = $r0.x, $r0.y`

`cmple $b0.bd = $r0.x, imm`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	1	1	0				bd			x						y									S	
0	1	0	0	1	0	1	1	1				bd			x						imm									S	

Determines whether the first operand is less than or equal to the second operand and stores the result in a branch register.

```
$b0.bd = $r0.x <= [$r0.y|imm];
```

cmpleu $r0.d = $r0.x, $r0.y

cmpleu $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	0	0	0	d						x						y									s	
0	1	0	0	1	1	0	0	1	d						x						imm									s	

Determines whether the first operand is less than or equal to the second operand in unsigned arithmetic and stores the result in an integer register.

```
$r0.d = (unsigned int)$r0.x <= (unsigned int)[$r0.y|imm];
```

cmpleu $b0.bd = $r0.x, $r0.y

cmpleu $b0.bd = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	0	1	0				bd			x						y									s	
0	1	0	0	1	1	0	1	1				bd			x						imm									s	

Determines whether the first operand is less than or equal to the second operand in unsigned arithmetic and stores the result in a branch register.

```
$b0.bd = (unsigned int)$r0.x <= (unsigned int)[$r0.y|imm];
```

cmplt $r0.d = $r0.x, $r0.y

cmplt $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	d						x						y									s	
0	1	0	0	1	1	1	0	1	d						x						imm									s	

Determines whether the first operand is less than the second operand and stores the result in an integer register.

```
$r0.d = $r0.x <= [$r0.y|imm];
```

cmplt $b0.bd = $r0.x, $r0.y

cmplt $b0.bd = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	1	0				bd			x						y									s	
0	1	0	0	1	1	1	1	1				bd			x						imm									s	

Determines whether the first operand is less than the second operand and stores the result in a branch register.

```
$b0.bd = $r0.x <= [$r0.y|imm];
```

cmpltu $r0.d = $r0.x, $r0.y

cmpltu $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	0	0	0	0	d						x						y									s	
0	1	0	1	0	0	0	0	1	d						x						imm									s	

Determines whether the first operand is less than the second operand in unsigned arithmetic and stores the result in an integer register.

```
$r0.d = (unsigned int)$r0.x <= (unsigned int)[$r0.y|imm];
```

`cmpltu $b0.bd = $r0.x, $r0.y`

`cmpltu $b0.bd = $r0.x, imm`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	0	0	1	0				bd			x						y									s	
0	1	0	1	0	0	0	1	1				bd			x						imm									s	

Determines whether the first operand is less than the second operand in unsigned arithmetic and stores the result in a branch register.

```
$b0.bd = (unsigned int)$r0.x <= (unsigned int)[$r0.y|imm];
```

`cmpne $r0.d = $r0.x, $r0.y`

`cmpne $r0.d = $r0.x, imm`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	0	1	0	0	d						x						y									s	
0	1	0	1	0	0	1	0	1	d						x						imm									s	

Determines whether the first operand is not equal to the second operand and stores the result in an integer register.

```
$r0.d = $r0.x != [$r0.y|imm];
```

`cmpne $b0.bd = $r0.x, $r0.y`

`cmpne $b0.bd = $r0.x, imm`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	0	1	1	0				bd			x						y									s	
0	1	0	1	0	0	1	1	1				bd			x						imm									s	

Determines whether the first operand is not equal to the second operand and stores the result in a branch register.

Notice that the immediate version of `CMPNE` reduces to a load immediate operation for branch registers when `x` is zero.

```
$b0.bd = $r0.x != [$r0.y|imm];
```

3.7.7 ALU selection instructions

The ρ-VEX ALU has single-cycle instructions for conditional moves and computation of the minimum and maximum of two integer values.

`slct $r0.d = $b0.bs, $r0.x, $r0.y`

`slct $r0.d = $b0.bs, $r0.x, imm`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	bs			0	d						x						y									S	
0	0	1	1	1	bs			1	d						x						imm									S	

Conditional move.

```
$r0.d = $b0.bs ? $r0.x : [$r0.y|imm];
```

`slctf $r0.d = $b0.bs, $r0.x, $r0.y`

`slctf $r0.d = $b0.bs, $r0.x, imm`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	bs			0	d						x						y									S	
0	0	1	1	0	bs			1	d						x						imm									S	

Conditional move, with operands swapped with respect to `SLCT`.

Notice that the immediate version of `SLCTF` reduces to a move from a branch register to an integer register when `x` is 0 and `y` is 1.

```
$r0.d = $b0.bs ? [$r0.y|imm] : $r0.x;
```

`max $r0.d = $r0.x, $r0.y`

`max $r0.d = $r0.x, imm`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	0	d						x						y									S	
0	1	1	0	0	1	0	1	1	d						x						imm									S	

Computes maximum of the input operands using signed arithmetic.

```
$r0.d = ($r0.x >= [$r0.y|imm]) : $r0.x ? [$r0.y|imm];
```

`maxu $r0.d = $r0.x, $r0.y`

`maxu $r0.d = $r0.x, imm`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	1	0	0	d						x						y									S	
0	1	1	0	0	1	1	0	1	d						x						imm									S	

Computes maximum of the input operands using unsigned arithmetic.

```
$r0.d = ((unsigned int)$r0.x >= (unsigned int)[$r0.y|imm]) : $r0.x ? [$r0.y|imm];
```

`min $r0.d = $r0.x, $r0.y`

`min $r0.d = $r0.x, imm`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	1	1	0	d						x						y									S	
0	1	1	0	0	1	1	1	1	d						x						imm									S	

Computes minimum of the input operands using signed arithmetic.

```
$r0.d = ($r0.x <= [$r0.y|imm]) : $r0.x ? [$r0.y|imm];
```

minu $r0.d = $r0.x, $r0.y

minu $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	0	0	0	0	d						x						y									s	
0	1	1	0	1	0	0	0	1	d						x						imm									s	

Computes minimum of the input operands using unsigned arithmetic.

```
$r0.d = ((unsigned int)$r0.x <= (unsigned int)[$r0.y|imm]) : $r0.x ? [$r0.y|imm];
```

3.7.8 ALU type conversion instructions

The ρ-VEX ALU is capable of supporting typecasts from 32-bit integers to 16-bit and 8-bit integers in a single cycle.

sxtb $r0.d = $r0.x

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0	1	1	0	d						x															s	

Performs sign extension for an 8-bit value.

```
$r0.d = (char)$r0.x;
```

sxth $r0.d = $r0.x

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	0	0	0	d						x															s	

Performs sign extension for a 16-bit value.

```
$r0.d = (short)$r0.x;
```

zxtb $r0.d = $r0.x

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	0	1	0	d						x															s	

Performs zero extension for an 8-bit value.

```
$r0.d = (unsigned char)$r0.x;
```

zxth $r0.d = $r0.x

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	0	d						x															s	

Performs zero extension for a 16-bit value.

```
$r0.d = (unsigned short)$r0.x;
```

3.7.9 ALU miscellaneous instructions

`nop`

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td colspan="22"></td><td>S</td><td></td></tr>
</table>

Performs no operation.

`clz $r0.d = $r0.x`

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td colspan="6">d</td><td colspan="6">x</td><td colspan="9"></td><td>S</td><td></td></tr>
</table>

This operations counts the number of leading zeros in the operand. That is, the value 0x80000000 returns 0 and the value 0 returns 32.

```
unsigned int in = $r0.x;
int out = 32;
while (in) {
  in >>= 1;
  out--;
}
$r0.d = out;
```

`movtl $l0.0 = $r0.y`

`movtl $l0.0 = imm`

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td colspan="12"></td><td colspan="6">y</td><td colspan="3"></td><td>S</td><td></td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td colspan="12"></td><td colspan="9">imm</td><td>S</td><td></td></tr>
</table>

Copies a general purpose register or immediate to the link register.

```
$l0.0 = [$r0.y|imm];
```

`movfl $r0.d = $l0.0`

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td colspan="6">d</td><td colspan="15"></td><td>S</td><td></td></tr>
</table>

Copies the link register to a general purpose register.

```
$r0.d = $l0.0;
```

`trap $r0.x, $r0.y`

`trap $r0.x, imm`

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td colspan="6"></td><td colspan="6">x</td><td colspan="6">y</td><td colspan="3"></td><td>S</td><td></td></tr>
<tr><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td colspan="6"></td><td colspan="6">x</td><td colspan="9">imm</td><td>S</td><td></td></tr>
</table>

Software trap. The first parameter is the trap argument, while the second parameter is the trap cause byte.

3.7.10 Multiply instructions

ρ-VEX pipelanes may be design-time configured to contain a multiplication unit. This unit supports 16x16 and 16x32 multiplications.

In the default pipeline configuration, these instructions are pipelined by two cycles. That is, the result of a multiply instruction is not available yet in the subsequent instruction.

`mpyll $r0.d = $r0.x, $r0.y`

`mpyll $r0.d = $r0.x, imm`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	d						x						y									s	
0	0	0	0	0	0	0	0	1	d						x						imm									s	

Signed 16x16-bit to 32-bit multiplication.

```c
$r0.d = (short)$r0.x * (short)[$r0.y|imm];
```

`mpyllu $r0.d = $r0.x, $r0.y`

`mpyllu $r0.d = $r0.x, imm`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1	0	d						x						y									s	
0	0	0	0	0	0	0	1	1	d						x						imm									s	

Unsigned 16x16-bit to 32-bit multiplication.

```c
$r0.d = (unsigned short)$r0.x * (unsigned short)[$r0.y|imm];
```

`mpylh $r0.d = $r0.x, $r0.y`

`mpylh $r0.d = $r0.x, imm`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	0	0	d						x						y									s	
0	0	0	0	0	0	1	0	1	d						x						imm									s	

Signed 16x16-bit to 32-bit multiplication, using the high halfword of `[$r0.y|imm]`.

```c
$r0.d = (short)$r0.x * (short)([$r0.y|imm] >> 16);
```

`mpylhu $r0.d = $r0.x, $r0.y`

`mpylhu $r0.d = $r0.x, imm`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	1	0	d						x						y									s	
0	0	0	0	0	0	1	1	1	d						x						imm									s	

Unsigned 16x16-bit to 32-bit multiplication, using the high halfword of `[$r0.y|imm]`.

```c
$r0.d = (unsigned short)$r0.x * (unsigned short)([$r0.y|imm] >> 16);
```

mpyhh $r0.d = $r0.x, $r0.y

mpyhh $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	0	d						x						y									S	
0	0	0	0	0	1	0	0	1	d						x						imm									S	

Signed 16x16-bit to 32-bit multiplication, using the high halfword of both operands.

```
$r0.d = (short)($r0.x >> 16) * (short)([$r0.y|imm] >> 16);
```

mpyhhu $r0.d = $r0.x, $r0.y

mpyhhu $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	0	d						x						y									S	
0	0	0	0	0	1	0	1	1	d						x						imm									S	

Unsigned 16x16-bit to 32-bit multiplication, using the high halfword of both operands.

```
$r0.d = (unsigned short)($r0.x >> 16) * (unsigned short)([$r0.y|imm] >> 16);
```

mpyl $r0.d = $r0.x, $r0.y

mpyl $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	d						x						y									S	
0	0	0	0	0	1	1	0	1	d						x						imm									S	

Signed 16x32-bit to 32-bit multiplication. `$r0.x` is the 32-bit operand, `[$r0.y|imm]` is the 16-bit operand. The upper 16 bits of the multiplication result are discarded.

```
$r0.d = (int)$r0.x * (short)[$r0.y|imm];
```

mpylu $r0.d = $r0.x, $r0.y

mpylu $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	1	0	d						x						y									S	
0	0	0	0	0	1	1	1	1	d						x						imm									S	

Unsigned 16x32-bit to 32-bit multiplication. `$r0.x` is the 32-bit operand, `[$r0.y|imm]` is the 16-bit operand. The upper 16 bits of the multiplication result are discarded.

This operation may be used as a primitive for 32x32-bit to 32-bit multiplication, computing the partial product of `$r0.x` and the lower half of `[$r0.y|imm]`. `MPYHS` is then used to compute the other partial product. A final `ADD` instruction combines the partial products.

```
$r0.d = (unsigned int)$r0.x * (unsigned short)[$r0.y|imm];
```

mpyh $r0.d = $r0.x, $r0.y

mpyh $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	d						x						y									s	
0	0	0	0	1	0	0	0	1	d						x						imm									s	

Signed 16x32-bit to 32-bit multiplication. `$r0.x` is the 32-bit operand, the upper halfword of `[$r0.y|imm]` is the 16-bit operand. The upper 16 bits of the multiplication result are discarded.

```
$r0.d = (int)$r0.x * (short)([$r0.y|imm] >> 16);
```

mpyhu $r0.d = $r0.x, $r0.y

mpyhu $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	1	0	d						x						y									s	
0	0	0	0	1	0	0	1	1	d						x						imm									s	

Unsigned 16x32-bit to 32-bit multiplication. `$r0.x` is the 32-bit operand, the upper halfword of `[$r0.y|imm]` is the 16-bit operand. The upper 16 bits of the multiplication result are discarded.

```
$r0.d = (unsigned int)$r0.x * (unsigned short)([$r0.y|imm] >> 16);
```

mpyhs $r0.d = $r0.x, $r0.y

mpyhs $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	0	0	d						x						y									s	
0	0	0	0	1	0	1	0	1	d						x						imm									s	

Signed 16x32-bit to 32-bit multiplication. `$r0.x` is the 32-bit operand, the upper halfword of `[$r0.y|imm]` is the 16-bit operand. The result is shifted left by 16, discarding the upper 32 bits of the 48-bit result.

This operation may be used as a primitive for 32x32-bit to 32-bit multiplication, computing the partial product of `$r0.x` and the upper half of `[$r0.y|imm]`. `MPYLU` is then used to compute the other partial product. A final `ADD` instruction combines the partial products.

```
$r0.d = ((int)$r0.x * (short)[$r0.y|imm]) << 16;
```

mpylhus $r0.d = $r0.x, $r0.y

mpylhus $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	0	0	1	0	0	d						x						y									s	
1	0	0	1	0	0	1	0	1	d						x						imm									s	

Mixed 16x32-bit to 32-bit multiplication. `$r0.x` is the 32-bit operand, interpreted as a signed number. `[$r0.y|imm]` is the 16-bit operand, interpreted as an unsigned number. The 48-bit result is shifted right by 32 bits.

Together with MPYHS, MPYLU, MPYHHS, ADD and ADDCG, a full signed 32x32-bit to 64-bit multiplication may be realized as follows.

```
mpylu   low1  = op1, op2
mpyhs   low2  = op1, op2
;;
mpylhus high1 = op1, op2
mpyhhs  high2 = op1, op2
;;
addcg   low, carry = <zero>, low1, low2
;;
addcg   high, carry = carry, high1, high2
;;
```

The first two multiply instructions and the first ADDCG compute the low word of the multiplication and carry bit for the high word. The remaining instructions do the same for the high part of the result.

```
$r0.d = ((long long)$r0.x * (short)[$r0.y|imm]) >> 32;
```

mpyhhs $r0.d = $r0.x, $r0.y

mpyhhs $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	0	0	1	1	0	d						x						y									s	
1	0	0	1	0	0	1	1	1	d						x						imm									s	

Signed 16x32-bit to 32-bit multiplication. $r0.x is the 32-bit operand, the upper halfword of [$r0.y|imm] is the 16-bit operand. The 48-bit result is shifted right by 16 bits.

This syllable is used in conjunction with other multiplication syllables for a 32x32-bit to 64-bit signed multiplication. Refer to MPYLHUS for more information.

```
$r0.d = ((long long)$r0.x * (short)[$r0.y|imm]) >> 32;
```

3.7.11 Memory instructions

Some ρ-VEX pipelanes have a memory unit. The memory unit supports byte, halfword and word operations. Sign or zero extension is part of the byte and halfword load instructions.

The addressing mode is always register + immediate. Note that attempts to read misaligned memory locations will fail with a TRAP_MISALIGNED_ACCESS trap.

In the default pipeline configuration, these instructions are pipelined by two cycles. That is, the result of a memory load instruction is not available yet in the subsequent instruction. However, the current cache and core guarantee that a memory write to address x immediately followed by a memory read from address x returns the newly written value.

ldw $r0.d = imm[$r0.x]

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	0	0	1	d						x						imm									s	

Loads a 32-bit word from memory.

```
$r0.d = *(int*)($r0.x + imm);
```

`ldh $r0.d = imm[$r0.x]`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	0	1	1	d						x						imm									s	

Loads a 16-bit halfword from memory and sign-extends it.

```
$r0.d = *(short*)($r0.x + imm);
```

`ldhu $r0.d = imm[$r0.x]`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	1	0	1	d						x						imm									s	

Loads a 16-bit halfword from memory and zero-extends it.

```
$r0.d = *(unsigned short*)($r0.x + imm);
```

`ldb $r0.d = imm[$r0.x]`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	1	1	1	d						x						imm									s	

Loads a byte from memory and sign-extends it.

```
$r0.d = *(char*)($r0.x + imm);
```

`ldbu $r0.d = imm[$r0.x]`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	1	0	0	1	d						x						imm									s	

Loads a byte from memory and zero-extends it.

```
$r0.d = *(unsigned char*)($r0.x + imm);
```

`ldw $l0.0 = imm[$r0.x]`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	1	1							x						imm									s	

Loads a word from memory. The result is written to the link register.

```
$l0.0 = *(int*)($r0.x + imm);
```

`ldbr imm[$r0.x]`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	1	1	0	1							x						imm									s	

Loads a byte from memory. The result is written to the entire branch register file at once. This is intended to improve context switching performance somewhat.

```
char tmp = *(char*)($r0.x + imm);
$b0.0 = tmp & 1;
$b0.1 = tmp & 2;
$b0.2 = tmp & 4;
$b0.3 = tmp & 8;
$b0.4 = tmp & 16;
$b0.5 = tmp & 32;
$b0.6 = tmp & 64;
$b0.7 = tmp & 128;
```

stw imm[$r0.x] = $r0.d

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	1	0	1	1	d						x						imm									S	

Stores a 32-bit word into memory.

```
*(int*)($r0.x + imm) = $r0.d;
```

sth imm[$r0.x] = $r0.d

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	1	1	0	1	d						x						imm									S	

Stores a 16-bit halfword into memory.

```
*(short*)($r0.x + imm) = $r0.d;
```

stb imm[$r0.x] = $r0.d

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	1	1	1	1	d						x						imm									S	

Stores a byte into memory.

```
*(char*)($r0.x + imm) = $r0.d;
```

stw imm[$r0.x] = $l0.0

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	0	1							x						imm									S	

Store word in memory, from link register.

```
*(int*)($r0.x + imm) = $l0.0;
```

stbr imm[$r0.x]

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	1	1	1	1							x						imm									S	

Store byte in memory, from branch register file.

```
char tmp = $b0.0;
tmp |= $b0.1 << 1
tmp |= $b0.2 << 2
tmp |= $b0.3 << 3
tmp |= $b0.4 << 4
tmp |= $b0.5 << 5
tmp |= $b0.6 << 6
tmp |= $b0.7 << 7
*(char*)($r0.x + imm) = tmp
```

3.7.12 Branch instructions

The highest-indexed pipelane in every ρ-VEX system (i.e., the pipelane that the last syllable in a bundle maps to) contains a branch unit. This unit supports the flow control operations outlined below.

Branch offsets are signed immediates relative to the next program counter (`PC+1`). Because there are certain alignment requirements to program counters, the lower two or three bits of the offset are not actually included in the bitfield. Whether this value is two or three depends on the value of the `BRANCH_OFFS_SHIFT` constant defined in `core_intIface_pkg.vhd`; it is three by default. It must be set to two to support branching to the start of any bundle when stop bits are fully enabled. This must then also be updated in the assembler.

Note that branch offsets and the stack adjust immediate are not eligible for long immediate instructions.

`goto offs`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	0	0	0	offs																						S	

Branches to `PC+1` + `offs` unconditionally.

```
PCP1 += offs;
```

`igoto $l0.0`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	0	0	1																							S	

Branches to the address in `$l0.0` unconditionally. This is used for branches to code that cannot be reached using the branch offset immediate.

```
PCP1 = $l0.0;
```

`call $l0.0 = offs`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	0	1	0	offs																						S	

Branches to `PC+1` + `offs` unconditionally, while storing `PC+1` in the link register. This is used for function calls.

```
$l0.0 = PCP1;
PCP1 += offs;
```

icall $l0.0 = $l0.0

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	0	1	1																							S	

Branches to the address in `$l0.0` unconditionally, while storing `PC+1` in the link register. In other words, it essentially swaps `PC+1` and `$l0.0`. This is used for dynamic function calls or calls to functions that cannot be reached using the branch offset immediate method.

```
unsigned int tmp = $l0.0;
$l0.0 = PCP1;
PCP1 = tmp;
```

br $b0.bs, offs

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	0	offs																			bs			S	

Branches to `PC+1` + `offs` only if `$b0.bs` is true. This instruction performs no operation if `$b0.bs` is false.

```
PCP1 += $b0.bs ? offs : 0;
```

brf $b0.bs, offs

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	offs																			bs			S	

Branches to `PC+1` + `offs` only if `$b0.bs` is false. This instruction performs no operation if `$b0.bs` is true.

```
PCP1 += $b0.bs ? 0 : offs;
```

return $r0.1 = $r0.1, stackadj, $l0.0

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	1	0	stackadj																						S	

Returns from a function by branching to `$l0.0` unconditionally, while adding `stackadj` to `$r0.1`. `stackadj` is interpreted as a signed immediate. This allows final stack pointer adjustment and returning to be done with a single syllable.

Notice that this instruction is identical to `IGOTO`, except for the fact that `IGOTO` does not access `$r0.1`.

```
$r0.1 += stackadj;
PCP1 = $l0.0;
```

`rfi $r0.1 = $r0.1, stackadj`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	1	1	stackadj																						S	

Returns from a trap service routine by branching to `CR_TP` unconditionally and restoring `CR_SCCR` to `CR_CCR`, while adding `stackadj` to `$r0.1`. `stackadj` is interpreted as a signed immediate. This allows final stack pointer adjustment and returning to be done with a single syllable.

```
$r0.1 += stackadj;
CR_CCR = CR_SCCR;
PCP1 = CR_TP;
```

`stop`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	0	0	0																							S	

Causes a `TRAP_STOP` trap to occur during execution of the next instruction. The `TRAP_STOP` trap will cause the B flag in `CR_DCR` to be set, which will stop execution. Thus, the processor will be stopped after the bundle in which the `STOP` instruction resides is executed.

3.7.13 Long immediate instructions

`limmh tgt, imm`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	tgt			imm																							S	

This special instruction forwards `imm` to lane `tgt`. Actually, only the least significant bit of `tgt` is used by the processor, to distinguish between the two possible long immediate forwarding paths. Refer to Section 3.4.5 for more information.

4 Control registers

The ρ-VEX processor has two control register files. These are the global control register file (`gbreg`) and the context control register file (`cxreg`).

The `gbreg` file contains mostly status information, such as a general purpose cycle counter, the current configuration vector and design-time configuration information. While the debug bus has read/write access to `gbreg`, the core can only read from it.

For more information, refer to Section 3.2.4.

4.1 Global control registers

The following table lists the global control registers of the ρ-VEX processor. The offsets listed are with respect to the control register base address. If you are viewing this manual digitally, you can click the register mnemonics on the right to jump to their documentation.

<table>
<tr><td>Offset</td><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td><td></td></tr>
<tr><td>0x000</td><td>R</td><td colspan="17"></td><td>E</td><td>B</td><td colspan="4">RID</td><td colspan="8"></td><td>CR_GSR</td></tr>
<tr><td>0x004</td><td colspan="32">BCRR</td><td>CR_BCRR</td></tr>
<tr><td>0x008</td><td colspan="32">CC</td><td>CR_CC</td></tr>
<tr><td>0x00C</td><td colspan="32">AF</td><td>CR_AFF</td></tr>
<tr><td>0x010</td><td colspan="32">CNT</td><td>CR_CNT</td></tr>
<tr><td>0x014</td><td colspan="24">CNTH</td><td colspan="8">CNT</td><td>CR_CNTH</td></tr>
<tr><td>...</td><td colspan="32"><i>Unused</i></td><td></td></tr>
<tr><td>0x0A0</td><td colspan="16">BORROW15</td><td colspan="16">BORROW14</td><td>CR_LIMC7</td></tr>
<tr><td>0x0A4</td><td colspan="16">BORROW13</td><td colspan="16">BORROW12</td><td>CR_LIMC6</td></tr>
<tr><td>0x0A8</td><td colspan="16">BORROW11</td><td colspan="16">BORROW10</td><td>CR_LIMC5</td></tr>
<tr><td>0x0AC</td><td colspan="16">BORROW9</td><td colspan="16">BORROW8</td><td>CR_LIMC4</td></tr>
<tr><td>0x0B0</td><td colspan="16">BORROW7</td><td colspan="16">BORROW6</td><td>CR_LIMC3</td></tr>
<tr><td>0x0B4</td><td colspan="16">BORROW5</td><td colspan="16">BORROW4</td><td>CR_LIMC2</td></tr>
<tr><td>0x0B8</td><td colspan="16">BORROW3</td><td colspan="16">BORROW2</td><td>CR_LIMC1</td></tr>
<tr><td>0x0BC</td><td colspan="16">BORROW1</td><td colspan="16">BORROW0</td><td>CR_LIMC0</td></tr>
<tr><td>0x0C0</td><td colspan="8">SYL15CAP</td><td colspan="8">SYL14CAP</td><td colspan="8">SYL13CAP</td><td colspan="8">SYL12CAP</td><td>CR_SIC3</td></tr>
<tr><td>0x0C4</td><td colspan="8">SYL11CAP</td><td colspan="8">SYL10CAP</td><td colspan="8">SYL9CAP</td><td colspan="8">SYL8CAP</td><td>CR_SIC2</td></tr>
<tr><td>0x0C8</td><td colspan="8">SYL7CAP</td><td colspan="8">SYL6CAP</td><td colspan="8">SYL5CAP</td><td colspan="8">SYL4CAP</td><td>CR_SIC1</td></tr>
<tr><td>0x0CC</td><td colspan="8">SYL3CAP</td><td colspan="8">SYL2CAP</td><td colspan="8">SYL1CAP</td><td colspan="8">SYL0CAP</td><td>CR_SIC0</td></tr>
<tr><td>0x0D0</td><td colspan="32"></td><td>CR_GPS1</td></tr>
<tr><td>0x0D4</td><td colspan="4"></td><td colspan="4">MEMAR</td><td colspan="4">MEMDC</td><td colspan="4">MEMDR</td><td colspan="4">MULC</td><td colspan="4">MULR</td><td colspan="4">ALUC</td><td colspan="4">ALUR</td><td>CR_GPS0</td></tr>
<tr><td>0x0D8</td><td colspan="32"></td><td>CR_SPS1</td></tr>
<tr><td>0x0DC</td><td colspan="4">MEMMC</td><td colspan="4">MEMMR</td><td colspan="4">MEMDC</td><td colspan="4">MEMDR</td><td colspan="4">BRC</td><td colspan="4">BRR</td><td colspan="4">ALUC</td><td colspan="4">ALUR</td><td>CR_SPS0</td></tr>
<tr><td>0x0E0</td><td colspan="32"></td><td>CR_EXT2</td></tr>
<tr><td>0x0E4</td><td colspan="32"></td><td>CR_EXT1</td></tr>
<tr><td>0x0E8</td><td colspan="4"></td><td>T</td><td colspan="3">BRK</td><td colspan="4"></td><td>C</td><td colspan="3">P</td><td colspan="13"></td><td>O</td><td>L</td><td>F</td><td>CR_EXT0</td></tr>
<tr><td>0x0EC</td><td colspan="16"></td><td colspan="4">BA</td><td colspan="4">NC</td><td colspan="4">NG</td><td colspan="4">NL</td><td>CR_DCFG</td></tr>
<tr><td>0x0F0</td><td colspan="8">VER</td><td colspan="8">CTAG0</td><td colspan="8">CTAG1</td><td colspan="8">CTAG2</td><td>CR_CVER1</td></tr>
<tr><td>0x0F4</td><td colspan="8">CTAG3</td><td colspan="8">CTAG4</td><td colspan="8">CTAG5</td><td colspan="8">CTAG6</td><td>CR_CVER0</td></tr>
</table>

Offset	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0	
0x0F8	COID	PTAG0	PTAG1	PTAG2	CR_PVER1
0x0FC	PTAG3	PTAG4	PTAG5	PTAG6	CR_PVER0

4.1.1 CR_GSR - Global status register

Offset	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0x000	R																		E	B	RID												CR_GSR
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
Core																																	
Debug	✓																																

This register contains miscellaneous status information.

R flag, bit 31

Reset flag. The entire ρ-VEX processor will be reset when the debug bus writes a one to this flag. Writing a zero has no effect.

E flag, bit 13

Reconfiguration error flag. This flag is set by hardware when an invalid configuration was requested. It is cleared once a valid configuration is requested.

B flag, bit 12

Reconfiguration busy flag. While high, reconfiguration requests are ignored.

RID field, bits 11..8

Reconfiguration requester ID. When a configuration is requested, this field is set to the context ID of the context that requested the configuration, or to 0xF if the request was from the debug bus. This may be used by the reconfiguration sources to see if they have won arbitration. Refer to Section 6.2 for more information.

4.1.2 CR_BCRR - Bus reconfiguration request register

Offset	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0x004	BCRR																																CR_BCRR
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
Core																																	
Debug	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	

This register may be written to by the debug bus only. When it is written, a reconfiguration is requested. Refer to Sections 6.1 and 6.2 for more information.

4.1.3 `CR_CC` - Current configuration register

Offset	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0x008	CC																																CR_CC
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
Core																																	
Debug																																	

This register is hardwired to the current configuration vector. Refer to Section 6.1 for more information.

4.1.4 `CR_AFF` - Cache affinity register

Offset	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0x00C	AF																																CR_AFF
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
Core																																	
Debug																																	

This register stores the cache block index (akin to a lane group) that most recently serviced an instruction fetch for a given context. This may be used for achieving the maximum possible instruction cache locality when reconfiguring.

Each nibble represents a lane group. The nibble value is the context index.

4.1.5 `CR_CNT` - Cycle counter register

Offset	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0x010	CNT																																CR_CNT
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
Core																																	
Debug																																	

Cycle counter. This register is simply always incremented by one in hardware. Simply overflows when it reaches 0xFFFFFFFF. Its intended use is to monitor real time. As an indication, this register overflows approximately every 85 seconds at 50 MHz.

4.1.6 `CR_CNTH` - Cycle counter register high

Offset	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0x014	CNTH																								CNT								CR_CNTH
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
Core																																	
Debug																																	

This register extends the `CR_CNT` register by 24 bits. The low byte is equal to the high byte of `CR_CNT`, similar to the performance counters, which allows the same algorithm to be used in order to read the value. Refer to Section 4.3 for more information. Note however, that unlike the other performance counters, this register always exists, regardless of the design-time configured performance counter width.

4.1.7 `CR_LIMCn` - Long immediate capability register n

Offset	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0	
`0x0A0`	BORROW15		BORROW14		`CR_LIMC7`
`0x0A4`	BORROW13		BORROW12		`CR_LIMC6`
`0x0A8`	BORROW11		BORROW10		`CR_LIMC5`
`0x0AC`	BORROW9		BORROW8		`CR_LIMC4`
`0x0B0`	BORROW7		BORROW6		`CR_LIMC3`
`0x0B4`	BORROW5		BORROW4		`CR_LIMC2`
`0x0B8`	BORROW3		BORROW2		`CR_LIMC1`
`0x0BC`	BORROW1		BORROW0		`CR_LIMC0`
Reset	* * * * * * * *	* * * * * * * *	* * * * * * * *	* * * * * * * *	
Core					
Debug					

This group of hardwired values represent the supported `LIMMH` forwarding routes.

BORROW$2n+1$ field, bits 31..16, a.k.a. `CR_BORROWi`

BORROW$2n$ field, bits 15..0, a.k.a. `CR_BORROWi`

Each bit in these fields represents a possible `LIMMH` forwarding route. The bit index within the field specifies the source syllable index, i.e. the `LIMMH` syllable; $i = (2n, 2n+1)$ is the index of the syllable that uses the immediate.

As an example, if bit 2 in `BORROW4` (`CR_LIMC2`) is set, it means that the third syllable in a bundle (index 2) can be a `LIMMH` instruction that forwards to the fifth syllable in a bundle (index 4).

For the purpose of generic binaries, the configuration is repeated beyond the number of physically available lanes.

4.1.8 `CR_SICn` - Syllable index capability register n

Offset	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0	
`0x0C0`	SYL15CAP	SYL14CAP	SYL13CAP	SYL12CAP	`CR_SIC3`
`0x0C4`	SYL11CAP	SYL10CAP	SYL9CAP	SYL8CAP	`CR_SIC2`
`0x0C8`	SYL7CAP	SYL6CAP	SYL5CAP	SYL4CAP	`CR_SIC1`
`0x0CC`	SYL3CAP	SYL2CAP	SYL1CAP	SYL0CAP	`CR_SIC0`
Reset	0 0 0 0 * * * 1	0 0 0 0 * * * 1	0 0 0 0 * * * 1	0 0 0 0 * * * 1	
Core					
Debug					

This group of hardwired values represent the capabilities of each syllable within a bundle.

SYL$4n+3$CAP field, bits 31..24, a.k.a. `CR_SYLiCAP`

SYL$4n+2$CAP field, bits 23..16, a.k.a. `CR_SYLiCAP`

SYL$4n+1$CAP field, bits 15..8, a.k.a. `CR_SYLiCAP`

SYL$4n$CAP field, bits 7..0, a.k.a. `CR_SYLiCAP`

Each bit within the field represents a functional unit or resource that is available to syllable index i within a bundle. The following encoding is used.

Bit index	*Function*
0	Always set, indicated that ALU class syllables are supported.
1	If set, multiplier class syllables are supported.
2	If set, memory class syllables are supported.
3	If set, branch class syllables and syllables with stop bits are supported.
4..7	Always zero, reserved for future expansion.

For the purpose of generic binaries, the configuration is repeated beyond the number of physically available lanes.

4.1.9 `CR_GPS1` - General purpose register delay register B

Offset	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
`0x0D0`																																	`CR_GPS1`
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
Core																																	
Debug																																	

This register is reserved for future expansion.

4.1.10 `CR_GPS0` - General purpose register delay register A

Offset	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
`0x0D4`					MEMAR				MEMDC				MEMDR				MULC				MULR				ALUC				ALUR				`CR_GPS0`
Reset	0	0	0	0	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	
Core																																	
Debug																																	

This register lists the key pipeline stages in which the core appears to read from and write to the general purpose register file. Forwarding is taken into consideration, so the core may not actually write to the register file in the listed stages, but from the perspective of the software it seems to.

From these values, the required number of bundles *between* an instruction that writes to a general purpose register and an instruction that reads from one can be determined, being $stage_{commit} - stage_{read} - 1$.

MEMAR field, bits 27..24

Hardwired to the stage in which the memory unit appears to read its address operands from the general purpose registers.

MEMDC field, bits 23..20

Hardwired to the stage in which the memory unit appears to commit the data loaded from memory to the general purpose registers.

MEMDR field, bits 19..16

Hardwired to the stage in which the memory unit appears to read the data to be stored to memory from the general purpose registers.

MULC field, bits 15..12

Hardwired to the stage in which the multiplier appears to commit its result to the general purpose registers.

MULR field, bits 11..8

Hardwired to the stage in which the multiplier appears to read its operands from the general purpose registers.

ALUC field, bits 7..4

Hardwired to the stage in which the ALU appears to commit its result to the general purpose registers.

ALUR field, bits 3..0

Hardwired to the stage in which the ALU appears to read its operands from the general purpose registers.

4.1.11 CR_SPS1 - Special delay register B

Offset	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0	
0x0D8					CR_SPS1
Reset	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	
Core					
Debug					

This register is reserved for future expansion.

4.1.12 CR_SPS0 - Special delay register A

Offset	31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0	
0x0DC	MEMMC	MEMMR	MEMDC	MEMDR	BRC	BRR	ALUC	ALUR	CR_SPS0
Reset	* * * *	* * * *	* * * *	* * * *	* * * *	* * * *	* * * *	* * * *	
Core									
Debug									

This register serves a similar purpose as CR_GPS0, but instead of being only for the general purpose registers, these values represents the delay for branch registers, the link register and memory.

MEMMC field, bits 31..28

Hardwired to the stage in which the memory unit actually commits the data from a store instruction to memory.

MEMMR field, bits 27..24

Hardwired to the stage in which the memory unit actually reads the data for a load operation from memory.

MEMDC field, bits 23..20

Hardwired to the stage in which the memory unit appears to commit the data loaded from memory to the link and branch registers.

MEMDR field, bits 19..16

Hardwired to the stage in which the memory unit appears to read the data to be stored to memory from the link and branch registers.

BRC field, bits 15..12

Hardwired to the stage in which the branch unit appears to commit the new program counter. This thus represents the number of branch delay slots. The next instruction is requested in stage 1 and its PC is forwarded combinatorially, thus the number of branch delay slots is $BRC - 2$. Note that the ρ-VEX processor does not actually execute its branch delay slots; it is invalidated when a branch is taken.

BRR field, bits 11..8

Hardwired to the stage in which the branch unit appears to read its operands from the branch and link registers.

ALUC field, bits 7..4

Hardwired to the stage in which the ALU appears to commit its result to the branch and link registers.

ALUR field, bits 3..0

Hardwired to the stage in which the ALU appears to read its operands from the branch and link registers.

4.1.13 `CR_EXT2` - Extension register 2

Offset	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
`0x0E0`																																	CR_EXT2
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
Core																																	
Debug																																	

This register is reserved for future expansion.

4.1.14 `CR_EXT1` - Extension register 1

Offset	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0x0E4																																	CR_EXT1
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
Core																																	
Debug																																	

This register is reserved for future expansion.

4.1.15 `CR_EXT0` - Extension register 0

Offset	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0x0E8					T	BRK							C	P																O	L	F	CR_EXT0
Reset	0	0	0	0	*	*	*	*	0	0	0	0	*	*	*	*	0	0	0	0	0	0	0	0	0	0	0	0	0	*	*	*	
Core																																	
Debug																																	

This register contains flags that specify the supported extensions and quirks of the processor as per its design-time configuration.

T flag, bit 27

Defines whether the trace unit is available. The trace unit has its own capability flags in `CR_DCR2`.

BRK field, bits 26..24

Defines the number of available hardware breakpoints.

C flag, bit 19

If set, cache-related performance counters exist.

P field, bits 18..16

This field represents the size in bytes of all performance counters except `CR_CNT`, which is always 64-bit. Refer to Section 4.3 for more information.

O flag, bit 2

This flag determines the unit in which the branch offset field is encoded. When this flag is cleared, the branch offset is encoded in 8-byte units. When it is set, the branch offset is encoded in 4-byte units.

L flag, bit 1

This flag is set when register `$r0.63` is mapped to `$l0.0`, to allow arithmetic to be performed on the link register directly. If it is cleared, these registers are independent.

F flag, bit 0

This flag is set when forwarding is enabled.

4.1.16 CR_DCFG - Design-time configuration register

Offset	31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0	
0x0EC		BA	NC	NG	NL	CR_DCFG
Reset	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	* * * *	* * * *	* * * *	* * * *	
Core						
Debug						

This register is hardwired to the key parameters that define the size of the processor, such as the number of pipelanes and the number of contexts.

BA field, bits 15..12

Specifies the minimum bundle alignment necessary. Specified as the alignment size in 32-bit words minus 1. For example, if this value is 7, each bundle must start on a 128-byte boundary, as $(7+1) \cdot 32 = 128$.

NC field, bits 11..8

Number of hardware contexts supported, minus one.

NG field, bits 7..4

Number of pipelane groups supported, minus one. This determines the degree of reconfigurability. Together with NC, it fully specifies the number of valid configuration words.

NL field, bits 3..0

Number of pipelanes in the design, minus one.

4.1.17 CR_CVER1 - Core version register 1

Offset	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0	
0x0F0	VER	CTAG0	CTAG1	CTAG2	CR_CVER1
Reset	0 0 1 1 0 0 1 1	0 * * * * * * *	0 * * * * * * *	0 * * * * * * *	
Core					
Debug					

This register specifies the major version of the processor and, together with CR_CVER0, a 7-byte ASCII core version identification tag.

VER field, bits 31..24, a.k.a. CR_CVER

Specifies the major version number of the ρ-VEX processor in ASCII. This will most likely always be '3'.

CTAG0 field, bits 23..16, a.k.a. `CR_CTAG`

First ASCII character in a string of seven characters, which together identify the core version, similar to how a license plate identifies a car. It is intended that a database will be set up which maps each tag to an immutable archive containing the source code for the core and a mutable errata/notes file.

4.1.18 `CR_CVER0` - Core version register 0

Offset	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0	
`0x0F4`	CTAG3	CTAG4	CTAG5	CTAG6	`CR_CVER0`
Reset	0 * * * * * * *	0 * * * * * * *	0 * * * * * * *	0 * * * * * * *	
Core					
Debug					

Refer to `CR_CVER1` for more information.

4.1.19 `CR_PVER1` - Platform version register 1

Offset	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0	
`0x0F8`	COID	PTAG0	PTAG1	PTAG2	`CR_PVER1`
Reset	* * * * * * * *	0 * * * * * * *	0 * * * * * * *	0 * * * * * * *	
Core					
Debug					

This register specifies the processor index within a platform and, together with `CR_PVER0`, uniquely identifies the platform using a 7-byte ASCII idenfitication tag.

COID field, bits 31..24, a.k.a. `CR_COID`

Unique processor identifier within a multicore platform.

PTAG0 field, bits 23..16, a.k.a. `CR_PTAG`

First ASCII character in a string of seven characters, which together identify the platform and bit file, similar to how a license plate identifies a car. It is intended that a database will be set up which maps each tag to an immutable archive containing the source code for the platform, synthesis logs and a bit file, as well as mutable `memory.map`, `rvex.h` and errata/notes files.

4.1.20 `CR_PVER0` - Platform version register 0

Offset	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0	
`0x0FC`	PTAG3	PTAG4	PTAG5	PTAG6	`CR_PVER0`
Reset	0 * * * * * * *	0 * * * * * * *	0 * * * * * * *	0 * * * * * * *	
Core					
Debug					

Refer to `CR_PVER1` for more information.

4.2 Context control registers

The following table lists the context control registers of the ρ-VEX processor. The offsets listed are with respect to the control register base address. If you are viewing this manual digitally, you can click the register mnemonics on the right to jump to their documentation.

<table>
<tr><th>Offset</th><th>31</th><th>30</th><th>29</th><th>28</th><th>27</th><th>26</th><th>25</th><th>24</th><th>23</th><th>22</th><th>21</th><th>20</th><th>19</th><th>18</th><th>17</th><th>16</th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th><th></th></tr>
<tr><td>0x200</td><td colspan="8">CAUSE</td><td colspan="8">BRANCH</td><td colspan="6"></td><td colspan="2">K</td><td colspan="2">C</td><td colspan="2">B</td><td colspan="2">R</td><td colspan="2">I</td><td>CR_CCR</td></tr>
<tr><td>0x204</td><td colspan="8">ID</td><td colspan="14"></td><td colspan="2">K</td><td colspan="2">C</td><td colspan="2">B</td><td colspan="2">R</td><td colspan="2">I</td><td>CR_SCCR</td></tr>
<tr><td>0x208</td><td colspan="32">LR</td><td>CR_LR</td></tr>
<tr><td>0x20C</td><td colspan="32">PC</td><td>CR_PC</td></tr>
<tr><td>0x210</td><td colspan="32">TH</td><td>CR_TH</td></tr>
<tr><td>0x214</td><td colspan="32">PH</td><td>CR_PH</td></tr>
<tr><td>0x218</td><td colspan="32">TP</td><td>CR_TP</td></tr>
<tr><td>0x21C</td><td colspan="32">TA</td><td>CR_TA</td></tr>
<tr><td>0x220</td><td colspan="32">BR0</td><td>CR_BR0</td></tr>
<tr><td>0x224</td><td colspan="32">BR1</td><td>CR_BR1</td></tr>
<tr><td>0x228</td><td colspan="32">BR2</td><td>CR_BR2</td></tr>
<tr><td>0x22C</td><td colspan="32">BR3</td><td>CR_BR3</td></tr>
<tr><td>0x230</td><td>D</td><td>J</td><td></td><td>I</td><td>E</td><td>R</td><td>S</td><td>B</td><td colspan="8">CAUSE</td><td colspan="2"></td><td colspan="2">BR3</td><td colspan="2"></td><td colspan="2">BR2</td><td colspan="2"></td><td colspan="2">BR1</td><td colspan="2"></td><td colspan="2">BR0</td><td>CR_DCR</td></tr>
<tr><td>0x234</td><td colspan="8">RESULT</td><td colspan="8"></td><td colspan="8">TRCAP</td><td>T</td><td>M</td><td>R</td><td>C</td><td>I</td><td colspan="2"></td><td>E</td><td>CR_DCR2</td></tr>
<tr><td>...</td><td colspan="32"><i>Unused</i></td><td></td></tr>
<tr><td>0x240</td><td colspan="32">CRR</td><td>CR_CRR</td></tr>
<tr><td>0x244</td><td colspan="32"><i>Unused</i></td><td></td></tr>
<tr><td>0x248</td><td colspan="32">WCFG</td><td>CR_WCFG</td></tr>
<tr><td>0x24C</td><td colspan="24"></td><td colspan="7">RUN</td><td>S</td><td>CR_SAWC</td></tr>
<tr><td>0x250</td><td colspan="32">SCRP1</td><td>CR_SCRP1</td></tr>
<tr><td>0x254</td><td colspan="32">SCRP2</td><td>CR_SCRP2</td></tr>
<tr><td>0x258</td><td colspan="32">SCRP3</td><td>CR_SCRP3</td></tr>
<tr><td>0x25C</td><td colspan="32">SCRP4</td><td>CR_SCRP4</td></tr>
<tr><td>0x260</td><td colspan="32">RSC</td><td>CR_RSC</td></tr>
<tr><td>0x264</td><td colspan="32">CSC</td><td>CR_CSC</td></tr>
<tr><td>0x268</td><td colspan="32">RSC1</td><td>CR_RSC1</td></tr>
<tr><td>0x26C</td><td colspan="32">CSC1</td><td>CR_CSC1</td></tr>
<tr><td>0x270</td><td colspan="32">RSC2</td><td>CR_RSC2</td></tr>
<tr><td>0x274</td><td colspan="32">CSC2</td><td>CR_CSC2</td></tr>
<tr><td>0x278</td><td colspan="32">RSC3</td><td>CR_RSC3</td></tr>
<tr><td>0x27C</td><td colspan="32">CSC3</td><td>CR_CSC3</td></tr>
<tr><td>0x280</td><td colspan="32">RSC4</td><td>CR_RSC4</td></tr>
<tr><td>0x284</td><td colspan="32">CSC4</td><td>CR_CSC4</td></tr>
<tr><td>0x288</td><td colspan="32">RSC5</td><td>CR_RSC5</td></tr>
<tr><td>0x28C</td><td colspan="32">CSC5</td><td>CR_CSC5</td></tr>
<tr><td>0x290</td><td colspan="32">RSC6</td><td>CR_RSC6</td></tr>
<tr><td>0x294</td><td colspan="32">CSC6</td><td>CR_CSC6</td></tr>
<tr><td>0x298</td><td colspan="32">RSC7</td><td>CR_RSC7</td></tr>
<tr><td>0x29C</td><td colspan="32">CSC7</td><td>CR_CSC7</td></tr>
<tr><td>...</td><td colspan="32"><i>Unused</i></td><td></td></tr>
<tr><td>0x300</td><td colspan="8">CYC3</td><td colspan="8">CYC2</td><td colspan="8">CYC1</td><td colspan="8">CYC0</td><td>CR_CYC</td></tr>
<tr><td>0x304</td><td colspan="8">CYC6</td><td colspan="8">CYC5</td><td colspan="8">CYC4</td><td colspan="8">CYC3</td><td>CR_CYCH</td></tr>
<tr><td>0x308</td><td colspan="8">STALL3</td><td colspan="8">STALL2</td><td colspan="8">STALL1</td><td colspan="8">STALL0</td><td>CR_STALL</td></tr>
<tr><td>0x30C</td><td colspan="8">STALL6</td><td colspan="8">STALL5</td><td colspan="8">STALL4</td><td colspan="8">STALL3</td><td>CR_STALLH</td></tr>
<tr><td>0x310</td><td colspan="8">BUN3</td><td colspan="8">BUN2</td><td colspan="8">BUN1</td><td colspan="8">BUN0</td><td>CR_BUN</td></tr>
<tr><td>0x314</td><td colspan="8">BUN6</td><td colspan="8">BUN5</td><td colspan="8">BUN4</td><td colspan="8">BUN3</td><td>CR_BUNH</td></tr>
<tr><td>0x318</td><td colspan="8">SYL3</td><td colspan="8">SYL2</td><td colspan="8">SYL1</td><td colspan="8">SYL0</td><td>CR_SYL</td></tr>
</table>

Offset	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0	
0x31C	SYL6	SYL5	SYL4	SYL3	CR_SYLH
0x320	NOP3	NOP2	NOP1	NOP0	CR_NOP
0x324	NOP6	NOP5	NOP4	NOP3	CR_NOPH
0x328	IACC3	IACC2	IACC1	IACC0	CR_IACC
0x32C	IACC6	IACC5	IACC4	IACC3	CR_IACCH
0x330	IMISS3	IMISS2	IMISS1	IMISS0	CR_IMISS
0x334	IMISS6	IMISS5	IMISS4	IMISS3	CR_IMISSH
0x338	DRACC3	DRACC2	DRACC1	DRACC0	CR_DRACC
0x33C	DRACC6	DRACC5	DRACC4	DRACC3	CR_DRACCH
0x340	DRMISS3	DRMISS2	DRMISS1	DRMISS0	CR_DRMISS
0x344	DRMISS6	DRMISS5	DRMISS4	DRMISS3	CR_DRMISSH
0x348	DWACC3	DWACC2	DWACC1	DWACC0	CR_DWACC
0x34C	DWACC6	DWACC5	DWACC4	DWACC3	CR_DWACCH
0x350	DWMISS3	DWMISS2	DWMISS1	DWMISS0	CR_DWMISS
0x354	DWMISS6	DWMISS5	DWMISS4	DWMISS3	CR_DWMISSH
0x358	DBYPASS3	DBYPASS2	DBYPASS1	DBYPASS0	CR_DBYPASS
0x35C	DBYPASS6	DBYPASS5	DBYPASS4	DBYPASS3	CR_DBYPASSH
0x360	DWBUF3	DWBUF2	DWBUF1	DWBUF0	CR_DWBUF
0x364	DWBUF6	DWBUF5	DWBUF4	DWBUF3	CR_DWBUFH

4.2.1 CR_CCR - Main context control register

Offset	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0x200	CAUSE								BRANCH														K		C		B		R		I		CR_CCR
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1	0	1	0	1	0	
Core																									✓	✓	✓	✓	✓	✓	✓	✓	
Debug	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓									✓	✓	✓	✓	✓	✓	✓	✓	

The primary purpose of the context control register is to store the primary control flags of the processor, for example whether interrupts are enabled. In addition, it also stores the trap cause and exposes the branch register file to the debug bus.

CAUSE field, bits 31..24, a.k.a. CR_TC

Trap cause. Set to the trap cause by hardware when the trap handler is called. Reset to 0 by hardware when an RFI instruction is encountered. Read-write by the debug bus, but the processor cannot write to this register.

BRANCH field, bits 23..16, a.k.a. CR_BR

Branch register file. Contains the current state of the branch registers. Only intended for use by the debug bus to see and modify the state of the branch register file. While the core is running, accessing this register is undefined due to it being dependent on the pipeline and forwarding state.

K field, bits 9..8

This register selects between kernel mode and user mode. Kernel mode is activated when the core is reset and when entering the trap or panic handlers. These must thus always point to code in hardware memory space. When `RFI` is executed, the state is restored from `CR_SCCR`.

In kernel mode, the register reads as `01`, while in user mode, it reads as `10`. The only way to enter user mode is by writing the user mode command to `CR_SCCR` and subsequently executing `RFI`. Neither the core nor the debug bus can write to this field directly.

Currently, the status of the kernel mode flag has no effect on the ρ-VEX. However, it is intended that this register will be used in the future for memory protection and/or security features. In particular, the reason that this flag can only be set by entering a trap and cleared by executing `RFI`, is that both of these mechanisms can be configured to do a full pipeline flush, which allows this flag to control whether address translation is enabled or disabled.

C field, bits 7..6

This register controls whether the context switch trap is enabled. It does not exist on hardware context 0. When the core is reset or the trap service routine is entered, the context switch trap is disabled. When `RFI` is executed, the state is restored from `CR_SCCR`.

When the context switch trap is enabled, this register reads as `01`. When it is disabled, it reads as `10`. Both the core and the debug bus can write to this register. Writing `00` has no effect, writing `01` enables the context switching trap, writing `10` disables it and writing `11` toggles the state. This prevents the need for read-modify-write operations.

Refer to `CR_RSC` for more information.

B field, bits 5..4

This register controls whether breakpoints are enabled in self-hosted debug mode. Its value is ignored in external debug mode. When the core is reset or the trap service routine is entered due to a debug trap in self-hosted debug mode, breakpoints are disabled. When `RFI` is executed, the state is restored from `CR_SCCR`.

When breakpoints are enabled, this register reads as `01`. When they are disabled, it reads as `10`. Both the core and the debug bus can write to this register. Writing `00` has no effect, writing `01` enables debug traps, writing `10` disables them and writing `11` toggles the state. This prevents the need for read-modify-write operations.

R field, bits 3..2

This register, named ready-for-trap, tentatively specifies if the processor is currently capable of servicing traps. However, since traps cannot be masked, any trap that occurs while ready-for-trap is cleared will cause a panic. Therefore, the only thing this register does in hardware is switch between the trap handler and panic handler address. When the core is reset or the trap service routine is entered, ready-for-trap is cleared. When `RFI` is executed, the state is restored from `CR_SCCR`.

When ready-for-trap is set (trap handler selected), this register reads as `01`. When it is cleared (panic handler selected), it reads as `10`. Both the core and the debug bus can write to this register. Writing `00` has no effect, writing `01` sets ready-for-trap, writing `10` clears it and writing `11` toggles the state. This prevents the need for read-modify-write operations.

I field, bits 1..0

This register selects whether external interrupts are enabled or not. When the core is reset or the trap service routine is entered, external interrupts are disabled. When `RFI` is executed, the state is restored from `CR_SCCR`.

When interrupts are enabled, this register reads as `01`. When they are disabled, it reads as `10`. Both the core and the debug bus can write to this register. Writing `00` has no effect, writing `01` enables external interrupts, writing `10` disables them and writing `11` toggles the state. This prevents the need for read-modify-write operations.

4.2.2 `CR_SCCR` - Saved context control register

Offset	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
`0x204`	ID																						K		C		B		R		I		`CR_SCCR`
Reset	*	*	*	*	*	*	*	*	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0	1	0	1	0	
Core																							✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
Debug																							✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	

This register saves the state of the primary control flags of the processor when entering the trap service routine. When `RFI` is executed, the state is restored from this register. In addition, this register contains the context ID, which contexts may read to identify themselves.

ID field, bits 31..24, a.k.a. `CR_CID`

This field is hardwired to the context index. Programs running on the ρ-VEX processor may use this field to determine which hardware context they are running on.

Note that `CR_CID` is not unique in a multi-processor system. If a unique processor ID is needed in such a case, `CR_COID` should be used as well.

K field, bits 9..8

When the trap service routine is entered, this register stores whether kernel the processor was in kernel mode or user mode. When `RFI` is executed, the state is set to this value.

Unlike the kernal mode field in `CR_CCR`, this field can be written. Writing `00` has no effect, writing `01` selects kernel mode, writing `10` selects user mode and writing `11` toggles the state. This prevents the need for read-modify-write operations. Read behavior is identical to the K field in `CR_CCR`.

C field, bits 7..6

When the trap service routine is entered, this register stores whether the context switching trap was enabled. When `RFI` is executed, the state is set to this value.

Core and debug bus access behavior is identical to the C field in `CR_CCR`.

B field, bits 5..4

When the trap service routine is entered, this register stores whether self-hosted debug breakpoints were enabled. When `RFI` is executed, the state is set to this value.

Core and debug bus access behavior is identical to the B field in `CR_CCR`.

R field, bits 3..2

When the trap service routine is entered, this register stores whether ready-for-trap was set. When `RFI` is executed, the state is set to this value.

Core and debug bus access behavior is identical to the R field in `CR_CCR`.

I field, bits 1..0

When the trap service routine is entered, this register stores whether interrupts were enabled. When `RFI` is executed, the state is set to this value.

Core and debug bus access behavior is identical to the I field in `CR_CCR`.

4.2.3 `CR_LR` - Link register

Offset	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0x208	LR																																CR_LR
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
Core																																	
Debug	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	

Contains the current link register (`$l0.0`) value. Only intended for use by the debug bus. While the core is running, accessing this register is undefined due to it being dependent on the pipeline and forwarding state.

4.2.4 `CR_PC` - Program counter

Offset	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0x20C	PC																																CR_PC
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
Core																																	
Debug	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	

Contains the current program counter. Only intended for use by the debug bus. When the register is written by the debug bus, the jump flag in `CR_DCR` is set, to ensure that the branch unit properly jumps to the new PC. This works even if the processor is running.

4.2.5 `CR_TH` - Trap handler

Offset	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0x210	TH																																CR_TH
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
Core	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
Debug	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	

Contains the address of the trap service routine. This is where the processor will jump to if a trap occurs while ready-for-trap in `CR_CCR` is set. Even if the design contains an MMU, this should be a hardware address, as the MMU is disabled when a trap occurs.

4.2.6 `CR_PH` - Panic handler

Offset	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0x214	PH																																CR_PH
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
Core	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
Debug	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	

Contains the address of the panic service routine. This is where the processor will jump to if a trap occurs while ready-for-trap in `CR_CCR` is NOT set. Even if the design contains an MMU, this should be a hardware address, as the MMU is disabled when a trap occurs.

The difference between the trap and panic service routines, is that the trap service routine has all state information of the processor at its disposal. That is, if the trap is recoverable, the program can continue after the trap service routine completes. The panic service routine, however, should assume that the state information of the processor is incomplete. Refer to Section 5 for more information.

4.2.7 `CR_TP` - Trap point

Offset	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0x218	TP																																CR_TP
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
Core	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
Debug	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	

When a trap occurs, this register is set to the address of the start of the offending bundle. The address is in user space if the MMU was enabled when the trap occured. In addition, when `RFI` is executed, the processor will jump back to this address to resume execution. This is the correct behavior for both external interrupts and traps that, after servicing, should return to the previously offending instruction, such as a page fault.

To support software context switching, the processor may write to this register to change the resumption address. `RFI` will then cause execution to be resumed in the new software context, assuming the rest of the processor state has been swapped in as well.

4.2.8 `CR_TA` - Trap argument

Offset	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0x21C	TA																																CR_TA
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
Core																																	
Debug	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	

When a trap occurs, this register is set to the trap argument. The significance of this value depends on the trap, which can be identified from the trap cause field in `CR_CCR`. Refer to Section 5 for more information.

4.2.9 `CR_BRn` - Breakpoint n

Offset	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0x220	BR0																																CR_BR0
0x224	BR1																																CR_BR1
0x228	BR2																																CR_BR2
0x22C	BR3																																CR_BR3
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
Core	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
Debug	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	

These registers hold the addresses for the hardware breakpoints and/or watchpoints. These registers only exist up to how many break-/watchpoints are design-time configured to be supported by the processor. The functionality of the breakpoints is configured in `CR_DCR`. These registers are always writable by the debug bus, but they are only writable by the core when the E flag is cleared, i.e. when self-hosted debug mode is selected.

4.2.10 `CR_DCR` - Debug control register 1

Offset	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0x230	D	J		I	E	R	S	B	CAUSE										BR3				BR2				BR1				BR0		CR_DCR
Reset	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
Core							✓												✓	✓			✓	✓			✓	✓			✓	✓	
Debug	✓			✓	✓	✓	✓	✓											✓	✓			✓	✓			✓	✓			✓	✓	

This register controls the debugging system of the ρ-VEX processor.

D flag, bit 31

Done/reset flag. This bit is set by hardware when a `STOP` instruction is encountered. It is cleared when a one is written to the R or S flags.

In addition, when a one is written to this flag, the control register file for this context is completely reset, as if the external context reset signal was asserted. Writing a zero has no effect. When combined with writing a one to the external debug flag, the core starts in external debug mode, and when combined with writing a one to B or the S flag, the core will stop execution before any instruction is executed, allowing the user to single-step from the start of the program. This works because I, E, S and B are not affected by a context reset.

Note that breakpoint information will have to be reloaded when the context is reset using this method.

J flag, bit 30

This bit is set by hardware when the debug bus writes to the PC register and is cleared when the processor jumps to it. It can thus be used as an acknowledgement flag for jumping. The flag is read only.

I flag, bit 28

Internal debug flag. Complement of the external debug flag. When the debug bus writes a one to this flag, the external debug flag is cleared, giving the processor control over debugging. Writing a zero has no effect. This flag is not affected by a context reset; it is only reset when the entire core is reset.

E flag, bit 27

External debug flag. Complement of the internal debug flag. When the debug bus writes a one to this flag, the external debug flag is set, enabling external debug mode. Writing a zero has no effect. This flag is not affected by a context reset; it is only reset when the entire core is reset.

While in external debug mode, debug traps cause the B flag to be set and the trap cause to be recorded in `CR_DCR` instead of the normal registers. This thus allows an external debugger to handle the debug traps instead, even if the processor is in the middle of a trap service routine and is not even ready for a trap. Writing a one to the R or the S flag is the equivalent of `RFI` for the external debugging system.

R flag, bit 26

Resume flag. When the debug bus writes a one to this flag, the B flag is cleared, causing the processor to resume execution if it was halted. Writing a zero has no effect; this flag is cleared by hardware when the first instruction is successfully fetched. It can thus be used as an acknowledgement flag for resuming execution.

In addition, debug traps are disabled for instructions that were fetched while this flag was set. This behavior allows the processor to step beyond the breakpoint that caused the processor to break, so there is no need to disable the triggered breakpoint in order to resume. This behavior is also used for single stepping; see below.

S flag, bit 25

Step flag. This flag may be set by the debug bus by writing a one to it. Doing so will also cause the R flag to be set and the B flag to be cleared, causing the processor to resume execution if it was halted. Writing a zero has no effect. The processor can also set this flag, but only if the E flag is cleared, i.e., if the processor is in self-hosted debug mode. This flag is not affected by a context reset; it is only reset when the entire core is reset.

While set, any instruction will cause a step debug trap. However, as noted above, all debug traps are disabled for the first instruction fetched after execution resumes. They should also be disabled while in the trap service routine through the breakpoint enable field in `CR_CCR`. This allows both an external debugger and the self-hosted debug system to single-step.

B flag, bit 24

Break flag. When this flag is set, the context stops fetching instructions and flushes the pipeline, as it would if the external run signal is low or if a reconfiguration is pending. It effectively halts execution. This flag is not affected by a context reset; it is only reset when the entire core is reset.

This flag may be set by the debug bus by writing a one to it, in order to pause execution. Writing a zero has no effect. In addition, the flag is set by hardware when a debug trap occurs while the E flag is set and when a `STOP` instruction is executed.

CAUSE field, bits 23..16, a.k.a. `CR_DCRC`

Trap cause for debug traps that should be handled by the external debug system. This is set to the debug trap cause by hardware when the B flag is set due to a debug trap. It is cleared when a one is written to resume or step, and set to `0x01` if a one is written to the B flag.

BR3 field, bits 13..12

Breakpoint 3 control field. This field only exists if the core is design-time configured to support all four hardware breakpoints. See also BR0.

BR2 field, bits 9..8

Breakpoint 2 control field. This field only exists if the core is design-time configured to support at least three hardware breakpoints. See also BR0.

BR1 field, bits 5..4

Breakpoint 1 control field. This field only exists if the core is design-time configured to support at least two hardware breakpoints. See also BR0.

BR0 field, bits 1..0

Breakpoint 0 control field. This field only exists if the core is design-time configured to support at least one hardware breakpoint.

The core can only write to BRn fields when the E flag is cleared, i.e. when self-hosted debug mode is selected. The encoding for the fields is as follows.

BRn = `00`: breakpoint/watchpoint disabled.
BRn = `01`: breakpoint enabled.
BRn = `10`: data write watchpoint enabled.
BRn = `11`: data read/write watchpoint enabled.

4.2.11 `CR_DCR2` - Debug control register 2

Offset	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
`0x234`	RESULT																TRCAP								T	M	R	C	I			E	`CR_DCR2`
Reset	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	*	*	*	*	*	0	0	*	0	0	0	0	0	0	0	0	
Core	✓	✓	✓	✓	✓	✓	✓	✓																	✓	✓	✓	✓	✓			✓	
Debug	✓	✓	✓	✓	✓	✓	✓	✓																	✓	✓	✓	✓	✓			✓	

This register controls the trace unit, if the core is design-time configured to support tracing. It also contains an 8-bit scratchpad register for communicating an execution result to the debug system.

RESULT field, bits 31..24, a.k.a. `CR_RET`

This field does not have a hardwired function. It is intended to be used to communicate the reason for executing a `STOP` instruction to the debug system. The default `_start.s` file will write the `main()` return value to this register before stopping.

TRCAP field, bits 15..8

This field lists the tracing capabilities of the core. The bit indices in this byte correspond to the bit indices in the control byte (the least significant byte of `CR_DCR2`). If a bit is high, the feature is available.

T flag, bit 7

Setting this bit enables trap tracing if the E flag is set and the core is design-time configured to support it.

M flag, bit 6

Setting this bit enables memory/control register tracing if the E flag is set and the core is design-time configured to support it.

R flag, bit 5

Setting this bit enables register write tracing if the E flag is set and the core is design-time configured to support it.

C flag, bit 4

Setting this bit enables cache performance tracing if the E flag is set and the core is design-time configured to support it.

I flag, bit 3

Setting this bit causes all fetched instructions to be traced if the E flag is set and the core is design-time configured to support it.

E flag, bit 0

Setting this bit enables tracing if the core is design-time configured to support it. If no other bits are set, only branch origins and destinations are traced.

4.2.12 `CR_CRR` - Context reconfiguration request register

Offset	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
`0x240`	CRR																																`CR_CRR`
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
Core	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
Debug																																	

This register may be written to by the core only. When it is written, a reconfiguration is requested. Refer to Section 6 for more information.

4.2.13 `CR_WCFG` - Wakeup configuration

Offset	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0x248	WCFG																																CR_WCFG
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
Core	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
Debug																																	

This register only exists on context 0. This configuration register is used in conjunction with the S flag in `CR_SAWC`. Refer to Section 6.3 for more information.

4.2.14 `CR_SAWC` - Sleep and wake-up control register

Offset	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0x24C																									RUN							S	CR_SAWC
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
Core																									✓	✓	✓	✓	✓	✓	✓	✓	
Debug																									✓	✓	✓	✓	✓	✓	✓		

This register only exists on context 0. This register contains special control features for sleeping (reconfiguring to a configuration with all lane groups disabled) and waking up other hardware contexts.

RUN field, bits 7..1

This field contains a bit for every other context, i.e., not all of these bits will be available if the core is not configured to support all eight hardware contexts. When reading this register, each bit represents the ones complement of the B flag in `CR_DCR` for each other context. Writing a one to a bit is equivalent to writing a one to the R flag in `CR_DCR` for each other context.

A scheduler running on context 0 may use this feature, combined with an interrupt controller that triggers an interrupt when the done output for any other context has a rising edge, to support task yielding for cooperative scheduling. A yield will then be equivalent to a `STOP` instruction, which will thus trigger an interrupt for the scheduler. The scheduler may then switch out the software context and subsequently restart the hardware context using these flags.

S flag, bit 0

Sleep flag. This enables or disables the sleep and wake-up system. Refer to Section 6.3 for more information.

4.2.15 CR_SCRPn - Scratchpad register *n*

Offset	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0x250	SCRP1																															CR_SCRP1
0x254	SCRP2																															CR_SCRP2
0x258	SCRP3																															CR_SCRP3
0x25C	SCRP4																															CR_SCRP4
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
Core	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
Debug	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	

Scratch pad registers. May be used at the discretion of the application and/or debug system.

4.2.16 CR_RSC - Requested software context

Offset	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0x260	RSC																															CR_RSC
Reset	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
Core	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
Debug	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	

This register does not exist on context 0. It is hardwired to RSC*n* in hardware context 0, and represents the software context that should be loaded into our hardware context, if it is not already loaded. The encoding of the register is at the user's discretion, but it is intended that this points to a memory region that contains the to be loaded context.

The contents of this register may also be written by hardware context 0 through RSC*n*, which is expected to run the scheduler. When this value does not equal the value in CSC and context switching is enabled in CR_CCR, the TRAP_SOFT_CTXT_SWITCH trap is caused. Refer to its documentation in Section 5 for more information.

4.2.17 CR_CSC - Current software context

Offset	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0x264	CSC																															CR_CSC
Reset	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
Core	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
Debug	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	

This register does not exist on context 0. It is hardwired to CSC*n* in hardware context 0. The value in this register should be set to the value in CR_RSC by the TRAP_SOFT_CTXT_SWITCH trap.

4.2.18 `CR_RSCn` - Requested swctxt on hwctxt *n*

Offset	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0	
`0x268`	RSC1				`CR_RSC1`
`0x270`	RSC2				`CR_RSC2`
`0x278`	RSC3				`CR_RSC3`
`0x280`	RSC4				`CR_RSC4`
`0x288`	RSC5				`CR_RSC5`
`0x290`	RSC6				`CR_RSC6`
`0x298`	RSC7				`CR_RSC7`
Reset	1 1 1 1 1 1 1 1	1 1 1 1 1 1 1 1	1 1 1 1 1 1 1 1	1 1 1 1 1 1 1 1	
Core	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	
Debug	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	

This register only exists on context 0, and only if the core is design-time configured to support hardware context n. This register is hardwired to `CR_RSC` in hardware context n. Refer to `CR_RSC` for more information.

4.2.19 `CR_CSCn` - Current swctxt on hwctxt *n*

Offset	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0	
`0x26C`	CSC1				`CR_CSC1`
`0x274`	CSC2				`CR_CSC2`
`0x27C`	CSC3				`CR_CSC3`
`0x284`	CSC4				`CR_CSC4`
`0x28C`	CSC5				`CR_CSC5`
`0x294`	CSC6				`CR_CSC6`
`0x29C`	CSC7				`CR_CSC7`
Reset	1 1 1 1 1 1 1 1	1 1 1 1 1 1 1 1	1 1 1 1 1 1 1 1	1 1 1 1 1 1 1 1	
Core					
Debug					

This register only exists on context 0, and only if the core is design-time configured to support hardware context n. This register is hardwired to `CR_CSC` in hardware context n. Refer to `CR_CSC` for more information.

4.2.20 `CR_CYC` - Cycle counter

Offset	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0	
`0x300`	CYC3	CYC2	CYC1	CYC0	`CR_CYC`
`0x304`	CYC6	CYC5	CYC4	CYC3	`CR_CYCH`
Reset	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	
Core				✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	
Debug				✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	

This performance counter increments every cycle while an instruction from this context is in the pipeline, even when the context is stalled.

Refer to Section 4.3 for more information about the structure of performance counters.

4.2.21 CR_STALL - Stall cycle counter

Offset	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0	
0x308	STALL3	STALL2	STALL1	STALL0	CR_STALL
0x30C	STALL6	STALL5	STALL4	STALL3	CR_STALLH
Reset	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	
Core				✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	
Debug				✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	

This performance counter increments every cycle while an instruction from this context is in the pipeline and the context is stalled. As long as neither CR_CYC nor CR_STALL have overflowed, CR_CYC - CR_STALL represents the number of active cycles.

Refer to Section 4.3 for more information about the structure of performance counters.

4.2.22 CR_BUN - Committed bundle counter

Offset	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0	
0x310	BUN3	BUN2	BUN1	BUN0	CR_BUN
0x314	BUN6	BUN5	BUN4	BUN3	CR_BUNH
Reset	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	
Core				✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	
Debug				✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	

This performance counter increments whenever the results of executing a bundle are committed. As long as neither CR_CYC, CR_STALL nor CR_BUN have overflowed, CR_CYC - CR_STALL - CR_BUN represents the number of cycles spent doing pipeline flushes, for example due to traps or the branch delay slot.

Refer to Section 4.3 for more information about the structure of performance counters.

4.2.23 CR_SYL - Committed syllable counter

Offset	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0	
0x318	SYL3	SYL2	SYL1	SYL0	CR_SYL
0x31C	SYL6	SYL5	SYL4	SYL3	CR_SYLH
Reset	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	
Core				✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	
Debug				✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	

This performance counter increments whenever the results of executing a non-NOP syllable are committed. As long as neither CR_BUN nor CR_SYL have overflowed, CR_SYL / CR_BUN represents average instruction-level parallelism since the registers were cleared.

Refer to Section 4.3 for more information about the structure of performance counters.

4.2.24 CR_NOP - Committed NOP counter

Offset	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0	
0x320	NOP3	NOP2	NOP1	NOP0	CR_NOP
0x324	NOP6	NOP5	NOP4	NOP3	CR_NOPH
Reset	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	
Core				✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	
Debug				✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	

This performance counter increments whenever a NOP syllable is committed. As long as neither CR_SYL nor CR_NOP have overflowed, CR_SYL / (CR_SYL + CR_NOP) represents average fraction of syllables that are NOP, i.e. the compression efficiency of the binary.

Refer to Section 4.3 for more information about the structure of performance counters.

4.2.25 CR_IACC - Instruction cache access counter

Offset	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0	
0x328	IACC3	IACC2	IACC1	IACC0	CR_IACC
0x32C	IACC6	IACC5	IACC4	IACC3	CR_IACCH
Reset	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	
Core				✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	
Debug				✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	

This performance counter increments for every instruction cache access.

Refer to Section 4.3 for more information about the structure of performance counters.

4.2.26 CR_IMISS - Instruction cache miss counter

Offset	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0	
0x330	IMISS3	IMISS2	IMISS1	IMISS0	CR_IMISS
0x334	IMISS6	IMISS5	IMISS4	IMISS3	CR_IMISSH
Reset	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	
Core				✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	
Debug				✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	

This performance counter increments every time there is a miss in the instruction cache.

Refer to Section 4.3 for more information about the structure of performance counters.

4.2.27 CR_DRACC - Data cache read access counter

Offset	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0	
0x338	DRACC3	DRACC2	DRACC1	DRACC0	CR_DRACC
0x33C	DRACC6	DRACC5	DRACC4	DRACC3	CR_DRACCH
Reset	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	
Core				✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	
Debug				✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	

This performance counter increments every time there is a read access to the data cache.

Refer to Section 4.3 for more information about the structure of performance counters.

4.2.28 CR_DRMISS - Data cache read miss counter

Offset	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0	
0x340	DRMISS3	DRMISS2	DRMISS1	DRMISS0	CR_DRMISS
0x344	DRMISS6	DRMISS5	DRMISS4	DRMISS3	CR_DRMISSH
Reset	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	
Core				✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	
Debug				✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	

This performance counter increments every time there is a read miss in the data cache.

Refer to Section 4.3 for more information about the structure of performance counters.

4.2.29 CR_DWACC - Data cache write access counter

Offset	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0x348	DWACC3								DWACC2								DWACC1								DWACC0								CR_DWACC
0x34C	DWACC6								DWACC5								DWACC4								DWACC3								CR_DWACCH
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
Core																									✓	✓	✓	✓	✓	✓	✓	✓	
Debug																									✓	✓	✓	✓	✓	✓	✓	✓	

This performance counter increments every time there is a write access to the data cache.

Refer to Section 4.3 for more information about the structure of performance counters.

4.2.30 CR_DWMISS - Data cache write miss counter

Offset	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0x350	DWMISS3								DWMISS2								DWMISS1								DWMISS0								CR_DWMISS
0x354	DWMISS6								DWMISS5								DWMISS4								DWMISS3								CR_DWMISSH
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
Core																									✓	✓	✓	✓	✓	✓	✓	✓	
Debug																									✓	✓	✓	✓	✓	✓	✓	✓	

This performance counter increments every time there is a write miss in the data cache.

Refer to Section 4.3 for more information about the structure of performance counters.

4.2.31 CR_DBYPASS - Data cache bypass counter

Offset	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0x358	DBYPASS3								DBYPASS2								DBYPASS1								DBYPASS0								CR_DBYPASS
0x35C	DBYPASS6								DBYPASS5								DBYPASS4								DBYPASS3								CR_DBYPASSH
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
Core																									✓	✓	✓	✓	✓	✓	✓	✓	
Debug																									✓	✓	✓	✓	✓	✓	✓	✓	

This performance counter increments every time there is a bypassed access to the data cache.

Refer to Section 4.3 for more information about the structure of performance counters.

4.2.32 CR_DWBUF - Data cache write buffer counter

Offset	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0x360	DWBUF3								DWBUF2								DWBUF1								DWBUF0								CR_DWBUF
0x364	DWBUF6								DWBUF5								DWBUF4								DWBUF3								CR_DWBUFH
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
Core																									✓	✓	✓	✓	✓	✓	✓	✓	
Debug																									✓	✓	✓	✓	✓	✓	✓	✓	

This performance counter increments every time the cache has to wait for the write buffer to flush in order to process the current request.

Refer to Section 4.3 for more information about the structure of performance counters.

4.3 Performance counter registers

All performance counters share the same nontrivial 64-bit structure, representing up to 56 bits worth of counter data. The actual size is design-time configurable using the CFG vector, and may be read from field P in `CR_EXT0`.

Each performance counter may be reset independently by writing an even value to the low register. Alternatively, all context-specific performance counters may be reset at the same time by writing an odd number to one of the performance counter low registers.

64-bit reads cannot be performed atomically in the ρ-VEX. Therefore, reliably reading the performance counters when they are configured to be larger than a 32-bit word is impossible to do in general, without additional hardware.

Typically, a holding register is implemented for either the low or the high word, which is loaded at the exact same time the other word is read. While this is fine in a single-processor environment, a multiprocessor environment would need such a holding register for each processor separately. To make matters worse, this holding register would also need to be saved and restored when a software context is swapped out. This makes this solution more trouble than it's worth.

In the ρ-VEX, this problem is not avoided completely, but it is mitigated. Each counter is limited to seven bytes, and the middle byte is mirrored by both the low and high register if the counter is larger than a 32-bit word. This permits the following algorithm for a semi-reliable performance counter read.

```c
/**
 * Loads a 40-bit, 48-bit or 56-bit performance counter value. Do not use this
 * when the counter size is set to 32-bit!
 */
uint64_t read_counter(
    volatile uint32_t *low,
    volatile uint32_t *high
) {

    // Perform the read.
    uint32_t l = *low;
    uint32_t h = *high;

    // Check if the counters have overflowed.
    if (l >> 24) != (h & 0xFF) {

        // There was an overflow, so clear the low value.
        l = 0;

    }

    // Combine the values and return.
    return ((uint64_t)h << 24) | l;

}
```

Note that this algorithm will *not* work when the counters are configured to be 32 bits wide. In this case the high word register is intentionally not implemented in order to save hardware, which means that the overflow check will not work properly.

The algorithm assumes that the value is monotonously increasing. This is true for all performance counters as long as it is impossible for them to be reset during the read. As long as there was no 32-bit overflow during the read, the returned value will always be a counter value between what is was when `low` was read and what it was when `high` was read. If there *was* such an overflow, there is a small chance (1/256 if the added value during the read would be uniformly distributed) that the returned value is slightly higher than what the counter value was when `high` was read.

As an example, the worst case scenario is that the counter is at 0xFFFFFF when `low` is read (`l` = 0xFFFFFF), and at 0x100000000 when `high` is read (`h` = (`l` = 0x100). This will result in 0x100FFFFFF being returned, or about 0.4% too much. This is, however, completely insignificant compared to the jitter which may be expected in the value when such a delay is possible between the two reads. It would require an extremely long interrupt service routine or software context switch happening at exactly the wrong time, and when such things are going on in the background.

Traps and interrupts 5

There are many systems in a processor that need to be able to interrupt normal program flow. For instance, an external interrupt may be requested, or a problem occured while trying to load a word from memory, such as a page fault. The naming conventions for such interruptions varies from processor to processor; in the ρ-VEX processor all such interruptions are called traps. The word ‘interrupt’ is reserved for the special trap that deals with external interrupts, i.e. asynchronous signals from outside the core. The word ‘fault’ is used to refer to traps that signal that an instruction could not be executed.

5.1 Trap sources

There are roughly six sources of traps in the ρ-VEX processor, which are handled in slightly different ways.

- Faults. A fault signals that an instruction could not be executed for some reason. They are always handled by the processor by jumping to the trap or panic handler. With the exception of page faults, these traps are usually non-recoverable, leading to abnormal termination of the executing task in an operating system environment, or the `STOP` instruction to be called in a bare-metal environment.

- Interrupts. The ρ-VEX processor core has an interface for an interrupt controller. When an interrupt is requested and interrupts are enabled through the I flag in `CR_CCR`, a `TRAP_EXT_INTERRUPT` trap will be generated. This trap causes the processor to jump to the trap or panic handler.

- Context switch request. When the values in `CR_RSC` and `CR_CSC` do not match and the context switching system is enabled by means of the the C flag in `CR_CCR`, a `TRAP_SOFT_CTXT_SWITCH` trap will be generated. This trap causes the processor to jump to the trap or panic handler.

- Breakpoints/debug traps. When a hardware or software breakpoint is hit while breakpoints are enabled through the B flag in `CR_CCR`, the processor will generate a debug trap. Debug traps can be handled in two ways, depending on the E and I flags in `CR_DCR`. When the I flag is set (this is the default), the traps will be handled as any other trap, i.e. by jumping to the trap or panic handler. However, when the E flag is set, the context will simply halt, and write the trap cause to the cause field in `CR_DCR`. This allows an external debugging system to handle the breakpoints instead of the processor itself. In addition, debug traps are disabled for every first instruction executed after returning from a trap handler or restarting the context, allowing either to jump over breakpoints.

- TRAP instructions. The TRAP instruction can be used to emulate any trap. If the cause maps to a debug trap, it is handled exactly as a debug trap, allowing it to be used as a software breakpoint. Otherwise, it is handled like a fault.

- STOP instructions. A STOP instruction halts the core by generating a TRAP_STOP trap during execution of the subsequent instruction. The TRAP_STOP trap is always handled by stopping the hardware context. In addition, the D flag in CR_DCR is set and the done output signal for the stopped hardware context is asserted.

5.2 Trap and panic handlers

As stated above, most traps are handled by jumping to the so-called trap or panic handler. These handlers are simply subroutines that typically end with either a RFI or STOP instruction. They should be pointed to by the CR_TH and CR_PH control registers; it is up to the initialization code to set up these links.

The hardware switches between the trap and panic handlers based on the R flag in CR_CCR. The only hardware difference between them is that this flag always switches to the panic handler upon servicing a trap, such that a trap that immediately follows another trap will always be handled by the panic handler.

The tentative difference between the two trap handlers is that one should attempt to jump back to the application (or alternatively, in the case of an operating system, kill the current process with the appropriate signal and context switch to another thread), and the other should not. The necessity of such a difference can be best illustrated with a simple example.

Consider a program that has just been trapped due to an interrupt. The first course of action in handling a trap must always be to save the state of the running program, so the trap cause and argument registers can be examined. Now consider that it is possible for these context-saving memory accesses to cause, say, a misaligned memory access, due to a programming error. A regular trap handler may in theory try to recover from the fault by emulating the faulting instruction and then jumping over it. However, if it would do so, the trap point, cause and argument control registers of the original interrupt trap will have been overwritten with the misaligned memory access. This data was simply lost when the second trap occured; there is no way around this. Thus, the program cannot continue.

Through the dual handler system as implemented in the ρ-VEX processor, the first trap will be handled by the regular trap handler. Upon jumping to this trap handler, the processor will automatically clear the ready-for-trap flag, such that the second trap will be handled by the panic handler.

What it comes down to, is that the trap handler may try to recover from a fault, handle an interrupt or breakpoint, etc., while the panic handler should simply display or log an error message if it can, and then stop execution or reset. The regular trap handler may also want to jump to the panic handler if it is posed with a fault trap that it cannot recover from.

5.3 Trap identification

In the ρ-VEX processor, traps are identified not by the address that the processor branches to (as there are only two of these addresses, as described in the previous section), but by the trap cause (`CR_TC`) and trap argument (`CR_TA`) control registers. The former stores an 8-bit value that identifies the cause of the trap. The latter is a 32-bit register whose significance depends on the trap cause.

The list below documents the trap causes as currently defined in the processor, and the significance of the trap argument. Note, however, that a `TRAP` instruction may emulate any of these traps with any argument.

- `TRAP_NONE` = 0x00

  Trap cause 0 is reserved to indicate normal operation. When an `RFI` instruction is executed, the trap cause register (cause field in `CR_CCR`) will be reset to 0, so an external debug system can always determine what a program is doing, unless nested traps are utilized.

- `TRAP_INVALID_OP` = 0x01

  This trap is generated by hardware in the following conditions.

  - An unknown opcode is encountered.
  - The stop bit was set such that the next bundle would start on an address violating the minimum design-time configured bundle alignment.
  - A branch opcode is encountered in a pipelane that does not have an active branch unit.
  - A memory opcode is encountered in a pipelane that is not design-time configured to include a memory unit.
  - A multiplier opcode is encountered in a pipelane that is not design-time configured to include a multiplier.

  The trap argument is set to the lane index that caused the trap.

- `TRAP_MISALIGNED_BRANCH` = 0x02

  This trap is generated by hardware when a branch to a misaligned address is requested. The trap argument is set to the branch target.

- `TRAP_FETCH_FAULT` = 0x03

  This trap is generated by hardware when an instruction fetch resulted in a bus fault. The trap argument is unused; the program counter can be determined from the trap point.

- `TRAP_MISALIGNED_ACCESS` = 0x04

  This trap is generated by hardware when a misaligned memory access was requested. That is, a 32-bit word access was attempted with an address that is not divisible by four, or a 16-bit word access was attempted with an odd address. The trap argument is set to the requested memory address.

- `TRAP_DMEM_FAULT` = 0x05

  This trap is generated by hardware when a data memory access resulted in a bus fault. The trap argument is set to the requested memory address.

- `TRAP_LIMMH_FAULT` = 0x06

  This trap is generated by hardware under the following conditions.

  - A `LIMMH` instruction is trying to forward to a lane for which no route is available in the core. Note that only the least significant bit of the target lane is actually checked, though. In this case, the trap argument is the index of the lane with the `LIMMH` instruction.
  - Two `LIMMH` instructions are trying to forward to the same lane. In this case, the trap argument is the index of the target lane.
  - A `LIMMH` instruction is attempting to forward to a syllable that is not using an immediate. In this case, the trap argument is also the index of the target lane.

- `TRAP_EXT_INTERRUPT` = 0x07

  This trap is generated by hardware when the external interrupt request line is asserted while interrupts are enabled by means of the I flag in `CR_CCR`. When the trap service routine is entered, the state of the external interrupt ID signal is saved as the trap argument in `CR_TA`, and in the same cycle, the interrupt is acknowledged. This ensures that the interrupt ID presented to the trap service routine always matches the acknowledged interrupt.

  There is a delay between the core registering that the external interrupt request line is asserted and generating the trap, and the actual entering of the trap service routine. This delay is due to the pipeline flush required to do this, and is in the order of a couple cycles; compared to actually servicing a trap this delay is negligible. However, if it is ever possible that an active interrupt is disabled before it is acknowledged by the core, it is possible that the core will enter the trap service routine due to an interrupt that was disabled before it could be handled. In this case, the interrupt controller should provide the core with an otherwise reserved interrupt ID indicating that there was no interrupt. The trap service routine should handle this special interrupt ID as no-operation.

- `TRAP_STOP` = 0x08

  This trap is generated by hardware in the instruction immediately following a `STOP` instruction. It is handled in a completely different way than the other traps are; the hardware will not jump to `CR_TH` or `CR_PH`. Instead, the D and B flags in `CR_DCR` are set, thus stopping execution, and the program counter is set to the trap point. This allows an external debugging or control system to resume processing after the stop trap by simply writing a one to the R flag in `CR_DCR`.

- TRAP_SOFT_CTXT_SWITCH = 0x09

  This trap is generated by hardware when the contents of CR_RSC differ from CR_CSC while this trap is enabled using the C flag in CR_CCR. The intended use of this trap is to allow hardware context 0 to control software context switching on the other hardware contexts. When used in this way, the trap service routine for this trap should perform the following tasks.

  – If CR_CSC $\neq$ -1, save the current context to the memory identified by CR_CSC.

  – Set CR_CSC to CR_RSC.

  – Restore the software context identified by CR_RSC from memory.

  The way in which CR_RSC and CR_CSC identify the software context to be exchanged is up to the operating system code.

- TRAP_SOFT_DEBUG_0 = 0xF8

- TRAP_SOFT_DEBUG_1 = 0xF9

- TRAP_SOFT_DEBUG_2 = 0xFA

  These traps are never generated by hardware, but are intended to be used as soft breakpoints using the TRAP instruction. That is, the debug system may override one of the syllables in a any bundle where a breakpoint is desired with a TRAP syllable. It may return control to the application by reverting the TRAP syllable back into the original syllable. If it is not the intention of the debugger to disable the breakpoint, it may single step over the instruction at the breakpoint, and then replace the TRAP syllable.

  Unlike the other undefined traps (which may be used as arbitrary software traps), these traps behave like hardware debug traps. That is, they will be handled by halting the core if the core is in external debug mode (i.e. the E flag in CR_DCR is set). This means that an external debugger can also use this system to support an arbitrary number of breakpoints.

  Likewise, disabling breakpoints using the B flag in CR_CCR will prevent even the TRAP instruction from actually generating a trap.

- TRAP_STEP_COMPLETE = 0xFB

  This trap is generated by hardware whenever the S flag in CR_DCR is set while debug traps are enabled. This allows the debug system to single-step. Refer to the documentation of CR_DCR for more information.

- TRAP_HW_BREAKPOINT_0 = 0xFC

- TRAP_HW_BREAKPOINT_1 = 0xFD

- TRAP_HW_BREAKPOINT_2 = 0xFE

- TRAP_HW_BREAKPOINT_3 $= 0\text{xFF}$

  These traps are generated by hardware when the corresponding hardware breakpoint or watchpoint is hit while debug traps are enabled.

5.4 State saving and restoration

Upon entering a trap, it is mostly up to the software to save and restore the processor state. Specifically, the software must ensure that the state of the general purpose registers, branch registers and the link register is as it was when the trap handler was entered when the RFI instruction is executed. The hardware will handle saving and restoration of the context control flags in CR_CCR and the program counter, as both of these are modified immediately when entering the trap handler. CR_CCR is saved in and restored from CR_SCCR, the program counter is saved in and restored from CR_TP.

Aside from restoring the state of the currently running task, an operating system environment may also wish to restore the state of a different task. In this case, the complete state of a task is defined by the contents of the general purpose register file, the branch register file, the link register, the program counter (to be accessed using CR_TP) and the context control register (to be accessed using CR_SCCR).

6 Reconfiguration and sleeping

The process in which the ρ-VEX processor switches between one large core and more smaller cores is called reconfiguration. Reconfigurations may be requested by the software running on the processor or the debugging interface by writing the requested configuration to a control register. The reconfiguration controller will then temporarily stop all contexts that will be affected by the reconfiguration, commit the new configuration, and (re)start any contexts that are part of the new configuration but are currently stopped.

6.1 Configuration word encoding

A configuration is described by means of a single register at most 32-bits in size. The actual size depends on the design-time configuration of the core; in particular, the number of lane groups and the number of contexts.

In the configuration word, each nibble (group of 4 bits, represented by a single hexadecimal digit) maps to a lane group. The nibble signifies the context that is to be run on that lane group. Disabling a lane group to save power is also possible, by selecting 'context' eight. This will never map to an actual context, as the maximum amount of hardware contexts supported by the design-time configuration system is also eight, and numbering starts at zero.

Obviously, not all 4.2 billion 32-bit values represent valid configurations. Configuration words must adhere to the following rules.

- The nibbles for existing pipelane groups may be set to either zero through the number of hardware contexts minus one to select a context, or eight to disable the pipelane group. For instance, the configuration word 0x7777 is illegal on an ρ-VEX processor that does not support eight hardware contexts. Configuration words like 0x9999 are reserved for future configurations, such as fault tolerant duplicate and triplicate modes.

- The nibbles for non-existant pipelane groups must be set to zero. For instance, the configuration word 0x88880000 is illegal for an ρ-VEX processor that is design-time configured to only support 4 lane groups, even though it may make more sense than the configuration word that was probably the intention here, which is simply zero.

- Any context may only be mapped to a power-of-two of contiguous pipelane groups. For instance, configuration words 0x1118 and 0x1231 are illegal, because the mapping for context 1 violates these rules.

- A set of pipelane groups mapped to a single context must be aligned. Mathematically, the index of the first pipelane group in the set must be divisible by the cardinality of the set. For instance, the configuration word 0x0112 is illegal, because the mapping for context 1 is improperly aligned.

The reconfiguration controller will ensure that a configuration word is valid before committing it to the processor. If an invalid configuration is requested, the E flag in `CR_GSR` is set and the request is otherwise ignored.

6.2 Requesting a reconfiguration

There are three ways in which a reconfiguration can be requested.

- Writing to the `CR_CRR` context control register from a program running on the core. This section primarily deals with this mechanism.
- Writing to the `CR_BCRR` global control register from the debug bus. This mechanism is equivalent to the first, except it is triggered from outside the core.
- Using the sleep and wake-up system, as described in Section sec:core-ug-reconf-saw.

Usually, when a reconfiguration is requested, the new configuration will be committed within something in the order of tens of cycles, depending on how long it takes the reconfiguration controller to pause the affected contexts. However, a reconfiguration may also be rejected, either another context or the bus is requesting a new configuration simultaneously and arbitration is lost, or because the requested configuration is invalid. The following C function correctly deals with arbitration, and performs a best-effort attempt at detecting errors without using locks implemented in software.

```
/**
 * Requests a reconfiguration. Returns 1 if reconfiguration was successful,
 * -1 if the requested configuration is invalid or 0 if it is not known
 * whether the configuration was valid or not.
 */
int reconfigure(unsigned int newConfiguration) {

    // Extract our own context ID from the register file, which we will use
    // to check if we won arbitration or not.
    int ourselves = CR_CID;

    // Used to store the ID of the winning context after the request.
    int winner;

    // Retry requesting the new configuration until we win arbitration.
    do {

        // Request the new configuration.
        CR_CRR = newConfiguration.

        // Load the GSR register for state information.
        gsr = CR_GSR;

        // Extract the reconfiguration requester ID field from GSR.
        int winner = (gsr & CR_GSR_RID_MASK) >> CR_GSR_RID_BIT;

    } while (winner != ourselves);

    // Busy-wait for reconfiguration to complete.
    while (gsr & CR_GSR_B_MASK) {
        gsr = CR_GSR;
    }

    // If our context is still the one that was the last to request a
    // reconfiguration, the error flag in GSR is also meant for us. If not,
    // there is no way to tell if the configuration we requested was valid
    // or not.
    if (((gsr & CR_GSR_RID_MASK) >> CR_GSR_RID_BIT) != ourselves) {
        return 0;
    }

    // If the error flag is set, return -1.
    if (gsr & CR_GSR_E_MASK) {
        return -1;
    }

    // Reconfiguration was successful.
    return 1;

}
```

6.3 Sleep and wake-up system

The sleep and wake-up system refers to two context control registers that only exist on context zero, through which the processor can be set up to automatically request a reconfiguration when the interrupt request input of context zero is asserted. More specifically, the wakeup system will activate when all of the following conditions are met.

- The S flag in `CR_SAWC` is set.
- An interrupt is pending on context 0.
- Context 0 is not already active in the current configuration.
- There is no reconfiguration in progress.

When activated, the following actions are performed.

- A reconfiguration to the configuration stored in `CR_WCFG` is requested.
- `CR_WCFG` is set to the old configuration.
- The S flag in `CR_SAWC` is cleared.

This system may be used to save power that is otherwise wasted in an idle loop, or to improve interrupt latency by dedicating hardware context zero to only handling interrupts. These use cases are described below.

6.3.1 Power saving

To conserve power, the user may want to switch to a configuration where all pipelane groups are idle until an interrupt occurs. This is called sleeping. On an FPGA this is merely a proof of concept, but in an ASIC the amount of power that might be saved by clock gating or powering down the computational resources may be very significant. To go to sleep, the program should take the following steps.

1. If other hardware contexts were running other tasks in parallel to context zero, which may be in a state in which the processor should not sleep, first request these tasks to pause gracefully. If necessary, request a reconfiguration to configuration zero, as described in Section 6.2. to disable all contexts except for context zero.

2. Disable interrupts using the I field in `CR_CCR`.

3. If necessary, ensure that no interrupt occured before interrupts were disabled that should cause the processor to stay awake. If this did happen, take the appropriate actions, such as re-enabling interrupts, before attempting to sleep again.

4. Copy `CR_CC`, the current configuration, to `CR_WCFG`, the wake-up configuration. This is an easy way to ensure that `CR_WCFG` will not contain an invalid configuration. Writing to `CR_WCFG` also sets the S flag in `CR_SAWC` to enable the wake-up system.

5. Request a reconfiguration to the configuration where all pipelane groups are disabled, for instance 0x8888 on a core that is design-time configured to have four pipelane groups, as described in Section 6.2.

6. Busy-loop until the S flag in `CR_SAWC` is cleared. This ensures that the program will not continue until after the processor has finished sleeping.

7. Enable interrupts using the I field in `CR_CCR` to service the interrupt. The fact that this is not done automatically also allows the interrupt request input to simply be used as a wake-up input in a simple system where no interrupts exist.

6.3.2 Decreasing interrupt latency

To decrease interrupt latency, context zero may be used as a dedicated context for servicing interrupts. This prevents the context zero trap handler from having to save and restore the state of the processor as it was before the interrupt trap, as this information is not relevant. The other hardware contexts may be used to run the main program; the reconfiguration system is then used for hardware context switching.

To initialize this system, the program should do the following in context zero.

1. Set up links to the trap and panic handlers for context 0 in `CR_TH` and `CR_PH`.

2. Copy `CR_CC`, the current configuration, to `CR_WCFG`, the wake-up configuration. This is an easy way to ensure that `CR_WCFG` will not contain an invalid configuration. Writing to `CR_WCFG` also sets the S flag in `CR_SAWC` to enable the wake-up system.

3. Request a reconfiguration as described in Section 6.2, to, for instance, 0x1111, if the main program is to run in hardware context 1.

4. Busy-loop until the S flag in `CR_SAWC` is cleared. This ensures that the program will not continue until after the first interrupt is requested.

5. Set ready-for-trap and enable interrupts using the R and I fields in `CR_CCR` to service the interrupt.

6. Busy-loop forever to wait for the interrupt to be serviced.

The other contexts can initialize in the usual manner. The context 0 trap handler should do the following.

1. Perform body of the regular trap handling tasks, i.e., everything except for saving and restoring the context and executing `RFI`.

2. Set ready-for-trap and enable interrupts using the R and I fields in `CR_CCR` to quickly service the next interrupt if one is already pending. Clear ready-for-trap and disable interrupts in the next cycle again; one cycle is enough for an interrupt to be handled.

3. Store the contents of `CR_WCFG` in a temporary register.

4. Copy `CR_CC`, the current configuration, to `CR_WCFG`, the wake-up configuration. This is an easy way to ensure that `CR_WCFG` will not contain an invalid configuration. Writing to `CR_WCFG` also sets the S flag in `CR_SAWC` to enable the wake-up system.

5. Request a reconfiguration to the configuration as stored in the temporary register, as described in Section 6.2.

6. Busy-loop until the S flag in `CR_SAWC` is cleared. This ensures that the program will not continue until after the first interrupt is requested.

7. Set ready-for-trap and enable interrupts using the R and I fields in `CR_CCR` to service the interrupt.

8. Busy-loop forever to wait for the interrupt to be serviced.

7 Debugging ρ-VEX software

There are two main approaches to debugging the ρ-VEX processor. This chapter documents the external debugger approach. In this approach, a computer is connected to the ρ-VEX is used to debug the processor and the software running on it. The computer is connected to the ρ-VEX using some interface, usually a serial port in the case of the ρ-VEX. The alternative approach is called self-hosted debug, where the debugger runs on the ρ-VEX itself in order to debug another thread. However, this approach requires a sophisticated multithreading operating system, such as a Linux kernel with the `ptrace` system call implemented for the ρ-VEX. Although the hardware should be ready for such a system, the software for it has not yet been implemented.

7.1 Setting up

The connection between the computer and the ρ-VEX is called the debug link. Currently, the following options exist.

- A serial port, through the ρ-VEX debug support peripheral.
- PCI express, developed in [8].
- Memory mapped on a Zynq FPGA, running Linaro Linux with `rvsrv` on the embedded ARM processor. The debug commands may be given in Linaro, or `rvd` can connect to the Zynq development board using ethernet.

Which connections are supported depends on the platform. The serial port option is available in all hardware platforms except for `zed-almarvi`. The PCI express connection is supported in addition to the the serial link by `ml605-grlib` to allow faster memory access. `zed-almarvi` only supports the memory-mapped option.

Whichever platform you use, you need to execute the following commands in a console from the root directory of the platform you are using to set up the debugging environment.

```
make debug
source debug
```

The first command generates a script called ‘debug’ that sets up environment variables to allow you to use `rvd`. The second command runs that script. The next step depends on whether the FPGA board is connected to your machine (Section 7.3) or to another machine (Section 7.2). In the latter case, you need to be able to `ssh` to that machine.

7.2 Connecting to a remote machine

To connect to the remote machine, we will use `ssh` to forward two TCP/IP ports. You can do this by running the following command in a second terminal (you will need to keep it running), obviously replacing `<user@host>` with the computer you are connecting to and your account name on that computer.

```
ssh -N -L 21078:localhost:21078 -L 21079:localhost:21079 <user@host>
```

Note that you will *not* drop to a terminal on the remote computer as `ssh` normally does. It will appear like it is not doing anything after requesting your password (if required). You can test the connection by running `rvd ?` in the original terminal. If that does not crash with the message `Failed to connect to rvsrv`, you are ready to move on to Section 7.4. Otherwise, `ssh` is not working, or more likely, `rvsrv` is not running on the remote machine. In the latter case, you can try to start it yourself by `ssh`'ing to the machine normally and following the steps in 7.3. If that does not work, you will have to ask the owner of the machine for help.

7.3 Connecting to the FPGA

This section assumes that you are using a serial port debug link. The PCI express connection is more complicated to set up due to the drivers required. If you are using the Zedboard, refer to the separate documentation in the `zed-almarvi` platform.

If this is the first time you are connecting to the FPGA, open the following file in a text editor.

```
<rvex-rewrite>/tools/debug-interface/configuration.cfg
```

If this file does not exist, create it by copying `default-configuration.cfg` from the `src` directory. This file describes the interfaces that the debug server (`rvsrv`) will connect to or expose. The relevant configuration key is `SERIAL_PORT`, which needs to be set to the `tty` corresponding to the serial port.

When that has been configured, the debug server can be started in the terminal in which we have sourced the `debug` script using the following command.

```
make server
```

You can now test the connection to the ρ-VEX by running `rvd ?`.

7.4 Running programs

The procedure for uploading and running a program differs from platform to platform, but usually, the following three commands will work.

```
make upload-<program>
make start-<program>
make run-<program>
```

The difference between them is that `upload` only uploads the program to the ρ-VEX without starting it, `start` uploads and then starts the program, and `run` also waits for completion and prints the performance counter values. Usually, running `make` without parameters will (among other things) print a list of the available programs.

7.5 Debugging programs

The standard and recommended way to send debug commands to the ρ-VEX is to use `rvd`. All documentation for using `rvd` is embedded inside the program: just run `rvd help`. To get command specific documentation, use `rvd help <command>`.

`rvd` has builtin commands for halting, resuming, single stepping, resetting execution and printing the current state of the processor, in addition to the raw memory access commands. More complicated things, such as breakpoints, need to be set manually by accessing the control registers of the ρ-VEX. You do not have to remember the control register addresses by heart though; you can use the control register names without `CR_` prefix directly.

`rvd` has a concept of contexts. By default, the debug interface for context 0 is used. To select a different context, you can either specify the context using the `-c` command line parameter (for example, `rvd -c3 resume`) or you can set it for future commands using the `rvd select` command. In addition to specifying a single context, you can also specify a range of contexts (`<from>..<to>` or all contexts (`all`). When more than one context is selected, `rvd` will simply execute the given command for all selected contexts sequentially.

An alternative to `rvd`'s interface, the `gdb` port can be used. In this case, the following command should be used.

```
rvd gdb -- <path_to_gdb> [parameters passed to gdb]
```

This runs `gdb` as a child process to `rvd`. The appropriate parameters are passed to `gdb` to have it connect to `rvd` using the remote serial protocol, in addition to the parameters specified on the command line. A description of how to use the ρ-VEX `gdb` port is beyond the scope of this manual.

7.6 Tracing execution

The ρ-VEX can be configured at design time to include a trace unit. This allows the hardware to output a stream of data describing everything that the processor is doing at various levels of detail. `rvd` supports tracing using the following command.

```
rvd trace <output_file> [level_of_detail] [condition]
```

When executed, `rvd` writes the specified level of detail or 1 by default to the trace control byte in `CR_DCR2`. It then resumes execution on the selected contexts and reads data from the trace buffer. Tracing stops when the specified condition evaluates to 0, or when no more data is available if no condition is specified.

Terminating a trace with `ctrl+c` is not recommended, because it prevents `rvd` from resetting the trace control byte and emptying the trace buffer. To terminate a trace gracefully when no condition is specified and the program is stuck in a loop, run `rvd break` in a separate terminal. This will make `rvd trace` assume that the program has finished executing.

`rvd trace` dumps the raw trace data to a file. This file can be converted to a human readable format using the `rvtrace` tool. If a disassembly file generated using `objdump -d` is specified in addition to the binary trace file, the disassembled instructions will be included in the trace output file.

Please note that the human readable trace files are much larger than the binary data format. It may thus take some time and a lot of disk space to generated the human readable file. You may want to pipe the output of `rvtrace` to `less` instead, so the output will only be saved in memory.

8 Design-time configuration

The ρ-VEX core is design-time configured by means of two different systems: the VHDL generics passed to the toplevel core entity and the configuration scripts.

VHDL generics

VHDL generics are used to configure the most important metrics of the core, such as the issue width, the degree of reconfigurability, the functional unit distribution, and complexity of the debug support system. Refer to Section 9.2.2.1 for more information about what exactly the generics control.

Because the generics are specified per instantiation of the core, it is possible to have differently configured ρ-VEX cores in a single design. This allows for heterogeneous multicore systems.

The values of the generics are represented as read-only registers in the global control register file in a generic way. The registers are designed such that future additions to the core are unlikely to require restructuring the existing registers, making them forward compatible. In addition, they are structured such that it is easy to extract information from the data, usually even by visually inspecting the hexadecimal values. The global control registers are described in detail in Section 4.1.

Configuration scripts

The 'configuration scripts' refer to a set of Python scripts residing in the `config` directory in the root of the ρ-VEX repository. When run by calling `make` in the root of the `config` directory, these scripts read a set of configuration and template files, to generate various sources in the repository. These sources vary from key VHDL sources for the ρ-VEX core, to memory map headers for `rvd` and the build system, to the LaTeX source files for this very document. The philosophy is that this not only makes it easier to change key components of the core, but that it should also stimulate developers to keep the documentation up-to-date, without the primary source for documentation needing to be comments in the VHDL sources.

The configuration scripts control the following processor features.

- Global and context control register file functionality, memory map and documentation (Section 8.1).
- Instruction set encoding and documentation, as well as assembly syntax (Section 8.2).
- Pipeline configuration of the ρ-VEX core (Section 8.3).

- Trap decoding and documentation (Section 8.4).

Each feature is controlled by a set of LaTeX-like files and/or key-value configuration files. These LaTeX-like files are not intended to be processed by anything other than the scripts — they cannot be processed by LaTeX directly. The only reason for their syntax to be derived from LaTeX is because it allows the documentation sections to be properly syntax-highlighted.

Each class of configuration files supports a set of non-standard commands that define the configuration. These commands are described in the following sections, as referenced in the list.

As stated, the configuration needs to be manually committed to the repository by calling `make` in the `config` directory. Changing the configuration files without doing this has no effect. This command also regenerates this PDF file, but it does not rebuild or test anything else. It is highly advised to run the conformance test suite in `platform/core-tests` after changing the configuration.

8.1 Control register files

The control register file configuration files reside in the `config/cregs` directory of the ρ-VEX repository. The configuration consists of a set of LaTeX-styled files, interpreted ordered alphabetically by their filenames. The configuration controls roughly the following things.

- The address of each control register, within hardcoded limits. The global register file is mapped from `0x000` to `0x100`, whereas the context control register file is mapped from `0x200` to `0x400`.
- The documentation for each control register, as it appears in Sections 4.1 and 4.2 of this manual.
- The functionality of each register, described using a special C-like language, which may be compiled to VHDL and C. The latter is intended for a cycle-accurate simulator, but this does not exist yet.
- The VHDL entity interface of the `cxreg` and `gbreg`, such that the implementations of the registers can communicate with the rest of the processor. If the interface is changed, the instantiation of `cxreg` and `gbreg` in `core.vhd` must be changed accordingly to make the connections.

The first of the following sections describes the LaTeX-style commands that are recognized by the configuration scripts. Any other commands are interpreted as being part of the LaTeX documentation sections. The remaining sections document the 'language-agnostic' mini-language used to describe the register logic implementations. This language-agnostic code can be transformed by the configuration scripts into both VHDL for the hardware and C for a simulator, although the latter is not yet utilized.

8.1.1 .tex command reference

The following LaTeX-like commands are interpreted by the Python scripts to define the control registers. They must be the only thing on a certain line aside from optional LaTeX-style comments at the end of the line, otherwise they are interpreted as part of a documentation section.

```
- \contextInterface{}, \globalInterface{}
   '- \ifaceGroup{title}
       '- \ifaceSubGroup{}
           |- \ifaceIn{unit}{name}{type}
           |- \ifaceOut{unit}{name}{type}{expr}
           |- \ifaceInCtxt{unit}{name}{type}
           '- \ifaceOutCtxt{unit}{name}{type}{expr}

- \defineTemplate{name}{parameter list}

- \register{mnemonic}{name}{offset},
  \registergen{python range}{mnemonic}{name}{offset}{stride}
   '- \field{range}{mnemonic}
       |- \reset{bit vector}
       |- \signed{}
       |- \id{identifier}
       |- \declaration{}
       |   |- \declRegister{name}{type}{expr}
       |   |- \declVariable{name}{type}{expr}
       |   '- \declConstant{name}{type}{expr}
       |
       |- \implementation{}
       |- \resetImplementation{}
       |- \finally{}
       '- \connect{output}{expr}

- \perfCounter{mnemonic}{name}{offset}
   |- \declaration{}
   |   |- \declRegister{name}{type}{expr}
   |   |- \declVariable{name}{type}{expr}
   |   '- \declConstant{name}{type}{expr}
   |
   '- \implementation{}
```

\contextInterface {}

\globalInterface {}

These commands describe the port map of the context register logic and the global register logic respectively. They may appear more than once in the configuration; their contents will simply be appended.

\ifaceGroup {<title>}

\ifaceSubGroup {}

These commands define port groups for code readability. The toplevel group has a title. Both group commands will interpret the text following the command as comments for the code.

\ifaceIn {<unit>}{<name>}{<type>}

`\ifaceOut {<unit>}{<name>}{<type>}{<expr>}`

`\ifaceInCtxt {<unit>}{<name>}{<type>}`

`\ifaceOutCtxt {<unit>}{<name>}{<type>}{<expr>}`

These commands define ports. For inputs, the signal name will be `<unit>2cxreg_<name>` or `<unit>2gbreg_<name>`. For outputs it will be `cxreg2<unit>_<name>` or `gbreg2<unit>_<name>`. `<type>` is a type specification as defined later. `<expr>` is an expression using only predefined constants (Section 8.1.4), input signals and literals. The command name determines whether the port is per-context or global and whether it is an input or output.

`\register {<mnemonic>}{<name>}{<offset>}`

This command starts a new register description. `<name>` is the title of the section. `<mnemonic>` is the mnemonic of the register, excluding the `CR_` prefix. The mnemonic must be mix of up to eight uppercase, number or underscore characters, and must be unique. The register may be referenced in LaTeX as `\creg {<mnemonic>}`; this will generate a hyperlink in the PDF to the register documentation. `<offset>` should be a hex number starting with `0x` divisible by 4, representing the byte offset from the control registers base. Global registers should be within the `0x000..0x0FF` range, context registers should be within `0x200..0x3FF`. `0x100..0x1FF` is reserved for the general purpose register file.

`\registergen {<python range>}{<mnemonic>}{<name>}{<offset>}{<stride>}`

Same as `\register` , but specifies a list of registers. `<python range>` is executed as a Python expression, expected to generate an iterable of integers. A register is generated for each of these iterations. The offset for each register is computed as `<offset>` $+iter*$ `<stride>`. `\n {}` expands to the number when used inline in `<mnemonic>` and `<name>`. In the documentation it expands to `$n$`.

`\field {<range>}{<mnemonic>}`

This command defines a field in the current register. A range specification is either a single bit index for a single-bit field, or of the form `<from>..<to>`, where `<from>` is the higher bit index, and both the `<from>` and `<to>` bits are included in the range. For example, `3..1` includes bits 1, 2 and 3. `<mnemonic>` should be a short, uppercase identifier for the field, which must be unique within the register. It should be as short as possible, in particular for single-bit fields, as it needs to fit in the layout of the documentation. It also needs to be a valid C and VHDL identifier, so for instance spaces and hyphens are not allowed.

`\reset {<bit vector>}`

This command sets the reset state of the previously defined field. If not specified, the reset state is assumed to be zero. The number of characters in `<bit vector>` must equal the number of bits in the field.

`\signed {}`

This command marks a field as being a signed number. The default is unsigned.

`\id {<identifier>}`

This command gives a field an alternative name for the C/VHDL/`rvd` definitions. This only works for 8-bit and 16-bit fields that are properly aligned.

`\declaration {}`

This command specifies the local register, variable and constant declaration section for this field implementation.

`\declRegister {<name>}{<type>}{<expr>}`
`\declVariable {<name>}{<type>}{<expr>}`
`\declConstant {<name>}{<type>}{<expr>}`
These commands specify registers, variables or constants respectively. These may be used by the implementation code. `<name>` must start with an underscore, and will expand to `cr_<register-mnemonic>_<field-mnemonic><name>`. It must be a valid C and VHDL identifier. `<type>` is a type name, as defined in Section 8.1.3. `<expr>` is an expression which defines the constant, initial value or reset value, using only predefined constants (Section 8.1.4) for a constant value, and only inputs or predefined constants for variables and registers.

`\implementation {}`
This command starts a language-agnostic code section as defined in Section 8.1.2, executed every rising clock edge with `clkEn` high and `reset` inactive. The following variables are predefined to interface with the core and debug busses for regular register fields.

- `_write`: the value being written if the corresponding `_wmask` bits are set.
- `_wmask`: write mask for each bit in the field, honoring both writes from the core directly and writes from the debug bus.
- `_wmask_dbg`: same as `_wmask`, but only honors debug bus accesses.
- `_wmask_core`: same as `_wmask`, but only honors accesses made by the core directly.
- `_read`: this must be written to in the `\implementation {}` section to specify the read value for the field.

The types of these variables are `bitvec`s with the same width as the field. As an example of how to use these, a simple register may be created as follows. This requires `_reg` to be declared using `\declRegister` as a `bitvec` of the field size.

```
_reg = (_reg & ~_wmask) | (_write & _wmask);
_read = _reg;
```

Performance counter implementations do not have these variables. They have `_add` instead. This is a `byte`-typed variable which specifies how much should be added to the performance counter in this cycle. The bus interfacing logic is generated to conform to Section 4.3.

`\resetImplementation {}`
This command starts a language-agnostic code section as defined in Section 8.1.2, executed every rising clock edge with `clkEn` high, while the global `reset` signal is inactive but the context-specific reset signal is active. This allows register implementations to override a soft context reset, for instance to make register values persistent in this case. This is necessary for, for instance, the B flag in `CR_DCR`, to allow the debugger to reset the core without immediately starting execution. This command is only allowed for context-specific registers.

\finally {}

This command starts a language-agnostic code section as defined in Section 8.1.2, executed every rising clock edge with `clkEn` high and `reset` inactive, after all `\implementation {}` sections have been processed. Variables from other fields and registers may be read in this section, in addition to all the objects which are accessible from `\implementation {}`. This allows registers to be written combinatorially from multiple field implementations, by having the regular field implementations prepare variables that describe the new value, and subsequently combining the variable values in a `\finally {}` section.

\connect {<output>}{<expr>}

This command combinatorially connects the specified output port with the specified expression. This expression may use registers and predefined constants (Section 8.1.4). This may be used to easily connect an output port to an internal register. Note however, that since inputs cannot be used in the expression, it still cannot make a combinatorial path from an input to an output. This is illegal specifically because such combinatorial paths would be needlessly difficult to model with a simulator.

\perfCounter {<mnemonic>}{<name>}{<offset>}

This command generates a performance counter register conforming to Section 4.3. The counter will occupy two 32-bit register slots from `<offset>` onwards, holding up to 7 bytes worth of counter data. The implementation expects `_add` to be set to the value which is to be added to the counter; all the bus interfacing logic is generated. The counter value register is accessible from other implementations as `CR_<mnemonic>_<mnemonic>0_r`.

8.1.2 Language-agnostic code (LAC) sections

The ‘language-agnostic code’ sections define the behavior of control registers. Language-agnostic code, or LAC, is a C-like domain-specific language developed specifically for describing registers in the ρ-VEX. The configuration scripts are capable of transforming LAC into both VHDL and C with relative ease. The latter is intended for a cycle-accurate simulator, but at the time of writing this simulator does not exist yet.

As LAC is C-like, LaTeX-style comments cannot coexist with it, due to the C `%` operator for modulo. Because of this, C-styled comments are used within the LAC sections. In order to prevent confusion with syntax-highlighting editors and ambiguity about where the section ends, LAC sections may be enclosed by `\begin {lstlisting}` and `\end {lstlisting}` tags. LaTeX syntax highlighters should disable highlighting in these blocks.

8.1.3 LAC type system

The primitive types supported by LAC and their VHDL equivalents are shown in Table 8.1. The C equivalents of the types range from `uint8_t` to `uint64_t`. The smallest available C type that the LAC/VHDL type fits in is used. Note that this means that `bitvec` and `unsigned` types of more than 64 bits are not supported.

In addition to the scalar primitives in the table, LAC also supports hardcoded aggregate types, i.e. the equivalent of a VHDL record or C struct. Rudimentary support is provided for array-typed aggregate members to be compatible with existing VHDL

Table 8.1: LAC primitive types.

LAC type name	Supported values	VHDL type name
boolean	true or false	boolean
natural	0..2147483647 (31 bit)	natural
bit	'0' or '1'	std_logic
bitvec<n>	n bits of '0' or '1'	std_logic_vector(n-1 downto 0)
unsigned<n>	n bits of '0' or '1'	unsigned(n-1 downto 0)

data structures at the time it was developed, though these arrays can only be indexed by integer literals.

In addition, objects can be instantiated per hardware context, which also results in an array. If per-context objects are used in a context control register implementation, they are implicitely indexed by the context which the register belongs to. Otherwise, a context may only be explicitely specified as an integer literal.

Arrays present a problem in VHDL code output. Of the primitive types, only bit actually has a VHDL array type. To get around this, and also to have the generated code be consistent with the human-written VHDL sources, a number of derived types are available. These are listed along with the supported aggregate types in Table 8.2.

Table 8.2: LAC derived types.

LAC	VHDL	C
byte (bitvec8)	rvex_byte_type rvex_byte_array	uint8_t
data (bitvec32)	rvex_data_type rvex_data_array	uint32_t
address (bitvec32)	rvex_address_type rvex_address_array	uint32_t
sylstatus (bitvec16)	rvex_sylStatus_type rvex_sylStatus_array	uint16_t
brregdata (bitvec8)	rvex_brRegData_type rvex_brRegData_array	uint8_t
trapcause (bitvec8)	rvex_trap_type rvex_trap_array	uint8_t
twobit (bitvec2)	rvex_2bit_type rvex_2bit_array	uint8_t
threebit (bitvec3)	rvex_3bit_type rvex_3bit_array	uint8_t
fourbit (bitvec4)	rvex_4bit_type rvex_4bit_array	uint8_t
sevenByte (bitvec56)	rvex_7byte_type rvex_7byte_array	uint64_t
trapinfo (aggregate)	trap_info_type trap_info_array	trapInfo_t
breakpointinfo (aggregate)	cxreg2pl_breakpoint_info_type cxreg2pl_breakpoint_info_array	breakpointInfo_t
cachestatus (aggregate)	rvex_cacheStatus_type rvex_cacheStatus_array	cacheStatus_t
cfgvect (aggregate)	rvex_generic_config_type -	cfgVect_t

Coercion and typecasts

Typically, LAC will take care of typing for you by coercing one type into another on the fly. If LAC does not know how to do it or a cast would be ambiguous, you can cast manually using C-style typecasts. The following rules apply.

- Conversions between `boolean` and `natural` works as they do in C. That is, `false` converts to zero and vice versa, `true` converts to one, and nonzero converts to `true`.
- Conversions between `boolean` and `bit` use positive logic. That is, `'1'` equals `true` and `'0'` equals `false`.
- `bit` and `bitvec1` are interchangeable.
- `bitvec<n>` and `unsigned<n>` are interchangeable.
- When a `bitvec<n>` is cast to a `bitvec` of different size, the vector is zero-extended or truncated.
- When a `bitvec<n>` is cast to a `natural` or vice versa, the value is zero-extended or truncated, with the `natural` behaving as a 31-bit value.

Access types

Aside from having a type that describes what kind of values are allowed for an object, LAC objects also have an 'access type'. This describes the access priviliges, scoping rules and general behavior of an object. The following access types are available.

- *Input*: represents an input port of the VHDL entity. They are available everywhere but in constant object initializers. They are read only.
- *Register output*: represents an output port of the VHDL entity, driven from the clocked process. They are write only.
- *Combinatorial output*: represents an output port of the VHDL entity, driven combinatorially using a `\connect` command. These objects may not be used in LAC sections.
- *Register*: represents a user defined register, declared using a `\declRegister` command. These may be read and written in any LAC section, regardless of where they are declared. They behave like VHDL signals; that is, when they are written to, the read value of a register is not affected until the next clock cycle.
- *Variable*: represents a user defined variable, declared using a `\declVariable` command. These may be read and written in the `\implementation` section of only the field to which they belong. Furthermore, they may be read in any `\finally` section.
- *Constant*: represents a user defined constant, declared using a `\declConstant` command. They are read only and globally scoped.

- *Predefined constant*: represents a predefined constant, such as a package constant or the `CFG` generic. They are read only and globally scoped. Section 8.1.4 lists the available predefined constants.

8.1.4 LAC predefined constants

The following predefined constants are available. These are hardcoded into the Python scripts.

- In context control register code only: `natural ctxt`. Represents the context to which the current register belongs.
- `cfgvect CFG` maps to the `CFG` generic.
- `bitvec65 RVEX_CORE_TAG` maps to the core version tag, to be stored in `CR_CTAG`.
- `natural BRANCH_OFFS_SHIFT` maps to the package constant of the same name, representing the way in which the branch offset field of instructions is encoded.
- `natural S_*` and `natural L_*` map to the pipeline stage and latency definitions defined in `core_pipeline_pkg.vhd`.

8.1.5 LAC literals

LAC supports the following literal syntaxes for literals for each primitive type.

- `boolean`: the literals for a `boolean` are `true` and `false`.
- `natural`: natural numbers can be represented in decimal, hexadecimal by prefixing `0x`, octal by prefixing `0` and binary by prefixing `0b`.
- `bit`: the literals for a `bit` are `'1'` and `'0'`.
- `bitvec`: `bitvec` literals can be represented as a binary string enclosed in double quotes, for instance `"0101"`. In addition, hexadecimal notation is supported by prefixing an `X`, for instance `X"DEADBEEF"`.
- `unsigned`: `unsigned` literals can be represented in the same way as `bitvec` literals by simply prefixing the literal with a `U`. For instance, `U"0101"` and `UX"DEADBEEF"`.

Aggregate types can be assigned and initialized using an compound literal. The syntax is similar to VHDL aggregates.

```
some_aggregate = {
  <name>                        => <expression>,
  <name>{<array index literal>} => <expression>,
  <name>{others}                => <expression>,
  others                        => <expression>
};
```

Unlike in VHDL, 'others' can assign any kind of combination of types, as long as the expression can be coerced to each member type. For instance, the aggregate others => 0 will initialize any aggregate which does not itself contain another aggregate to all zeros.

Aggregate literals can not be used anywhere in an expression like the other kinds of literals. They can only be assigned directly to an object or used as an initialization expression.

8.1.6 LAC object objects and references

All objects except for predefined constants (Section 8.1.4) need to be declared using the \iface * and \decl * commands. These objects can then be referenced as follows, for both reads and writes.

- The most basic way to reference an object is to use its full name. For inputs, this takes the form <unit>2cxreg_<name> or <unit>2gbreg_<name>, depending on whether it is part of the context or global register file interface. Likewise, outputs take the form cxreg2<unit>_<name> or gbreg2<unit>_<name>. Finally, objects declared using the \decl * commands take the form cr_<register-mnemonic>_<field-mnemonic>_<name>. The register and field mnemonics are included to make them unique within global scope.

- In the LAC sections for a certain field, objects that are declared in the same field can be referenced using just _<name>. That is, the cr_<register-mnemonic>_<field-mnemonic> part of the object name is implicit in the reference.

- To explicitly specify a context for context-specific objects, an @ symbol and a natural literal determining the context may be appended. For instance, _pc@2 references the value of _pc for context 2. Note that the context *must* be a literal, not even constant objects are permitted. The only thing that is permitted is \n {} in generated registers, as this expands to a natural literal before parsing, similar to how a C macro would behave. If the context is not explicitly specified, the current context is used. If there is no current context, the context must be explicitly specified.

- Member access for aggregate types is done by appending a . (decimal point) followed by the member name. Rudimentary support is provided for array members using {<index>}. As with explicitly specified contexts, only natural literals and \n {} (in a \registergen environment) are allowed for specifying the index.

- VHDL-like bit indexes and slices are supported for unsigned and bitvec types. Indexing a single bit is done by appending [<position>] to the end of the reference, where <position> may be any natural-typed expression. The result of the indexing operation is a bit. Slicing is done using [<position>, <length>], where <position> still accepts any natural-typed expression, but <length> only supports natural literals. This is necessary to determine the resulting type at compile time, which is an unsigned or bitvec of size <length>. <position> specifies the lower bit index of the slice range.

8.1.7 LAC operators

Table 8.3 lists all the operators that are available in LAC. Note that member access, member array indexing, the explicit context `@` symbol and slices are not considered to be operators, but parts of a reference.

LAC operator precedence is identical to C operator precedence. As can be expected, parentheses can be used to explicitly specify evaluation order.

Table 8.3: LAC operators.

Prec.	**Op.**	**Description**	**Assoc.**

1	`!`
	`~`
	`(type)`
		Logical complement
		One's complement
		Type cast
			Right-to-left

2	`*`
	`/`
	`%`
		Multiplication
		Division
		Modulo
			Left-to-right

3	`+`
	`-`
	`$`
		Addition
		Subtraction
		`bitvec`/`unsigned` concatenation
			Left-to-right

4	« »	Left shift Right shift	Left-to-right

5	`<`
	`<=`
	`>`
	`>=`
		Less than
		Less than or equal to
		Greater than
		Greater than or equal to
			Left-to-right

6	`==`
	`!=`
		Equality
		Inequality
			Left-to-right

7	`&`	Bitwise and	Left-to-right
8	ˆ	Bitwise xor	Left-to-right
9	`|`	Bitwise or	Left-to-right
10	`&&`	Boolean and	Left-to-right
11	ˆˆ	Boolean xor	Left-to-right
12	`||`	Boolean or	Left-to-right

8.1.8 LAC statements

LAC only supports the following statements.

`<reference> = <expression>;`
Assignment statement.

`if (<expression>) <statement>`
Conditional statement without `else`.

`if (<expression>) <statement> else <statement>`
Conditional statement with `else`.

`{ <statement*> }`
C-style block statement.

`<?vhdl ... ?>`

`<?c ... ?>`
Verbatim block statements. Anything written in place of the ellipsis is in principle outputted straight to the VHDL or C output. This allows the usage of constructs unknown to LAC. Even in these sections however, it is possible to have the code generator convert LAC-style references to the target language. This is particularly useful for C output, where the LAC objects are part of special data structures. The syntaxes for such converted references are as follows.

```
@read <name>
@read <name>@<context>
@lvalue <name>
@lvalue <name>@<context>
```

In addition to being convenient syntactic sugar, the LAC generator keeps track of which objects are read from and written to. Not using this syntax may result in incorrect optimizations.

8.2 Instruction set

The instruction set configuration files reside in the `config/opcodes` directory of the ρ-VEX repository. The configuration consists of a set of LaTeX-styled files, interpreted ordered alphabetically by their filenames, and a single key-value configuration file (`encoding.ini`), containing miscellaneous information for the instruction decoder. The complete configuration controls roughly the following things.

- The opcode for each syllable.
- The functionality of each syllable, by means of specifying the control signals. The functional units themselves are *not* controlled by this configuration.
- The assembly syntax for each syllable.
- Documentation for each syllable, as it appears in Section 3.7.
- The encoding of the branch offset field.

The next section describes the structure of the LaTeX-style configuration files, and the subsequent section provides a command reference. The last section describes the key-value configuration file.

8.2.1 `.tex` file structure

Every description of a syllable/opcode starts with a `\syllable` command. Any unrecognized commands or textual lines following a `\syllable` command are considered to be LaTeX documentation for the syllable. To provide structure among the many instructions, `\section` commands are used to group syllables. There is only one level of hierarchy this way (i.e. there is no `\subsection` etc.), and it must be used. That is, `\syllable` commands before the first `\section` command are illegal. Any unrecognized command or text between a `\section` and `\syllable` command is considered to be LaTeX documentation for the syllable group.

All commands other than `\section` and `\syllable` specify attributes for the syllables. These are used to describe the characteristics and functionality of the instructions. These may be placed anywhere in the configuration files; their position relative to the `\section` and `\syllable` commands determine to which syllables they apply.

- Attribute commands placed before the first `\section` commands apply to all syllables. They can be thought of as being the default attributes. All syllable attributes *must* have a default value.
- Attribute commands placed between a `\section` and `\syllable` command apply to all syllables in the group, overriding the global defaults.
- Attribute commands placed after a `\syllable` command apply to that syllable, overriding the global and group defaults.

8.2.2 .tex command reference

The following LaTeX-like commands are interpreted by the Python scripts to define the instruction set. They must be the only thing on a certain line aside from optional LaTeX-style comments at the end of the line, otherwise they are interpreted as part of a documentation section.

`\section {<name>}`
This command starts a group of syllable definitions. `<name>` will appear as a section header in the documentation. Any unrecognized command or text between `\section` and the first `\syllable` command is interpreted as LaTeX documentation for the group.

`\syllable {<opcode>}{<mnemonic>}{<syntax>}`
This command starts the definition of a syllable.

- `<opcode>` should be a 9-bit binary string, used by the hardware to identify the syllable. Dashes may be used for don't cares. The 9 bits map to syllable bit 31..23. The value of the LSB is not really part of the opcode (the opcode field is only 8 bits wide), but defines whether the instruction can be used with only a register for the second operand (0), only an immediate for the second operand (1), or both (-). This bit is referred to as the `imm_sw` (immediate switch) bit.
- `<mnemonic>` is the name of the syllable. It will be made lowercase for the assembler syntax and uppercase for the documentation.
- `<syntax>` describes the assembler syntax of the syllable. In this, the LaTeX-like commands from Table 8.4 may be used inline. The `{}` may be omitted here.

Any textual lines between `\syllable` and `\section` or the next `\syllable` is interpreted as LaTeX documentation for the syllable. These text sections may also use the inline commands from Table 8.4, but here the `{}` may *not* be omitted.

Table 8.4: Mapping commands from assembly syntax to instruction encoding.

Command	**In docs.**	**Description**	**Encoding**	**Condition**
`\rd {}`	`$r0.d`	Integer destination register	Bit 22..17	–
`\rx {}`	`$r0.x`	Integer operand 1 register	Bit 16..11	–
`\ry {}`	`$r0.y` `imm`	Integer operand 2 register Integer immediate	Bit 10..5 Bit 10..2	`imm_sw` = 0 `imm_sw` = 1
`\rs {}`	`$r0.1`	Stack pointer	–	–
`\bd {}`	`$r0.bd`	Branch destination register	Bit 19..17 Bit 4..2	`brFmt` = 0 `brFmt` = 1
`\bs {}`	`$r0.bs`	Branch operand register	Bit 4..2 Bit 26..24	`brFmt` = 0 `brFmt` = 1
`\lr {}`	`$l0.0`	Link register	–	–
`\of {}`	`offs`	Branch offset immediate	Bit 23..5	–
`\sa {}`	`stackadj`	Stack adjustment immediate	Bit 23..5	–
`\lt {}`	`tgt`	Long immediate lane target	Bit 27..25	–
`\li {}`	`imm`	Long immediate	Bit 24..2	–

`\class {<name>}`

This command specifies the resource class. `<name>` must be `ALU`, `MEM`, `MUL`, `BR` or `LIMMH`.

`\datapath {<key>}{<value>}`

`\alu {<key>}{<value>}`

`\branch {<key>}{<value>}`

`\memory {<key>}{<value>}`

`\multiplier {<key>}{<value>}`

These commands specify the control signals for the syllable. Which keys and values are recognized depend on the VHDL code in `core_opcode*_pkg.vhd`. They are documented in the comments of the code. Note that the configuration scripts do *not* perform error checking. Instruction set configuration errors thus will not appear until the VHDL code is compiled. Of course, if you are making changes here, you should test the core anyway.

`\noasm {}`

This attribute specifies that this syllable cannot appear in user-written assembly code. This is the case for LIMMH instructions, which are inferred by the assembler.

8.2.3 `encoding.ini` reference

This file currently defines only a single value. It determines which encoding is used for the relative branch offset, and may be set to 2 or 3. When set to 3, the LSB of the branch offset has a weight of to 64 bits. When set to 2, the branch offset is shifted right by one bit, to allow branching to 32 bit boundaries. The syntax of the file is shown below.

```
[encoding]
branch_offset_shift = 3
```

8.3 Pipeline

The pipeline configuration consists of a single key-value file in the `config/pipeline` directory of the ρ-VEX repository. The configuration describes how the ρ-VEX pipeline is organized by specifying the first stage and latencies of a multitude of configurable blocks, such as register read and writeback, branch unit determination, PC+1 computation, etc. The file is self-documenting by means of its comments.

8.4 Traps

The trap configuration files reside in the `config/traps` directory of the ρ-VEX repository. The configuration consists of a set of LaTeX-styled files, interpreted ordered alphabetically by their filenames. The configuration controls roughly the following things.

- The trap names and numeric identifiers.
- Decoding signals for each trap; debug and interrupt traps are handled differently by the processor.
- A pretty-printing macro for each trap.
- Documentation for each trap, as it appears in Section 5.3.

The next section describes the structure of the LaTeX-style configuration files. The subsequent section provides a command reference.

8.4.1 `.tex` file structure

The `\trap` and `\trapgen` commands start the definition of a trap or a number of similar traps respectively. Any unrecognized command or text following such a command is interpreted as being LaTeX documentation for the trap. The obligatory `\description` command defines the formatting string used to pretty-print the trap information. The remaining commands are optional decoding attributes for the traps.

8.4.2 `.tex` command reference

The following LaTeX-like commands are interpreted by the Python scripts to define the traps. They must be the only thing on a certain line aside from optional LaTeX-style comments at the end of the line, otherwise they are interpreted as part of a documentation section.

`\trap {<index>}{<mnemonic>}{<name>}`
The command starts a trap description. `<index>` is the trap index, which may range from 1 to 255. `<mnemonic>` is the trap identifier, which must be a valid C and VHDL identifier and should be uppercase. It is prefixed with `RVEX_TRAP_` in the header files. `<name>` is the LaTeX-formatted friendly name of the trap, used as the section title in the documentation.

`\trapgen {<python range>}{<start index>}{<mnemonic>}{<name>}`
This command works the same as `\trap` , but specifies a list of traps. `<python range>` is executed as a Python expression, expected to generate an iterable of integers. A trap specification is generated for each of these iterations. The index for each trap is computed as `<offset>`$+iter$. `\n {}` expands to the iterator value when used inline in `<mnemonic>` and `<name>`, as well as in `\description {<desc>}` below. In the documentation it expands to `$n$`.

`\description {<desc>}`
This command defines a formatting string used to pretty-print the trap information. It is used by the debug systems to allow the user to quickly identify the trap. In this description, the following commands may be used inline.

- `\at {}` expands to " `at <trap point>`" if the trap point is known, or to nothing if the trap point is not known. The trap point is expressed in hexadecimal notation.
- `\arg {u}` expands to the trap argument in unsigned decimal notation.
- `\arg {s}` expands to the trap argument in signed decimal notation.
- `\arg {x}` expands to the trap argument in hexadecimal notation.

`\debug {}`
Marks that this trap is a debug trap. The `{}` is required.

`\interrupt {}`
Marks that this trap is an interrupt trap. The `{}` is required.

9 Instantiation

This section describes how the ρ-VEX core and processing systems should be instantiated, what the functions of all the external signals are, and what generics are available. The first section lists the basic signal data types that will be used throughout the interfaces. The remaining sections document instantiation of the bare ρ-VEX processor core and two processing systems that incorporate the processor and local memories or cache, one that does not depend on GRLIB and one which does.

9.1 Data types

The following basic VHDL data types are used for the ports and generics. They are defined in `common_pkg`.

```
subtype rvex_address_type     is std_logic_vector(31 downto  0);
subtype rvex_data_type        is std_logic_vector(31 downto  0);
subtype rvex_mask_type        is std_logic_vector( 3 downto  0);
subtype rvex_syllable_type    is std_logic_vector(31 downto  0);
subtype rvex_byte_type        is std_logic_vector( 7 downto  0);

type rvex_address_array       is array (natural range <>) of rvex_address_type;
type rvex_data_array          is array (natural range <>) of rvex_data_type;
type rvex_mask_array          is array (natural range <>) of rvex_mask_type;
type rvex_syllable_array      is array (natural range <>) of rvex_syllable_type;
type rvex_byte_array          is array (natural range <>) of rvex_byte_type;
```

The `address`, `data` and `syllable` types all represent 32-bit words. The distinction is made only for clarity; one can not simply give the ρ-VEX processor 64-bit address map by widening the address type.

The `mask` type is used for byte-masking the data vectors for bus operations. As all memory operations operate on 32-bit words, the `mask` type has four bits to mask each byte. The most significant bit of the these masks maps to the most significant byte of the 32-bit word, and thus to the lowest byte address, as the ρ-VEX system is big endian.

The `byte` type should be self-explanatory.

9.2 Bare ρ-VEX processor

This section describes how the bare ρ-VEX core should be instantiated. It is intended for HDL designers who wish to design their own processing system.

9.2.1 Instantiation template

The following listing serves as an instantiation template for the core. The code is documented in the following sections.

If you get errors when instantiating the core with this template, the documentation might be out of date. Fear not, for the signals are also documented in the entity description in `core.vhdl`.

```vhdl
library rvex;
use rvex.common_pkg.all;
use rvex.core_pkg.all;

-- ...

rvex_inst: entity rvex.core
  generic map (

    -- Core configuration.
    CFG => rvex_cfg(
      numLanesLog2                => 3,
      numLaneGroupsLog2           => 2,
      numContextsLog2             => 2
      -- ...
    ),
    CORE_ID                       => CORE_ID,
    PLATFORM_TAG                  => PLATFORM_TAG

  )
  port map (

    -- System control.
    reset                         => reset,
    resetOut                      => resetOut,
    clk                           => clk,
    clkEn                         => clkEn,

    -- Run control interface.
    rctrl2rv_irq                  => rctrl2rv_irq,
    rctrl2rv_irqID                => rctrl2rv_irqID,
    rv2rctrl_irqAck               => rv2rctrl_irqAck,
    rctrl2rv_run                  => rctrl2rv_run,
    rv2rctrl_idle                 => rv2rctrl_idle,
    rctrl2rv_reset                => rctrl2rv_reset,
    rctrl2rv_resetVect            => rctrl2rv_resetVect,
    rv2rctrl_done                 => rv2rctrl_done,

    -- Common memory interface.
    rv2mem_decouple               => rv2mem_decouple,
    mem2rv_blockReconfig          => mem2rv_blockReconfig,
    mem2rv_stallIn                => mem2rv_stallIn,
    rv2mem_stallOut               => rv2mem_stallOut,
    mem2rv_cacheStatus            => mem2rv_cacheStatus,

    -- Instruction memory interface.
    rv2imem_PCs                   => rv2imem_PCs,
    rv2imem_fetch               => rv2imem_fetch,
    rv2imem_cancel              => rv2imem_cancel,
    imem2rv_instr               => imem2rv_instr,
    imem2rv_affinity            => imem2rv_affinity,
    imem2rv_busFault            => imem2rv_busFault,

    -- Data memory interface.
    rv2dmem_addr                => rv2dmem_addr,
    rv2dmem_readEnable          => rv2dmem_readEnable,
    rv2dmem_writeData           => rv2dmem_writeData,
    rv2dmem_writeMask           => rv2dmem_writeMask,
    rv2dmem_writeEnable         => rv2dmem_writeEnable,
    dmem2rv_readData            => dmem2rv_readData,
    dmem2rv_ifaceFault          => dmem2rv_ifaceFault,
    dmem2rv_busFault            => dmem2rv_busFault,

    -- Control/debug bus interface.
    dbg2rv_addr                 => dbg2rv_addr,
    dbg2rv_readEnable           => dbg2rv_readEnable,
    dbg2rv_writeEnable          => dbg2rv_writeEnable,
    dbg2rv_writeMask            => dbg2rv_writeMask,
    dbg2rv_writeData            => dbg2rv_writeData,
    rv2dbg_readData             => rv2dbg_readData,

    -- Trace interface.
    rv2trsink_push              => rv2trsink_push,
    rv2trsink_data              => rv2trsink_data,
    rv2trsink_end               => rv2trsink_end,
    trsink2rv_busy              => trsink2rv_busy

  );
```

9.2.2 Interface description

As you can see in the template, the generics and signals are grouped by their function. The following subsections will document each group.

9.2.2.1 Core configuration

These generics parameterize the core.

- `CFG : rvex_generic_config_type`

  This generic contains the configuration parameters for the core. `rvex_generic_config_type` is a `record` type with the following members.

  - `numLanesLog2 : natural`

    This parameter specifies the binary logarithm of the number of lanes to instantiate. The range of acceptable values is 0 through 4, although only 1, 2 and 3 are tested. The default is 3, which specifies an 8-way ρ-VEX processor.

  - `numLaneGroupsLog2 : natural`

This parameter specifies the binary logarithm of the number of lane groups to instantiate. Each lane group can be disabled individually to save power, operate on its own, or work together on a single thread with other lane groups. May not be greater than 3 (due to configuration register size limits) or `numLanesLog2`. It is only tested up to `numLanesLog2`-1. The default is 2, specifying 4 lane groups.

- `numContextsLog2 : natural`

  This parameter specifies the binary logarithm of the number of hardware contexts in the core. May not be greater than 3 due to configuration register size limits. The default is 2, specifying 4 hardware contexts.

- `genBundleSizeLog2 : natural`

  This parameter specifies the binary logarithm of the number of syllables in a generic binary bundle. When a branch address is not aligned to this and `limmhFromPreviousPair` is set, then special actions will be taken to ensure that the relevant syllables preceding the trap point are fetched before operation resumes. The default is 3, specifying 8-way generic binary bundles.

- `bundleAlignLog2 : natural`

  The ρ-VEX processor will assume (and enforce) that the start addresses of bundles are aligned to the specified amount of syllables. When this is less than `numLanesLog2`, the stop bit system is enabled. The value may not be greater than `numLanesLog2`. The default is 3, disabling the stop bit system.

- `multiplierLanes : natural`

  This parameter defines what lanes have a multiplier. Bit 0 of this number maps to the first lane, bit 1 to the second lane, etc. The default is `0xFF`, specifying that each lane has a multiplier.

- `memLaneRevIndex : natural`

  This parameter specifies the lane index for the memory unit, counting down from the last lane in each lane group. So `memLaneRevIndex` = 0 results in the memory unit being in the last lane in each group, `memLaneRevIndex` = 1 results in it being in the second to last lane, etc. The default is 1.

- `numBreakpoints : natural`

  This parameter specifies how many hardware breakpoints are instantiated. The maximum is 4 due to the register map only having space for 4. The default is also 4.

- `forwarding : boolean`

  This parameter specifies whether or not register forwarding logic should be instantiated. With forwarding disabled, the core will use less area and might run at higher frequencies, but much more NOPs are necessary between data-dependent instructions. The forwarding logic is enabled by default.

- `limmhFromNeighbor : boolean`

  When this parameter is true, syllables can borrow long immediates from the neighboring syllable in a syllable pair. This is enabled by default.

- `limmhFromPreviousPair : boolean`

  When this parameter is true, syllables can borrow long immediates from the previous syllable pair. This is enabled by default. This is not supported when stop bits are enabled, i.e. when `bundleAlignLog2` < `numLanesLog2`. Therefore, when stop bits are enabled, this should be disabled.

- `reg63isLink : boolean`

  When this parameter is true, general purpose register 63 maps directly to the link register. When false, `MOVTL`, `MOVFL`, `STW` and `LDW` must be used to access the link register, but an additional general purpose register is available. This exists for compatibility with the ST200 series processors. It is disabled by default.

- `cregStartAddress : rvex_address_type`

  This paramater specifies the start address of the 1kiB control register file as seen from the processor. It must be aligned to a 1kiB boundary. The core is not able to access data memory in the specified region. The default value is `0xFFFFFC00`, i.e. the block from `0xFFFFFC00` to `0xFFFFFFFF`.

- `resetVectors : rvex_address_array(7 downto 0)`

  This parameter specifies the reset address for each context, if not overruled at runtime by connecting the optional `rctrl2rv_resetVect` signal. When less than eight contexts are instantiated, the higher indexed values are unused. The default is 0 for all contexts.

- `unifiedStall : boolean`

  When this parameter is true, the stall signals for each group will be connected to the same signal. That is, if one lane group has to stall, all lane groups necessarily have to stall. This may be a requirement of the memory subsystem connected to the core; when this is enabled, the memory architecture can be made simpler, but cannot make use of the possible performance gain due to being able to stall only part of the core. This parameter is disabled by default, meaning that the stall signals are independent.

- `gpRegImpl : natural`

  This parameter specifies the general purpose register implementation to use. The following values are accepted.

  * `RVEX_GPREG_IMPL_MEM` (default): block RAM + LVT implementation for FPGAs.
  * `RVEX_GPREG_IMPL_SIMPLE`: behavioral implementation for Synopsis.

- `traceEnable : boolean`

  This parameter specifies whether the trace unit should be instantiated. It is disabled by default.

- `perfCountSize : natural`

  This parameter specifies the size of the performance counters in bytes. Up to 7 bytes are supported. The default is 4 bytes.

  - cachePerfCountEnable : boolean

    This parameter enables or disables the cache performance counters. When enabled, the number of lane groups must equal the number of contexts, because the signals from the cache blocks are mapped to the contexts directly. In the future, the cache performance counters are to be placed in the cache instead of the core. This parameter is off by default.

Typically, one will want to use the rvex_cfg function to specify this value. This function takes as its arguments values for all the record members as specified above, but has default values for each of them, meaning that not all of them have to be specified. In addition, a base argument of type rvex_generic_config_type may be specified, which will be used as the default value for unspecified parameters. This permits mutation of the CFG record as it passes from entity to subentity, which is otherwise impossible to do with record generics.

- CORE_ID : natural

  This value is used to uniquely identify this core within a multicore platform. It is made available to the programs running on the core and the debug system through CR_COID.

- PLATFORM_TAG : std_logic_vector(55 downto 0)

  This value is to uniquely identify the platform as a whole. It is intended that this value be generated by the toolchain by hashing the source files and synthesis options. It is made available to the programs running on the core and the debug system through CR_PTAG.

9.2.2.2 System control

The system control signals include the clock source for the core, a synchronous reset signal and a global clock enable signal. clk and reset are required std_logic input signals. clkEn is an optional std_logic input signal.

The core is clocked on the rising edge of clk while clkEn is high. When a rising edge on clk occurs while reset is high, most components of the core will be reset, regardless of the state of clkEn. The only component of the core that is not reset by this is the general purpose register file. This is because this register file is implemented using block RAMs, which have no physical reset input in Xilinx FPGAs.

The resetOut signal is asserted high for one cycle when the debug bus writes a one to the reset bit in CR_GSR. This signal may be used to reset support systems as well as the core, or it may be ignored.

9.2.2.3 Run control

The run control signals provide an interface between the core and an interrupt controller or a master processor if the ρ-VEX is used as a coprocessor. All signals are optional. All signals are arrays of some sort, indexed by hardware context IDs in descending order.

- `rctrl2rv_irq : in std_logic_vector(`*number of contexts - 1* `downto 0)`
- `rctrl2rv_irqID : in rvex_address_array(`*number of contexts - 1* `downto 0)`
- `rv2rctrl_irqAck : out std_logic_vector(`*number of contexts - 1* `downto 0)`

  When `rctrl2rv_irq` is high, an interrupt trap will be generated within the indexed context as soon as possible, if the interrupt enable flag in the context control register is set. Interrupt entry is acknowledged by `rv2rctrl_irqAck` being asserted high for one `clkEnabled` cycle. `rctrl2rv_irqID` is sampled in exactly that cycle and is made available to the trap handler through the trap argument register. When not specified, `rctrl2rv_irq` is tied to `'0'` and `rctrl2rv_irqID` is tied to `X"00000000"`.

  When `rv2rctrl_irqAck` is high, an interrupt controller would typically release `rctrl2rv_irq` and set `rctrl2rv_irqID` to a value signalling that no interrupt is active on the subsequent clock edge. Alternatively, if more interrupts are pending, `rctrl2rv_irq` may remain high and `rctrl2rv_irqID` may be set to the code identifying the next interrupt.

  Releasing `rctrl2rv_irq` before an interrupt is acknowledged may still cause an interrupt trap to be caused. This is due to the fact that traps take time to propagate through the pipeline. The core will still assert `rv2rctrl_irqAck` upon entry of the trap service routine in this case. In order to properly account for this behavior, interrupt controllers should ignore `rv2rctrl_irqAck` if no interrupt is active, and there should be a special `rctrl2rv_irqID` value that signals ‘no interrupt’. The trap service routine should return to application code as soon as possible in this case.

- `rctrl2rv_run : in std_logic_vector(`*number of contexts - 1* `downto 0)`
- `rv2rctrl_idle : out std_logic_vector(`*number of contexts - 1* `downto 0)`

  When `rctrl2rv_run` is asserted low, the indexed context will stop executing instructions as soon as possible. It will finish instructions that were already in the pipeline and have already committed data, and set the program counter to point to the next instruction that should be issued for the program to resume correctly later. As soon as `rctrl2rv_run` is asserted high again, the context will resume, assuming there is nothing else preventing it from running. When `rctrl2rv_run` is not specified, it is tied to `'1'`.

  Only when the context has completely stopped, i.e., there are no instructions in the pipeline, will `rv2rctrl_idle` be asserted high. This may also happen while `rctrl2rv_run` is high, when the core is being halted for a different reason. Such reasons include preparing for reconfiguration, the context not having lane groups assigned to it, and the B flag in `CR_DCR`. `rv2rctrl_idle` remains high until the next instruction is fetched.

- `rctrl2rv_reset : in std_logic_vector(`*number of contexts - 1* `downto 0)`
- `rctrl2rv_resetVect : in rvex_address_array(`*number of contexts - 1* `downto 0)`

- `rv2rctrl_done : out std_logic_vector(`*number of contexts* `- 1 downto 0)`

  When `rctrl2rv_reset` is asserted high, the context control registers for the indexed context are synchronously reset in the next `clkEn`abled cycle. Note that this behavior is different from the master `reset` signal, which ignores `clkEn`. When it is not specified, it is tied to `'0'`.

  `rctrl2rv_resetVect` determines the reset vector for each context, i.e. the initial program counter. When it is not specified, it is tied to the reset vector specified by the `CFG` generic.

  `rv2rctrl_done` is connected to the D flag in `CR_DCR`, which is set when the processor executes a `STOP` instruction. The only way to clear this signal without debug bus accesses is to assert `reset` or `rctrl2rv_reset`.

  When the ρ-VEX is running as a co-processor, `rctrl2rv_reset` could be used as an active low flag indicating that the currently loaded kernel needs to be executed, in which case `rv2rctrl_done` signals completion. `rctrl2rv_resetVect` marks the entry point for the kernel.

9.2.2.4 Common memory interface

These control signals are common to both the data and instruction memory interface.

- `rv2mem_decouple : out std_logic_vector(`*number of lane groups* `- 1 downto 0)`

  This vector represents the current runtime configuration of the core. In particular, it specifies which lane groups are working together to execute code within a single context. When a bit in this vector is high, the indexed lane group is 'decoupled' from the next lane group, i.e., is operating within a different context. When a bit is low, the indexed lane group is working as a slave to the next higher indexed lane group for which the bit is set.

  Due to constraints in the core, the indices of pipelane groups working together are always aligned to the number of pipelane groups in the group. As an example, if pipelane groups 0 and 1 are working together, group 2 cannot join them without group 3 also joining them. This allows binary tree structures to be used in the coupling logic. This means that, in the default core configuration, only the following decouple vectors are legal: `"1111"`, `"1110"`, `"1011"`, `"1010"` and `"1000"`.

  The state of the `rv2mem_decouple` signal has several implications on the behavior of the memory ports on the ρ-VEX.

  - The PCs presented by the instruction memory ports will always be contiguous and aligned for groups that are working together. The fetch and cancel signals will always be equal.
  - The ρ-VEX assumes that the `mem2rv_blockReconfig` and `mem2rv_stallIn` signals are equal for coupled pipelane groups. Behavior is completely undefined if these assumptions are violated.

- `mem2rv_blockReconfig : in std_logic_vector(`*number of lane groups - 1* `downto 0)`

  This signal can be used by the memories to block reconfiguration due to ongoing operations. When a bit in this vector is high, the context associated with the indexed group is guaranteed to not reconfigure. The ρ-VEX will assume that the associated bits in the `mem2rv_blockReconfig` signal will always be released eventually when no operations are requested by those pipelane groups, otherwise the system may deadlock. When pipelane groups are coupled, their respective `mem2rv_blockReconfig` signals must be equal. When this signal is not specified, it is tied to all zeros.

- `mem2rv_stallIn : in std_logic_vector(`*number of lane groups - 1* `downto 0)`

  Stall input signals for each pipelane group. When the stall signal for a pipelane group is high, the next rising edge of the clock signal will be ignored. When pipelane groups are coupled, their respective `mem2rv_stallIn` signals must be equal. When this signal is not specified, it is tied to all zeros.

- `rv2mem_stallOut : out std_logic_vector(`*number of lane groups - 1* `downto 0)`

  Stall output signals for each pipelane group. This serves as a combined stall signal from all stall sources, indicating whether a pipelane group is actually stalled or not. When `rv2mem_stallOut` is high, all memory request signals from the associated pipelane group should be considered to be undefined. Memory access requests should thus be initiated (and registered) only at the rising edge of the `clk` signal when `clkEn` is high and the associated `rv2mem_stallOut` signal is low. In addition, the result of a previously requested memory operation should remain valid until the next `clkEnabled` cycle where the `rv2mem_stallOut` signal is low, as this is when the core will sample the signal.

  When pipelane groups are coupled, their respective `rv2mem_stallOut` signals will be equal. In addition, the `unifiedStall` configuration parameter in the `CFG` record may be set to `true` to enforce equal stall signals for all pipelane groups at all times, should this be desirable for the memory implementation.

- `mem2rv_cacheStatus : in rvex_cacheStatus_array(`*number of lane groups - 1* `downto 0)`

  This signal may be driven with cache status information. This is used by the trace unit only. The data type is a record defined in `core_pkg` as follows.

```
type rvex_cacheStatus_type is record
  instr_access      : std_logic;
  instr_miss        : std_logic;
  data_accessType   : std_logic_vector(1 downto 0);
  data_bypass       : std_logic;
  data_miss         : std_logic;
  data_writePending : std_logic;
end record;
```

All signals must be externally gated by the stall signals of the core for compatibility with performance counters in the future. Otherwise, the `instr_` prefixed signals share the timing of the instruction fetch result, and `data_` prefixed signals share the timing of the data memory access result.

`instr_access` should be high when an instruction fetch was performed. In this case, `instr_miss` may also be high to signal that the fetch caused a cache miss.

`data_access` should be set to `01` if a read access was performed, to `10` if a 32-bit write access was performed and to `11` if a partial write was performed. `00` logically means no operation. If an access was performed that bypassed the cache, `data_bypass` should be set. If an access was performed that caused a cache miss, `data_miss` should be set. If an access was performed by a cache block that had a nonempty write buffer when the request was made, `data_writePending` should be set.

9.2.2.5 Instruction memory interface

These signals interface between the ρ-VEX and the instruction memory or cache. All signals in this section are clock gated by not only `clkEn`, but also by the respective signal in `rv2mem_stallOut`. They should be considered to be invalid when the respective `rv2mem_stallOut` signal is high. The number of enabled clock cycles without stalls after which the reply for a request is assumed to be valid is defined by `L_IF`, which is defined in `core_pipeline_pkg`. `L_IF` defaults to 1.

- `rv2imem_PCs : out rvex_address_array(`*number of lane groups - 1* `downto 0)`

  Program counter outputs for each lane group. These will always be aligned to the size of an instruction for a full lane group. When lane groups are coupled, the PC for the first lane group will always be aligned to the size of the instruction to be executed on the set of lane groups, and the PCs for those lane groups will be contiguous.

- `rv2imem_fetch : out std_logic_vector(`*number of lane groups - 1* `downto 0)`

  Read enable output. When high, the instruction memory should supply the instructions pointed to by `rv2imem_PCs` on `imem2rv_instr` after `L_IF` processor cycles.

- `rv2imem_cancel : out std_logic_vector(`*number of lane groups - 1* `downto 0)`

  Cancel signal. This signal will go high combinatorially (regardless of the stall input from the memory) when it has been determined that the result of the most recently requested instruction fetch will not be used. In this case, the memory may cancel the request in order to be able to release the stall signal earlier. This signal can safely be ignored for correct operation.

- `imem2rv_instr : in rvex_syllable_array(`*number of* lanes *- 1* `downto 0)`

  Syllable input for each lane. Expected to be valid `L_IF` processor cycles after `rv2imem_fetch` is asserted if `rv2imem_cancel` and `imem2rv_fault` are low.

- `imem2rv_affinity : in std_logic_vector(`$n \log(n)$ `-` *1* `downto 0)`
  Where n = *number of lane groups*

  Optional block affinity input signal for reconfigurable caches. If used, it is expected to have the same timing as the `imem2rv_instr` signal. Each lane group has log(*number of lane groups*) bits in this signal, forming an unsigned integer that indexes the lane group that serviced the instruction read. When the processor wants to reconfigure, it may use this signal as a hint to determine which program should be placed on which lane group next, assuming that there will be fewer cache misses if the currently running application is mapped to the lane group indexed by the affinity signal. Its value is made available to the program using the `CR_AFF` register.

- `imem2rv_busFault : in std_logic_vector(`*number of lane groups* `-` *1* `downto 0)`

  Instruction fetch bus fault input signal. Expected to have the same timing as the `imem2rv_instr` signal. When high, a `TRAP_FETCH_FAULT` trap is generated and the instruction defined by `imem2rv_instr` will not be executed.

9.2.2.6 Data memory interface

These signals interface between the ρ-VEX and the data memory or cache. All signals in this section are clock gated by not only `clkEn`, but also by the respective signal in `rv2mem_stallOut`. They should be considered to be invalid when the respective `rv2mem_stallOut` signal is high. The number of enabled clock cycles after which the reply for a request is assumed to be valid is defined by `L_MEM`, which is defined in `core_pipeline_pkg`. `L_MEM` defaults to 1.

- `rv2dmem_addr : out rvex_address_array(`*number of lane groups* `-` *1* `downto 0)`

  Memory address that is to be accessed if `rv2dmem_readEnable` or `rv2dmem_writeEnable` is high. The two least significant bits of the address will always be `"00"` and may be ignored. Note that a configurable 1 kiB block within this 4 GiB memory space is inaccessible, because it is replaced by the core control registers. This is configurable through the `cregStartAddress` entry in `CFG`, which defaults to 0xFFFFFC00, meaning that addresses 0xFFFFFC00 through 0xFFFFFFFF are inaccessible.

- `rv2dmem_readEnable : out std_logic_vector(`*number of lane groups* `-` *1* `downto 0)`

  Active high read enable signal from the core for each memory unit. When high during an enabled rising clock edge, the ρ-VEX expects the access result to be valid `L_MEM` enabled cycles later.

- `rv2dmem_writeData : out rvex_data_array(`*number of lane groups* `-` *1* `downto 0)`

- `rv2dmem_writeMask : out rvex_mask_array(`*number of lane groups* `-` *1* `downto 0)`

  These signals define the write operation to be performed when `rv2dmem_writeEnable` is high. `rv2dmem_writeMask` contains a bit for each byte in `rv2dmem_writeData`, which determines whether the byte should be written or not: when high, the respective byte should be written; when low, the byte should not be affected. Mask bit i governs data bits $i*8+7$ downto $i*8$. This corresponds to byte address $a+3-i$, where a is the word address specified by `rv2dmem_addr`, because the ρ-VEX is big endian.

- `rv2dmem_writeEnable : out std_logic_vector(`*number of lane groups - 1* `downto 0)`

  Active high write enable signal from the core for each memory unit. When high during an enabled rising clock edge, the ρ-VEX expects either that the write request defined by `rv2dmem_addr`, `rv2dmem_writeData` and `rv2dmem_writeMask` will be performed, or that `dmem2rv_ifaceFault` or `dmem2rv_busFault` is asserted high `L_MEM` cycles later.

- `dmem2rv_readData : in rvex_data_array(`*number of lane groups - 1* `downto 0)`

  This is expected to contain the read data for read requested by `rv2dmem_readEnable` and `rv2dmem_addr` `L_MEM` enabled cycles earlier, unless `dmem2rv_ifaceFault` or `dmem2rv_busFault` are high.

- `dmem2rv_ifaceFault : in std_logic_vector(`*number of lane groups - 1* `downto 0)`

  These signals are expected to be valid `L_MEM` enabled cycles after a read or write request. `dmem2rv_ifaceFault` being high indicates that the read or write could not be performed because the memory system is incapable of servicing the specific type of memory access. For instance, the reconfigurable cache asserts this signal if more than one request is made at a time by coupled lane groups. `dmem2rv_busFault` being high indicates that some kind of bus fault occured, for example if a memory access was made to unmapped memory.

  In either case, a `DMEM_FAULT` trap will be issued. The trap argument will be set to the address that was requested.

9.2.2.7 Debug bus interface

The debug bus provides an optional slave bus interface capable of accessing most of the registers within the core.

- `dbg2rv_addr : in rvex_address_type`

- `dbg2rv_readEnable : in std_logic`

- `dbg2rv_writeEnable : in std_logic`

- `dbg2rv_writeMask : in rvex_mask_type`

- `dbg2rv_writeData : in rvex_data_type`

- rv2dbg_readData : out rvex_data_type

  Debug interface bus. dbg2rv_readEnable and dbg2rv_writeEnable are active high and should not be active at the same time. rv2dbg_readData is valid one clkEnabled cycle after dbg2rv_readEnable is asserted and contains the data read from dbg2rv_addr as it was while dbg2rv_readEnable was asserted. dbg2rv_writeMask, dbg2rv_writeData and dbg2rv_addr define the write request when dbg2rv_writeEnable is asserted. All input signals are tied to '0' when not specified.

  The debug bus can read from and write to all ρ-VEX registers. 1024 bytes are used per context, thus the size of the debug bus control register block is $1024 \cdot numContexts$ bytes. As the upper address bits are simply ignored, this block is mirrored across the full 32-bit address space.

  The memory map of an ρ-VEX with two contexts is shown in Table 9.1. Note that the mappings per context equal those of direct accesses to the control registers from the ρ-VEX memory units (Section 3.2.4), with the addition of the general purpose registers. Additional contexts specified at design time simply appear after the first two.

  Table 9.1: Debug bus memory map for 2 contexts.

Address	**Mapping**
0x000-0x0FF	Global control registers
0x100-0x1FF	Context 0 general purpose registers
0x200-0x3FF	Context 0 control registers
0x400-0x4FF	Mirror of global control registers
0x500-0x5FF	Context 1 general purpose registers
0x600-0x7FF	Context 1 control registers

9.2.2.8 Trace interface

The trace interface provides an optional write-only bus to some memory system or peripheral, which the core may send trace information to. The trace system is disabled by default and must be enabled in the CR_DCR2 control register. In addition, the trace unit hardware is only instantiated when traceEnable is set in the CFG vector.

- rv2trsink_push : out std_logic

  When high, rv2trsink_data and rv2trsink_end are valid and should be registered in the next cycle where clkEn is high.

- rv2trsink_data : out rvex_byte_type

  Trace data signal. Valid when rv2trsink_push is high.

- rv2trsink_end : out std_logic

When high, this is the last byte of this trace packet. May be used to flush buffers downstream, or may be ignored.

- `trsink2rv_busy :  in std_logic`

  When high while `rv2trsink_push` is high, the trace unit is stalled. While stalled, `rv2trsink_push` will stay high and `rv2trsink_data` and `rv2trsink_end` will remain stable.

9.3 Standalone processing system

The ρ-VEX standalone processing system has the following features.

- Single cycle local instruction memory implemented in block RAMs.
- Local data memory implemented in block RAMs that is single cycle for up to two accesses at a time.
- The initial contents of the local memories can be set.
- Optionally, the cache can be instantiated. In this case, a unified instruction/data memory is instantiated in block RAMs. The access latency of this memory is configurable at runtime to mimic a more realistic memory access latency for cache tests.
- An external bus for peripherals or other memories may be connected through a bus master interface. Without the cache, the ρ-VEX cannot read instructions from this bus, but it can access it using memory operations.
- A slave bus interface allows access to the ρ-VEX debug port, a trace buffer, and the local memories, as well as the cache control register if the cache is instantiated.
- The cache, if instantiated, is coherent only for accesses made by the ρ-VEX itself. A cache flush is required using the cache control register if the debug bus is used to write to the local memories.

9.3.1 Instantiation template

The following listing serves as an instantiation template for the system. The code is documented in the following sections.

If you get errors when instantiating the core with this template, the documentation might be out of date. Fear not, for the signals are also documented in the entity description in `rvsys_standalone.vhd`.

```
library rvex;
use rvex.common_pkg.all;
use rvex.bus_pkg.all;
use rvex.bus_addrConv_pkg.all;
use rvex.core_pkg.all;
use rvex.cache_pkg.all;
use rvex.rvsys_standalone_pkg.all;

-- ...

rvex_standalone_inst: entity rvex.rvsys_standalone
  generic map (

    -- System configuration.
    CFG => rvex_sa_cfg(
      core => rvex_cfg(
        numLanesLog2              => 3,
        numLaneGroupsLog2         => 2,
        numContextsLog2           => 2
        -- ...
      ),
      core_valid => true
      -- ...
    ),
    CORE_ID                     => CORE_ID,
    PLATFORM_TAG                => PLATFORM_TAG,
    MEM_INIT                    => MEM_INIT

  )
  port map (

    -- System control.
    reset                       => reset,
    clk                         => clk,
    clkEn                       => clkEn,

    -- Run control interface.
    rctrl2rv_irq                => rctrl2rv_irq,
    rctrl2rv_irqID              => rctrl2rv_irqID,
    rv2rctrl_irqAck             => rv2rctrl_irqAck,
    rctrl2rv_run                => rctrl2rv_run,
    rv2rctrl_idle               => rv2rctrl_idle,
    rctrl2rv_reset              => rctrl2rv_reset,
    rctrl2rv_resetVect          => rctrl2rv_resetVect,
    rv2rctrl_done               => rv2rctrl_done,

    -- Peripheral interface.
    rvsa2bus                    => rvsa2bus,
    bus2rvsa                    => bus2rvsa,

    -- Debug interface.
    debug2rvsa                  => debug2rvsa,
    rvsa2debug                  => rvsa2debug

  );
```

9.3.2 Interface description

As you can see in the template, the generics and signals are grouped by their function. The following subsections will document each group.

9.3.2.1 System configuration

These generics parameterize the system.

- `CFG : rvex_sa_generic_config_type`

  This generic contains the configuration parameters for the core. `rvex_sa_generic_config_type` is a `record` type with the following members.

  - `core : rvex_generic_config_type`

    This parameter specifies the ρ-VEX core configuration as passed to the bare ρ-VEX processor core. Refer to Section 9.2.2.1 for more information.

  - `cache_enable : boolean`

    This parameter selects whether or not the cache should be instantiated. This is false by default.

  - `cache_config : cache_generic_config_type`

    This parameter specifies the size of the cache blocks. `cache_generic_config_type` is a record type with two `natural`-typed members: `instrCacheLinesLog2` and `dataCacheLinesLog2`. The sizes are determined as follows.

    $$\text{Instr. cache size} = 4 \cdot N_{lanes} \cdot 2^{instrCacheLinesLog2} \cdot N_{laneGroups}$$

    $$\text{Data cache size} = 4 \cdot 2^{dataCacheLinesLog2} \cdot N_{laneGroups}$$

    The number of lane groups is part of the equation because the number of lines are specified per block, and a different block is instantiated for each lane group.

  - `cache_bypassRange : addrRange_type`

    This parameter specifies the range of addresses for which the cache (if instantiated) is bypassed. This range is `0x80000000..0xFFFFFFFF` by default.`addrRange_type` is a record containing four `rvex_address_type` members: `low`, `high`, `mask`, and `match`. An address is considered to be part of the range if the following VHDL expression is true.

    ```
    unsigned(addr and mask) >= unsigned(low) and
    unsigned(addr and mask) <= unsigned(high) and
    std_match(addr, match)
    ```

    This record may be set using the `addrRange` function, which allows parameters to be omitted. The defaults for each parameter specify the complete 32-bit address range, so it is usually sufficient to only set one or two of the parameters.

  - `imemDepthLog2B : natural`

  - `dmemDepthLog2B : natural`

    These parameters specify the sizes of the local instruction and data memories respectively if the cache is not used. Otherwise, `dmemDepthLog2B` specifies the size of the unified memory and `imemDepthLog2B` is ignored. The size is specified as the logarithm of the number of bytes. The default value is 16 for both of these, resulting in 64 kiB memories.

- traceDepthLog2B : natural

  This parameter specifies the size of the trace buffer in the same way that the memory sizes are specified. The default value is 13, resulting in a trace buffer 8 kiB in size. This size is required if the serial debug interface is to be used, due to the way in which bulk data transfers are implemented in the serial protocol.

- debugBusMap_imem : addrRangeAndMapping_type
- debugBusMap_dmem : addrRangeAndMapping_type
- debugBusMap_rvex : addrRangeAndMapping_type
- debugBusMap_trace : addrRangeAndMapping_type

  These parameters specify which addresses on the debug bus are mapped to which device. These parameters may be specified with the addrRangeAndMap function, which takes the same parameters as the addrRange function discussed for cache_bypassRange. In addition, it also allows the designer to change how the address bits are mapped from source to peripheral address. Refer to the comments in bus_addrConv_pkg.vhd for more information.

  By default, the instruction memory is mapped to 0x10000000..0x1FFFFFFF and to 0x30000000..0x3FFFFFFF, the data memory is mapped to 0x20000000..0x3FFFFFFF, the ρ-VEX debug port is mapped to 0xF0000000..0xFFFFFFFF and the trace buffer is mapped to 0xE0000000..0xEFFFFFFF. Note that the range 0x30000000..0x3FFFFFFF maps to both the instruction and data memories. This range allows the instruction and data memory to be written simultaneously, limiting the upload time using the debug unit.

- debugBusMap_mutex : boolean

  This parameter specifies whether logic needed to handle overlaps in the debug bus address map is to be instantiated. If it is set to false, this logic is instantiated, allowing bus write commands to access multiple memories at the same time. This is the default. If it is set to true, overlaps are not supported, but a some area may be saved.

- rvexDataMap_dmem : addrRangeAndMapping_type
- rvexDataMap_bus : addrRangeAndMapping_type

  These parameters specify where data accesses from the ρ-VEX are to be routed. They work the same way as the debugBusMap parameters. By default, the lower half of the address space, 0x00000000..0x7FFFFFFF, is mapped to the data memory, and the remainder is mapped to the bus. Overlaps are not allowed. Accesses made to unmapped addresses cause a bus fault.

Typically, one will want to use the rvex_sa_cfg function to specify this value. This function takes as its arguments values for all the record members as specified above, but has default values for each of them, meaning that not all of them have to be specified. In addition, a base argument of type rvex_generic_config_type may be specified, which will be used as the default value for unspecified parameters. This permits mutation of the `CFG` record as it passes from entity to subentity, which is otherwise impossible to do with record generics.

Important note: in order to allow the function to detect whether the `core` and `cache_config` fields are specified, the `core_valid` and `cache_config_valid` parameters must be set to true, or the defaults will be substituted!

- `CORE_ID : natural`

  This value is used to uniquely identify this core within a multicore platform. It is made available to the programs running on the core and the debug system through `CR_COID`.

- `PLATFORM_TAG : std_logic_vector(55 downto 0)`

  This value is to uniquely identify the platform as a whole. It is intended that this value be generated by the toolchain by hashing the source files and synthesis options. It is made available to the programs running on the core and the debug system through `CR_PTAG`.

- `MEM_INIT : rvex_data_array`

  This value is used to initialize the instruction and data memories. If left unspecified, the memories are initialized to zero.

9.3.2.2 System and run control interfaces

These interfaces are identical to those specified for the bare ρ-VEX core in Sections 9.2.2.2 and 9.2.2.3.

9.3.2.3 Peripheral and debug interfaces

- `rvsa2bus :  out bus_mst2slv_type`

- `bus2rvsa :  in bus_slv2mst_type`

  These signals form a master ρ-VEX bus device, allowing the ρ-VEX to access memory or peripherals outside the processing system. A number of bus interconnection primitives are available in `rvex_rewrite/lib/rvex/bus`. Instantiation of these primitives is beyond the scope of this manual.

- `debug2rvsa :  in bus_mst2slv_type`

- `rvsa2debug :  out bus_slv2mst_type`

  These signals form a slave ρ-VEX bus device, allowing devices outside the processing system, such as the debug serial port peripheral, to access the local memories, trace buffer and the ρ-VEX control registers.

  The memory map of the debug interface is specified using generics. If the cache is instantiated, The cache control register is mapped to the same address as `CR_AFF`. Because `CR_AFF` is read-only and the cache control register is write only, this does not cause conflicts. The cache control register has the following layout.

31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0
LAT		DFL	IFL

LAT field, bits 31..24

Must be written to a value between 1 and 254 inclusive for correct operation. That amount of cycles plus one are added to the bus access delay in case of a cache bypass, write or miss.

DFL field, bits 15..8

Each of these bits corresponds to an ρ-VEX lane group. Writing a one to a bit causes the data cache block corresponding to the indexed lane group to be flushed. Writing a zero has no effect.

IFL field, bits 7..0

Each of these bits corresponds to an ρ-VEX lane group. Writing a one to a bit causes the instruction cache block corresponding to the indexed lane group to be flushed. Writing a zero has no effect.

9.4 GRLIB processing system

The ρ-VEX GRLIB-based processing system has the following features.

- One AHB master interface per ρ-VEX lane group.
- Cache snooping on the AHB bus guarantees cache coherency with other processors and the debug interface sharing the same bus.
- A LEON3 interrupt controller compatible interface is exposed. This allows the ρ-VEX to use the interrupt controller that comes with GRLIB.
- For simulation, an S-record file specifying the expected memory contents can be specified. Every instruction fetch and data access made by the ρ-VEX is snooped and checked against this memory. The memory automatically updates when the ρ-VEX writes a value. Whenever the cache returns an unexpected or inconsistent value, a VHDL warning is printed.

9.4.1 Instantiation template

The following listing serves as an instantiation template for the system. The code is documented in the following sections.

If you get errors when instantiating the core with this template, the documentation might be out of date. Fear not, for the signals are also documented in the entity description in `rvsys_standalone.vhd`.

```
library rvex;
use rvex.common_pkg.all;
use rvex.bus_pkg.all;
use rvex.bus_addrConv_pkg.all;
use rvex.core_pkg.all;
use rvex.cache_pkg.all;
use rvex.rvsys_grlib_pkg.all;

library grlib;
use grlib.amba.all;
use grlib.devices.all;

library gaisler;
use gaisler.leon3.all;

-- ...

rvex_grlib_inst: entity rvex.rvsys_grlib
  generic map (

    -- System configuration.
    CFG => rvex_grlib_cfg(
      core => rvex_cfg(
        numLanesLog2              => 3,
        numLaneGroupsLog2         => 2,
        numContextsLog2           => 2
        -- ...
      ),
      core_valid => true,
      cache => cache_cfg(
        instrCacheLinesLog2       => 18,
        dataCacheLinesLog2        => 18
      ),
      cache_valid => true
    ),
    PLATFORM_TAG                => PLATFORM_TAG,
    AHB_MASTER_INDEX_START      => RVEX_MST_INDEX,
    CHECK_MEM                   => false,
    CHECK_MEM_FILE              => ""
  )
  port map (

    -- System control.
    clki                        => clki,
    rstn                        => rstn,

    -- AHB interface.
    ahbmi                       => ahbmi,
    ahbmo                       => ahbmo_rvex,
    ahbsi                       => ahbsi,

    -- Debug interface.
    bus2dgb                     => bus2dgb,
    dbg2bus                     => dbg2bus,

    -- LEON3 compatible interrupt controller interface.
    irqi                        => irqi,
    irqo                        => irqo
);
```

9.4.2 Interface description

As you can see in the template, the generics and signals are grouped by their function. The following subsections will document each group.

9.4.2.1 System configuration

These generics parameterize the system.

- `CFG : rvex_grlib_generic_config_type`

  This generic contains the configuration parameters for the core. `rvex_grlib_generic_config_type` is a `record` type with the following members.

  - `core :  rvex_generic_config_type`

    This parameter specifies the ρ-VEX core configuration as passed to the bare ρ-VEX processor core. Refer to Section 9.2.2.1 for more information.

  - `cache :  cache_generic_config_type`

    This parameter specifies the size of the cache blocks. `cache_generic_config_type` is a record type with two `natural`-typed members: `instrCacheLinesLog2` and `dataCacheLinesLog2`. The sizes are determined as follows.

    $$\text{Instr. cache size} = 4 \cdot N_{lanes} \cdot 2^{instrCacheLinesLog2} \cdot N_{laneGroups}$$

    $$\text{Data cache size} = 4 \cdot 2^{dataCacheLinesLog2} \cdot N_{laneGroups}$$

    The number of lane groups is part of the equation because the number of lines are specified per block, and a different block is instantiated for each lane group.

  Similar to the bare ρ-VEX and the standalone platform, the `rvex_grlib_cfg` function is available to set this record.

  Important note: in order to allow the function to detect whether the `core` and `cache` fields are specified, the `core_valid` and `cache_valid` parameters must be set to true, or the defaults will be substituted!

- `PLATFORM_TAG : std_logic_vector(55 downto 0)`

  This value is to uniquely identify the platform as a whole. It is intended that this value be generated by the toolchain by hashing the source files and synthesis options. It is made available to the programs running on the core and the debug system through `CR_PTAG`.

- `AHB_MASTER_INDEX_START : natural`

  This value must be set to the AHB master index of the first lane group. The remaining lane groups are mapped to subsequent master indices. In addition, this value is made available to the programs running on the core and the debug system through `CR_COID`, to allow a program to uniquely identify which ρ-VEX it is running on in a multicore system.

- `CHECK_MEM : boolean`

- `CHECK_MEM_FILE : string`

  These parameters configure the simulation-only memory consistency checking system. `CHECK_MEM` enables or disables the system. If the system is enabled, `CHECK_MEM_FILE` must specify the filename of an S-record file holding the initial memory contents. The filename must be relative to the simulator search path.

9.4.2.2 System control interface

The system control signals include the clock source and the reset signal. All registers are rising-edge triggered. The reset signal is active-low to comply with the AHB standard. It is inverted in the system before it is passed to the ρ-VEX logic blocks, which assume an active-high reset signal.

9.4.2.3 AHB interface

- `ahbmi : in ahb_mst_in_type`

- `ahbmo : out ahb_mst_out_vector_type(`*number of lane groups - 1* `downto 0)`

  These signals represent the AHB master interfaces that the cache blocks use as their data source. One master interface is required for each ρ-VEX lane group. The master indices must be a contiguous range, starting at the index specified by the `AHB_MASTER_INDEX_START` generic.

- `ahbsi : in ahb_slv_in_type`

  This signal must be tied to the signal that the AHB interconnect logic broadcasts to all AHB slaves. It is used by the cache to monitor the bus for cache coherence purposes.

9.4.2.4 Debug interface

- `bus2dgb : in bus_mst2slv_type`

- `dbg2bus : out bus_slv2mst_type`

  The debug interface is a slave ρ-VEX bus device. It allows access to all control registers in the system and the trace buffer. It should be connected to the AHB bus using the `ahb2bus` bridge, possibly through additional ρ-VEX bus interconnect primitives. This allows a single AHB to ρ-VEX bus bridge to be used for multiple

ρ-VEX devices, similar to how an APB bridge allows multiple APB peripherals to share a single AHB slave interface.

The ρ-VEX bus primitives are available in `rvex_rewrite/lib/rvex/bus`. Instantiation of these primitives is beyond the scope of this manual.

The debug interface port has a fixed memory map, shown below.

Trace buffer	`0x3FFF`	`0x2000`
ρ-VEX context 7 registers	`0x1FFF`	`0x1D00`
Unused	...	
ρ-VEX context 6 registers	`0x1BFF`	`0x1900`
Unused	...	
ρ-VEX context 5 registers	`0x17FF`	`0x1500`
Unused	...	
ρ-VEX context 4 registers	`0x13FF`	`0x1100`
Unused	...	
ρ-VEX context 3 registers	`0x0FFF`	`0x0D00`
Unused	...	
ρ-VEX context 2 registers	`0x0BFF`	`0x0900`
Unused	...	
Cache control register	`0x0803`	`0x0800`
ρ-VEX context 1 registers	`0x07FF`	`0x0500`
Unused	...	
Reset register	`0x0400`	
ρ-VEX context 0 registers	`0x03FF`	`0x0100`
ρ-VEX global control registers	`0x00FF`	`0x0000`

The reset register resets the entire processing system when written. The only thing that is not reset is the AHB bus bridge to prevent deadlocks due to the AHB bus interconnect not being reset.

The cache control register has the following layout. Note that it is different from the standalone system cache control register.

Offset	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1	0
`0x0800`	IFL	DFL			B

IFL field, bits 7..0

Each of these bits corresponds to an ρ-VEX lane group. Writing a one to a bit causes the instruction cache block corresponding to the indexed lane group to be flushed. Writing a zero has no effect. The register always reads as 0.

DFL field, bits 15..8

Each of these bits corresponds to an ρ-VEX lane group. Writing a one to a bit causes the data cache block corresponding to the indexed lane group to be flushed. Writing a zero has no effect. The register always reads as 0.

B flag, bit 0

When this bit is set, the data cache is always bypassed. When it is cleared, the cache is only bypassed for memory accesses to the upper half of the address space, i.e. `0x80000000..0xFFFFFFFF`. The flag resets to 0.

9.4.2.5 Interrupt controller interface

- `irqi : in irq_in_vector(0 to` *number of contexts - 1*`)`
- `irqo : out irq_out_vector(0 to` *number of contexts - 1*`)`

  These signals should be tied to a GRLIB `irqmp` interrupt controller, with the number of processors set to the number of ρ-VEX contexts.

Bibliography

[1] J. van Straten, "A Dynamically Reconfigurable VLIW Processor and Cache Design with Precise Trap and Debug Support," Master's thesis, Delft University of Technology, the Netherlands, 2016.

[2] A. Brandon and S. Wong, "Support for Dynamic Issue Width in VLIW Processors using Generic Binaries," in *Proc. Design, Automation & Test in Europe Conference & Exhibition*, (Grenoble, France), pp. 827 – 832, March 2013.

[3] J. A. Fisher, P. Faraboschi, and C. Young, *Embedded Computing.* Morgan Kaufmann, 2005.

[4] Hewlett-Packard Company, "HP Labs : Downloads: VEX." Available: `http://www.hpl.hp.com/downloads/vex/` (visited on Feb. 12, 2016).

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

[6] STMicroelectronics, "ST200 Micro Toolset Releases." Available: `http://ftp.stlinux.com/pub/tools/products/st200tools/index.htm` (visited on Feb. 12, 2016).

[7] R. Seedorf, "Fingerprint Verification on the VEX Processor," Master's thesis, Delft University of Technology, the Netherlands, 2010.

[8] H. van der Wijst, "An Accelerator based on the ρ-VEX Processor: an Exploration using OpenCL," Master's thesis, Delft University of Technology, the Netherlands, 2015.